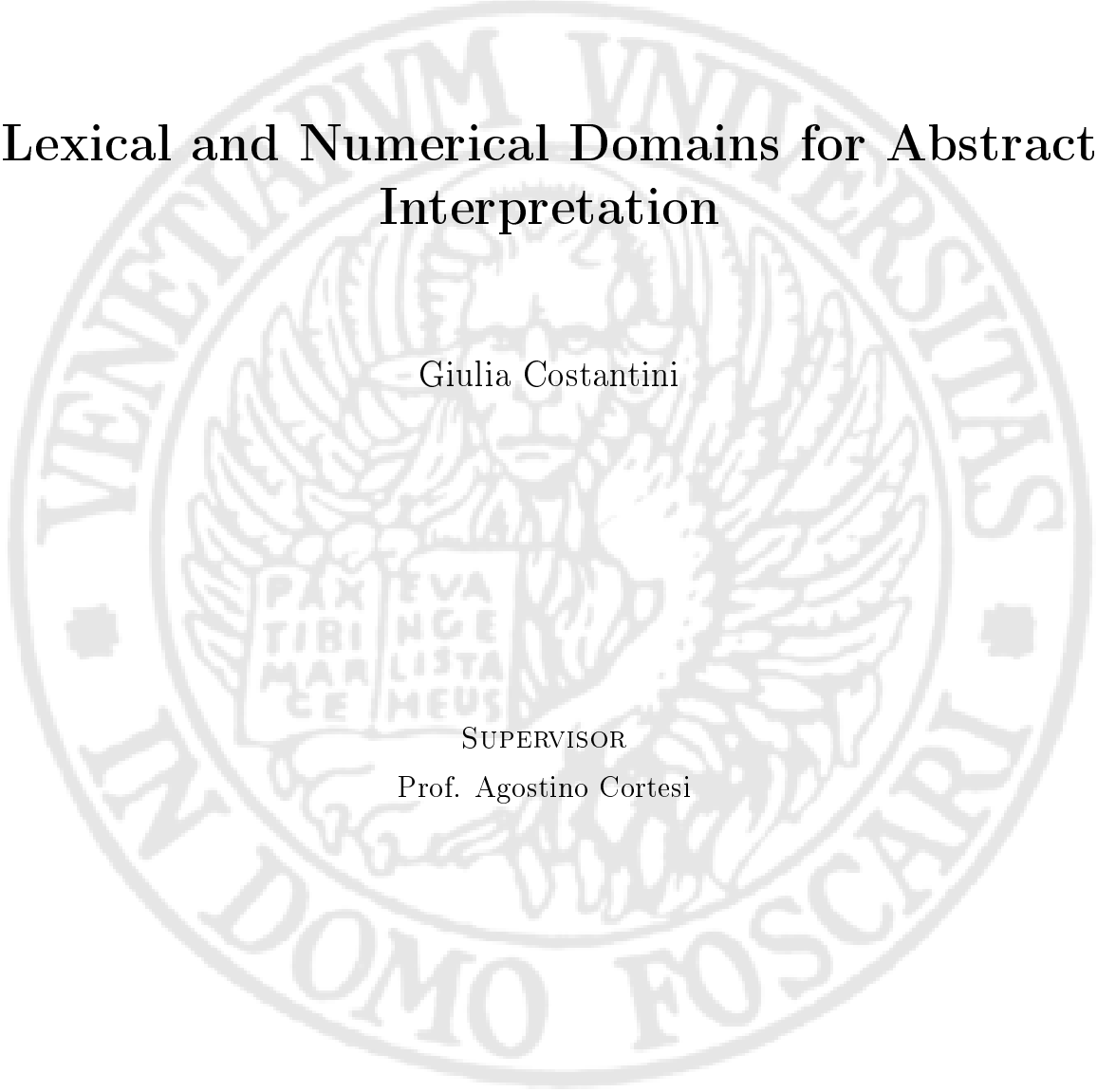
Università Ca' Foscari di Venezia

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis: 955835

Lexical and Numerical Domains for Abstract Interpretation

Giulia Costantini



Supervisor
Prof. Agostino Cortesi

Academic Year 2012/2013

Author's e-mail: malvoria@gmail.com

Author's address:

Dipartimento di Informatica
Università Ca' Foscari di Venezia
Via Torino, 155
30172 Venezia Mestre – Italia
tel. +39 041 2348411
fax. +39 041 2348419
web: http://www.dsi.unive.it

To Rebecca, the light of my eyes

Abstract

The goal of this thesis is to contribute to the field of formal methods employed for the static verification of computer program properties. The context is the Abstract Interpretation framework, one of the various possible techniques to perform static analyses. In particular, we focus on the design of novel abstract domains to analyze the basic building blocks of computer programs: lexical and numerical variables, as well as relationships between variables.

We start by considering lexical variables, i.e. strings, which have become very important in the last years because of the spread of programs building HTML pages, dynamically creating SQL queries or XML documents and so on. Ensuring the security of such applications is fundamental. However, most of the existing approaches to string analysis are specific in one way or another (for example, they can only verify one property, or they can be used within only one language, etc.). We build a highly generic framework to approximate string values. We do this by defining a suite of abstract domains not focused on a specific language or property. The domains that we design are very different in terms of precision and performance: in particular, three of them are simple and can only verify basic properties but they have a low computational cost. The other two domains are more complex and costly, but they are also very precise. The user of the framework can then choose the most appropriate domain to use for the specific analysis to perform, based on the computational constraints and the kind of property to be verified.

Then, we move to numerical domains. We focus on a specific field, i.e. that of hybrid systems, with the purpose of improving the precision of an existing abstraction, the IVSF domain [24]. Such domain abstracts the inputs of hybrid systems coming from the environment through interval-valued step functions. This abstraction is rough because each step of the abstract function consists of a simple fixed interval. We define a new domain where the abstraction inside each step is made by two linear functions, allowing for a greater degree of precision. We also introduce a sound abstraction function for both domains (IVSF and our new domain, TSF) and we give the abstract semantics of arithmetic operations over continuous functions, to allow for possible future applications of TSF outside hybrid systems.

Finally, we consider the problem of analyzing programs where variables are strongly related. The two main existing approaches to this issue are relational abstract domains and power-set extensions. Both approaches share strong perfor-

mance limits (especially the power-set extensions) reducing their practical applicability. Taking inspiration from the practical field of physics simulations inside computer games software, we define a novel parametric abstract domain which approximates the whole state of the program (i.e., the values of all variables) through a tuple of abstract values (coming from a base abstract domain), one for each variable. An abstract element of our domain is made by a set of such tuples, tracking a disjunctive kind of information. The name of the domain (Parametric Hypercubes) outlines the idea that each tuple represents an hypercube in the space of variable values: an abstract state is composed by a set of hypercubes. The parametric nature of the domain makes it flexible and usable in various contexts, not only physics simulations.

In order to provide experimental evidence of their actual applicability, we implemented our domains and we applied them to a suite of case studies. The framework for string analysis has shown the trade-off between performance and precision of the various domains, ranging from quick and rough to slower but more precise analyses. The most precise domain of the framework yields to optimal results in a challenging case study which required the use of the widening because of a loop with unknown condition. The improved version of IVSF (TSF) has indeed shown to be more precise: we applied our domain to the same case study used in the presentation of IVSF and verified that we reach better results without affecting the performance of the computation. Finally, the domain of Parametric Hypercubes gave very encouraging results when applied to a small but significant case study coming from games software (a bouncing ball) and was effective when applied to a generic context as well.

Acknowledgments

I always said I would have skipped the *acknowledgments* part of my thesis, because I believe that the people I want to thank already know how much I am indebted to them. However, now that I actually find myself at the end of my Ph.D., I think that they deserve to have a written mention (because *verba volant*... but *scripta manent*!). So, here we are:

To Tino, my professor and advisor: thank you because you taught me many things, which not only apply to Computer Science but, more importantly, to life in general. You consistently encouraged me to aim high. You showed me how to see a glass half-full, instead of half-empty. Each time I knocked to your door, I knew I would find a welcoming smile and a positive attitude. Thank you also because, during the course of my Ph.D., you always had my best interests at heart.

To Pietro: thank you for your never-ending support. Your continuous presence (even if from long-distance and outside your strict academic duties) has made this journey much easier and enjoyable. Questions, advices, clarifications... you were there for anything I needed! I think very highly of you, both professionally and personally. For me, you have not only been a supervisor, but also a friend.

To my parents (these days better known as "nonno Aldo" and "nonny"), the two people I know I can count on, ever. Your love is so big that you stand by side even when I'm wrong; you put up with my bad tempers; when I'm in need, you do everything in your powers to help me. You are great parents and wonderful grandparents. Thank you for always being there. I really hope to have made you proud of me.

To Eppe, my husband, my best friend, the father of my child, and my biggest supporter: thank you for believing in me even when I didn't, and for pushing me to reach my potential. We grew together: now and then, the road had its bumps, but I hope we will continue sharing the journey of life side by side, because we are much stronger united than apart. Thank you because, if it weren't for your insightful stubbornness, we wouldn't be where we are now... a pretty damn good place to be! Dank je wel, mijn liefde.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Computer programs often contain errors. Executing a bugged program can lead to failures with catastrophic consequences, especially when the program manipulates important data (think about databases containing sensitive information) or when it is used to control electronic instrumentation (like in avionics or medical devices). Two extreme examples are the overflow bug that caused the failure of the Ariane 5 launcher in 1996 [7] resulting in the self-destruction of the rocket 37 seconds after launch, and the cumulated imprecision errors in a Patriot missile defense system that caused it to miss its Scud missile target, resulting in 28 people being killed in 1992 [2].

For this kind of programs, testing is not enough, since it is only able to expose a finite subset of all the traces (i.e., possible executions) of the program. When stronger guarantees about the correctness of a program are needed, static analysis must be used. Static analysis is suite of different techniques sharing a common approach to soundly verify specific properties of programs, by approximating all their traces. In the last years, the commercial application of static analysis tools has grown considerably, especially in the verification of properties of software used in safety-critical computer systems and to locate potentially vulnerable code (Section 7.3 of [113] cites many of such applications). Three examples of this growth come from the medical, nuclear and embedded software industries: (i) the U.S. Food and Drug Administration (FDA) has identified the use of static analysis for medical devices as means to improve the quality of their software [6]; (ii) the UK Health and Safety Executive recommends the use of static analysis on Reactor Protection Systems [4]; and (iii) a recent study by VDC Research reports that 28.7% of the embedded software engineers surveyed currently use static analysis tools and 39.7% expect to use them within 2 years [3].

In this thesis, we wish to *contribute to the field of formal methods used in the verification of the correctness of programs.* In particular, the focus of this thesis will be the *creation of novel abstract domains to improve the analyses of lexical and numerical properties of programs within the Abstract Interpretation framework.* In Section 1.2 we intuitively describe how Abstract Interpretation works and the variety of existing tools (static analyzers) used to put such technique into practice, i.e. to automatically analyze programs. In Section 1.3 we do a brief survey of the

objects manipulated by programs and the existing techniques to deal with their static analysis. Based on these findings, in Section 1.4 we pinpoint exactly some areas where we see the possibility of improvements and we explain in detail which goals we expect to reach with this thesis. In Section 1.5 we recap the results obtained through our research and how they respond to the objectives established in Section 1.4. In Section 1.6 we explain how this thesis is structured.

1.2 Methodology

To perform static analysis on a program, one has to choose between various specific techniques, the main ones being dataflow analysis, control flow analysis, model checking, program verification and Abstract Interpretation [132].

Abstract Interpretation [55] is a mathematical theory that allows one to build up completely automatic static analyses that may apply directly on the source code. This is not the case of model checking, another static analysis approach, as it requires a model of the program (usually a Kripke structure) provided by the user as an input. It also differs from theorem proving techniques, as they often require an interaction with a specialized user (i.e., someone that thoroughly knows how the theorem prover works) to generate the proofs. During the last years, Abstract Interpretation has been widely used and also successfully applied to some important scenarios [18]. For these reasons, we choose to work inside the Abstract Interpretation framework.

1.2.1 Abstract Interpretation

Abstract Interpretation is a mathematical theory of approximation of semantics developed by Patrick Cousot and Radhia Cousot in 1977 [53, 57], invented in order to deal systematically with abstractions and approximations. Applied to static analysis of programs, Abstract Interpretation allows to approximate an uncomputable concrete semantics with a computable abstract one. The approximation makes the result correct but incomplete. In fact, the inferred properties are satisfied by all the possible results of the concrete semantics, but if a property is not inferred in the abstract semantics it may still be satisfied by the concrete one. The main idea of Abstract Interpretation is to define the semantics of a program as the fixpoint of a monotonic function.

We can informally generalize the basic idea of Abstract Interpretation as follows:

- The starting point is the concrete domain together with the concrete semantics. The **concrete domain** is used to define a formal description of a computation state, while the **concrete semantics** is a function defined on the concrete domain, function which associates a meaning to program statements. The concrete semantics is the most precise mathematical expression of the behaviour of a program.

- An **abstract domain** has to be determined, which is an approximation of the concrete domain. An abstract domain models *some* properties of the concrete computations, leaving out superfluous information.
- From the abstract domain we can derive an **abstract semantics** and prove its correctness. The abstract semantics allows us to execute the program on the abstract domain in order to compute the properties modelled by such domain.
- Applying a **fixpoint algorithm** we can statically compute a concrete approximation of the concrete semantics. If the abstract domain chosen for the analysis respects a certain condition (ACC; see Chapter 2 for more details), the abstract semantics can be computed in a finite time. Otherwise, we need a widening operator in order to make the analysis convergent, still obtaining sound (even if more approximated) results.

This approach is schematically depicted in Figure 1.1, where α represents the abstraction function (which maps concrete values into abstract ones), γ represents the concretization function (which maps abstract values into concrete ones), f represents the concrete semantics and $f^{\#}$ represents the abstract semantics.

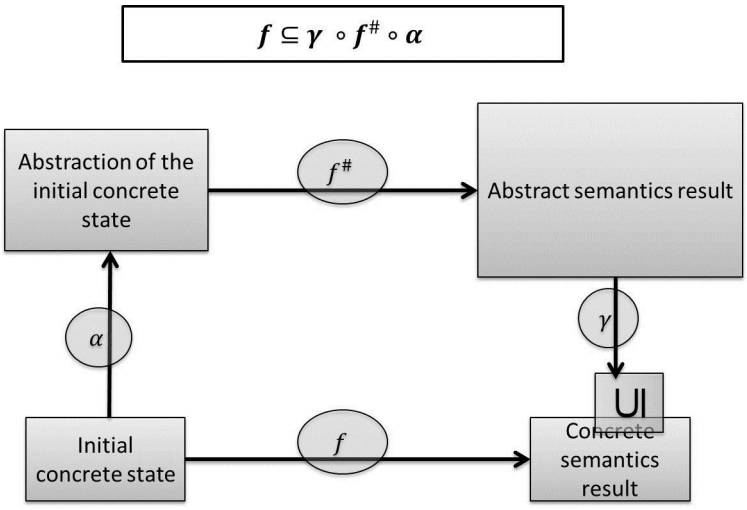


Figure 1.1: Abstract Interpretation analysis

Abstract Interpretation, then, is a general theory which formalizes the notion of **approximation**. Such approximation can be more or less precise, depending on the considered observation level. The most important concept in Abstract Interpretation is that of *abstract domain*, i.e., a class of properties together with a set of operators to manipulate them. Each abstract domain embeds some sort of approximation and it does not exist a single, all-purposes abstract domain. Various factors influence the choice of the abstract domain to use during a specific analysis, for example the

kind of property to verify, the language constructs used in the analyzed program, the amount of available computing resources, and so on. However, a single abstract domain can generally be used in more than one context. Creating a new abstract domain for each new specific application is not feasible in the long run, since it would produce an excessive proliferation: abstract domains should, ideally, be as reusable as possible. One of the big challenges of Abstract Interpretation is exactly to create abstract domains which can be used in as many contexts as possible, while retaining a satisfactory level of precision and performance. Obviously, such domains can be defined from scratch, or by improving (in terms of performance or precision) existing ones.

1.2.2 Static Analyzers

Abstract Interpretation allows, among other applications, the design of static analyses that are sound by construction. Many generic static analyzers based on Abstract Interpretation have been proposed in the recent years. These static analyzers support the use of different domains (in order to obtain faster and more approximated or slower and more refined analyses) and they can analyze different properties. The main advantage of this approach is that the most part of an analyzer can be reused to analyze different properties, and tuned at different levels of efficiency and precision through approximation and refinement. A brief overview of some known static analyzers based on the Abstract Interpretation framework follows:

Astrée is a static program analyzer aiming at proving the absence of Run Time Errors (RTE) in programs written in the C programming language. Astrée analyzes structured C programs, with complex memory usages, but without dynamic memory allocation and recursion. This encompasses many embedded programs as found in earth transportation, nuclear energy, medical instrumentation, aeronautic, and aerospace applications, in particular synchronous control/command such as flight controls [65, 140], or space vehicle manoeuvres [21].

Polyspace is a static code analysis tool inspired by the failure of the maiden flight of Ariane 5 where a run time error resulted in destruction of the launch vehicle. It is the first example of large-scale static code analysis by Abstract Interpretation to detect and prove the absence of certain run-time errors in source code for the C, C++, and Ada programming languages. Polyspace also checks source code for adherence to MISRA C and other related code standards [66].

Fluctuat and HybridFluctuat Fluctuat's features include the static analysis of C and ADA programs, sensitivity analysis of program variables, worst-case generation, interactive analysis and analysis of hybrid systems. HybridFluctuat [22] extends the static analysis of embedded programs by considering the physical environment in which they are executed. The analyzer considers

programs written in C-ANSI and ordinary differential equations presented as C++ functions. The tool then automatically derives invariants on the whole system.

Apron is a library dedicated to the static analysis of the numerical variables of a program by Abstract Interpretation. Such library is intended to be a common interface to various underlying abstract domains and to provide additional services that can be implemented independently from the underlying library/abstract domain. Currently, APRON provides C, Java and OCaml interfaces.

Julia is a software tool [141] which performs the static analysis of full sequential Java bytecode. This tool is generic in the sense that no specific abstract domain (analysis) is embedded in it. Instead, abstract domains are provided as external classes that specialise the behaviour of JULIA. Static analysis is performed through a denotational fixpoint calculation, focused on some program points called watchpoints (which can be automatically placed by the abstract domain or manually provided by the user). JULIA can be instructed to include a given set of Java library classes in the analysis, in order to improve its precision. Moreover, it gives abstract domains the opportunity to soundly approximate control and data-flow arising from exceptions and subroutines.

Sample [1] (Static Analyzer of Multiple Programming LanguagEs) [72, 73, 75, 154] is a generic analyzer based on the Abstract Interpretation theory. Sample can be composed with different heap abstractions, approximations of value information (e.g., numeric domains or information flow), properties of interest, and languages. Several heap analyses, value and numerical domains have been already plugged. The analyzer works on an intermediate language called Simple. Up to now, Sample supports the compilation of Scala, Java bytecode and Touchdevelop code to Simple.

Given the wide choice of analyzers available, we can rely on an existing one instead of creating one from scratch, thus being able to focus only on the definition of abstract domains which will be plugged in the chosen analyzer. We decided to use Sample because it provides a good combination of accessibility (for example, Polyspace is a commercial tool), genericity and flexibility (two important requirements since we want to deal with different kinds of applications). Figure 1.2 depicts the structure of Sample. However, our contribution does not depend on the specific analyzer. We will build new *abstractions* to improve the static analysis of programs: the implementation of such abstractions is linked to Sample, but their mathematical structures are completely "analyzer-independent".

[1] `http://www.pm.inf.ethz.ch/research/semper/Sample`

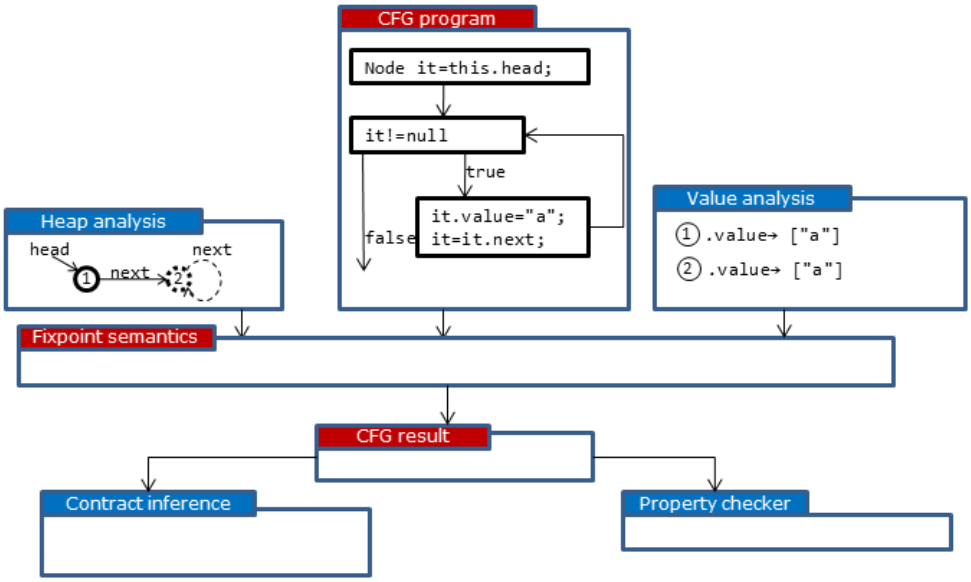


Figure 1.2: The structure of Sample

1.3 Context

Abstract Interpretation focuses on abstracting the data and operations of programs. Programs *manipulate data* (i.e., values), and the operations that you can execute on values strictly depend on their type. The most basic data types are often referred to as *primitive data types* and they are the building blocks for more complex code. Even though the actual range of primitive data types that are available is dependent upon the specific programming language that is being used, the most classic primitive data types are common to almost all languages. They are: **numbers** (integers and floating point), **characters** and **boolean** [2]. Programs are then mainly built by manipulating two basic building blocks: *text* (in the form of characters and their composition, i.e. strings) and *numbers*. We will now analyze in more details the existing approaches to the static analysis of these two data types, to find out possible limitations and opportunities for improvements. Moreover, since the variables of a program almost always interact with each other, we will also reason about relationships between them and explore how this issue has been dealt with until now in Abstract Interpretation.

[2]For example, in Java there are eight primitive data types: int, long, short, byte, double, float, char, boolean.

1.3.1 Lexical Variables

In computer and machine-based telecommunications terminology, a *character* is a unit of information that roughly corresponds to a grapheme, grapheme-like unit, or symbol, such as in an alphabet or syllabary in the written form of a natural language. Examples of characters include letters, numerical digits, common punctuation marks (such as "." or "-"), and whitespace. Characters are typically combined into *strings*, an important and useful datatype which is implemented in nearly every programming language (and in some of them it is also considered a primitive data type). In computer programming, a string is traditionally a sequence of characters. In formal languages, which are used in mathematical logic and theoretical computer science, a string is a finite sequence of symbols that are chosen from a set called an alphabet. The connection between the two definitions is that a string data type is a data type modelled on the idea of a formal string.

In the last years, the use of strings in programs has become widespread: SQL queries, XML documents, HTML pages, reflection and so on are part of many programs. It has become crucial to provide some way to analyze string-manipulating programs, in order to guarantee at compile time that some negative scenarios will never happen. In fact, a lot of approaches to string analysis have been developed through the years, as the use of type systems [99, 100, 110], automata and context-free grammars [35, 98, 144, 150], and abstract domains [33, 109]. What characterizes most of these approaches is their *specificity*: usually, they can be used only on a small subset of all the possible verification problems that can arise in string analysis. Some are focused on a specific application domain (like the generation of XML documents), some are not fully automatic (that is, they may require manual annotations), some others support only few string operations. Also, for most of these approaches the level of precision is completely fixed and not tunable depending on the specific problem to tackle: in some cases, the analysis could be too precise and slow, while in some others it could be unable to verify the property of interest.

1.3.2 Numerical Variables

In the past 30 years, the analysis of numerical properties of programs has been intensively investigated, and a broad spectrum of abstractions has been developed based upon results from linear algebra, arithmetic [87], linear programming [62] or graph theory [123]. Some examples of numerical abstract domains found in the literature are Polyhedra [31], Octagons [123], Octahedrons [38], TVPI [139], linear (in)equalities [62], congruences [86, 87], affine relationships [108], and so on. Each of these domains defines a representation for a set of properties of interest and algorithms to manipulate them. They vary in expressiveness and in the cost/precision trade-off. In general, the scalability of algorithms which discover complex program invariants is limited. However, techniques that partition the variables of the program in small-size packets that are tractable by these algorithms have been recently

applied to the verification of large safety-critical software with success [18, 148]. For floating-point variables, there exist several methods for bounding the final error of a program, including interval arithmetic, forward and backward analysis [97, 149], or Abstract Interpretation-based static analysis (that includes Astrée [60, 128] and Fluctuat [64, 84]).

1.3.3 Relationships Between Variables

Regardless of their type (numerical, lexical, boolean, etc.), the variables of a program can have relationships between them: often it happens that the value of one variable strongly influences the life of other variables. When such relationships are weak, the precision of the result is not negatively affected by considering each variable on its own. However, when dealing with programs with strongly inter-related variables, the abstractions which consider separately each variable can suffer a lot in terms of precision.

In Abstract Interpretation, the usual approach to deal with this issue is to use *relational domains*. Relational domains are able to express relationships between variables, that is, arithmetic properties involving several variables at a time, such as $\mathtt{x} = \mathtt{y} + \mathtt{z}$. Some examples of relational numerical abstract domains are: convex polyhedra [62], octagons [123], difference-bound matrices [121], linear equalities [108], and combinations thereof (like [32, 38, 71, 120, 139] and many others). The convexity of such domains is the key of a tractable analysis of software for buffer overflows, null pointer dereferences and floating point errors. The problem is that classical relational domains are more precise but also *much more costly* than non-relational ones, compromising their applicability in practical settings. Another problem of classical relational domains (like octagons or polyhedra) arises from their convexity: their abstract union operator is an imprecise over-approximation of the concrete union, because (in the case of non-convex sets of values) it introduces spurious values in the analysis. Convexity, then, causes the analysis to fail in many common cases because it limits the ability of these domains to represent sets of states. The classical example is an $\mathtt{if} - \mathtt{then} - \mathtt{else}$ statement which (based on some unknown condition) assigns 1 or -1 to an integer variable (let us call it `x`). After computing the least upper bound of the two possible flow paths, convex domains infer the information that $[-1, 1]$ is the interval of possible values of `x`, thus including also the 0 value (which `x` can never assume in the concrete semantics).

The simplest way to solve this problem is to introduce disjunction in the analysis, in the form of *power-set extension* [55, 80]. The power-set extension of an abstract domain refines the abstract domain by adding elements to represent disjunctions precisely. The presence of disjunction helps in removing the convexity limitation, but it also increases the complexity of the analysis: power-set domains can be exponentially more expensive compared to the base domain, due to the large number of disjuncts that can be produced during the analysis. Furthermore, special techniques to lift the widening from the base domain up to the disjunctive domain [16]

are needed. It is fundamental to manage the high complexity of power-set extensions by controlling the production of disjuncts during the analysis and avoiding their excessive proliferation. One well-known approach consists in syntactically bounding the number of disjuncts: the challenge in this case becomes the choice of which disjuncts to merge (to limit as much as possible the loss of precision). For example, some path sensitive data-flow analysis techniques implicitly manage complexity by joining abstractions only when the property to be proved remains unchanged as a result [63], or "semantically" by careful domain construction [16, 116]. Another technique for disjunctive static analysis has been proposed and implemented in [137]: this analysis is formulated for a generic numerical domain, and an heuristic function based on the Hausdorff distance is used to merge related disjuncts. Finally, another known approach is trace partitioning [94, 136], a technique which annotates control flow objects by partial trace information. This idea was first introduced in [94], but still their proposal was not practical, especially for large programs, while the framework proposed in [136] is more general and flexible. In this framework, a token representing some conditions on the execution flow is attached to a disjunct, and formulae with similar tokens are merged together. In this way, each element of the disjunction is related to some property about the history of concrete computations, such as "which branch of the conditional was taken". The choice of the relevant partitioning is a rather difficult and crucial point. In practice, it can be necessary to make this choice at analysis time.

1.4 Objectives

The purpose of this thesis is *the study of new domains (both lexical and numerical) to effectively contribute to the automatic verification of software through the Abstract Interpretation framework.*

Note that an abstract domain can be designed following two possible methods:

- choosing a specific problem to address and creating a domain especially calibrated for it. The advantage of this method is that the resulting domain should give very good result on the problem at hand, but it could be totally useless in other contexts. Obviously, it is not feasible to create a new domain for every kind of verification problem, so the ideal result of this process would be to obtain a domain which is usable also in other settings than the specific one for which it was originally conceived.

- choosing the object to abstract (i.e., a certain data type) and creating an abstraction for it which is as more generic as possible, i.e. with no specific application in mind. The generality of this approach is at the same time a strength and a weakness: one must be careful to not abstract too much, otherwise the risk is that there is no practical context where the inferred information is sufficient to verify interesting properties.

For each of the three building blocks of programs pinpointed above (lexical variables, numerical variables, relationships between variables), we are going to follow a different approach, based on what already exists in that context (in terms of abstract domains or other static verification techniques) and on the goal that we want to reach. In particular:

1. In the case of lexical variables, in Section 1.3 we hinted at the various existing approaches to string analysis, a topic which has gained more and more importance together with the diffusion of web technologies and databases. However, almost all of these approaches tend to be quite specific. For example, some verify the consistency of a generated document with respect to a specific standard (HTML, XML, etc.) and others are related to a specific programming language and its constructs (Java, PHP, etc.). Also, other approaches impose limitations on the kind of program that can be analyzed and others are not fully automatic, requiring the intervention of the user of the analysis. Finally, some approaches suffer from a potentially very high complexity, while others are not precise enough to analyze non-trivial code.

   What we think is missing in string analysis is a unifying approach, which is able to deal with many kind of programs and properties, and which trade-off between precision and complexity can be tuned to the desired level. Our purpose is to fill this gap, by creating an abstraction that overcomes these problems and limitations. We are then going to follow the second method between the two explained before: choosing the object to abstract (the string data type) and trying to create a highly generic and scalable abstraction.

2. In the case of numerical values, there already exist a lot of well functioning approaches for their analysis (as shown in Section 1.3). In fact, numerical values have always been the core of any program, and almost every application could benefit from numerical static analysis. However, exactly because of the wide range of types of programs manipulating numbers, it is certainly useful to create specialized abstract domains, which are able to better understand and exploit the specific features of the chosen application domain (instead of using simple and totally generic domains like *Intervals*). To contribute to numerical analysis, then, we use the first method between the two explained above: we choose a specific problem to address and we create an abstraction tailored for that problem.

   We choose the application field of *hybrid systems*, which has emerged and has gained a lot of importance in the last years. A hybrid system is a dynamic system that exhibits both continuous and discrete behaviour. We will talk in more detail about hybrid systems in Chapter 4: for now note that, in order to successfully analyze this kind of programs, the analysis techniques must take into account the program interaction with the external world. One of the most important interactions to consider is the one between the program and

the physical environment [24, 52, 83]. The simplest abstraction of these interactions consists in bounding both inputs and outputs within intervals. This abstraction is sound but very rough, because it does not consider that a continuous variable cannot jump discretely from a value to another (for example, from the minimum value of the abstracted interval to the maximum one). A more precise abstraction should take into account this fact, and consider the behaviour of a continuous variable as that of a continuous univariate function, where the input variable is the time. The first step in this direction was taken by Bouissou and Martel in 2008 [24], when they proposed a new approach to abstract continuous functions. They presented an abstract domain called IVSF (Interval Valued Step Functions), which basic idea is to partition the timeline into (not necessarily regular) steps, and then to choose for each step an over- and under-approximation of the function on this step. This approximation *inside* each step is still very rough, since it is represented by a simple interval, fixed throughout all the step. Our purpose is to improve the abstraction of IVSF.

3. In the case of relationships between variables, two main approaches exist to keep them into account, as explained in Section 1.3: relational domains or power-set extensions (i.e., disjunction). On the one hand, the problems of relational domains are the (high) complexity, the fixed kind of relationships that they can track (i.e., $x < y + z$) and the convexity of the representation (which can decrease the precision of the result in certain cases). On the other hand, the problem of power-set extensions is mainly their (very high) complexity. Another important approach (trace partitioning) is based on the partitioning choice, which could be difficult to make for the user.

   Then, we set as our goal to find a new approach to the analysis of programs where variables are inter-related, trying to reduce the limitations of the other existing approaches. We also choose a specific application domain to which apply our abstraction. The inspiration for this part of our work, in fact, has come from a specific application field, where static analysis is still not effectively applied: physics simulations within games software. Nowadays the video-games industry has a great role in the economy, and the impact of reducing the testing and debugging time in video-game development could be very big. Almost every game contains some physics and the variables of physics simulations are always strongly inter-related: for these reasons, such field is a very interesting workbench for our purpose. Moreover, physics simulations are widely used also outside the game industry, so our result could hopefully be useful in many other applications.

1.5 Contributions

Our main goal during this thesis research has been to provide new advanced abstract domains that are both firmly grounded mathematically and of practical interest. As explained in more detail in Sections 1.3 and 1.4, we found three venues where we judged there was space for improvements. In particular, we set the following goals:

1. the creation of a unifying approach for string analysis, tunable in terms of precision and performance;

2. the improvement of the existing abstraction IVSF to consider the continuous inputs of hybrid systems;

3. the creation of a new approach to deal with relationships between variables, specifically tailored to keep into account the peculiarities of physics simulations within games software.

Each of these three contributions can be conceived in two layered levels. Firstly, we give theoretical results: mathematical definitions, algorithms, and theorems. Secondly, most of these results have been implemented and plugged inside the `Sample` static analyzer and tested on small program fragments. In particular, our contributions are the following ones:

1. We created a framework for string analysis which comprises five different abstract domains. In fact, creating a single abstract domain generic enough to support precisely the analysis of different kind of programs and properties is a difficult (if not impossible) task. For this reason, we thought that the best solution to reach our goal would have been to create many domains, each one with a different level of precision (and therefore performance). In this way, we offer various abstractions (inside a unified framework) and the user can choose the most appropriate depending on the specific verification problem to be solved. For each domain we provided the semantics of some string operators (the most commonly used when dealing with strings and which are present in any language supporting strings). After plugging the implementation of our domains into Sample, we tested the framework on some case studies. The experimental results are satisfactory and confirm the different levels of precision and performance of the five domains. Note that the framework can be easily extended to other string operators, by simply defining (and then implementing) their abstract semantics on the five domains.

2. We created a new abstraction for univariate continuous functions. In fact, the idea of [24] to represent the values of the inputs of a hybrid system through a step-function depending on the time (IVSF domain) is certainly insightful. However, the abstraction contained in each step of IVSF is too rough, since it is simply a fixed interval for each time instant of that step. We decided to exploit

their idea and to go one step forward: in particular, we kept the underlining definition of the step-function but we modified the abstraction inside each step. In our abstraction, the possible values assumed by the function inside a single step belong to the area enclosed by two lines which (instead of being horizontal as in IVSF) are sloped. Geometrically, the shape of this area for a single step resembles a trapezoid, from which comes the name of our novel abstract domain TSF (Trapezoid Step Functions). The implementation of the domain has shown that we indeed improved the precision of IVSF as it was our goal: TSF is a strictly more precise (*and* still efficient) approximation of continuous functions with respect to IVSF. Also, we designed a sound abstraction function which, given a continuous function (respecting certain conditions), creates its abstraction in both IVSF and TSF. Finally, our result opens the door to new applications and research directions, for example the one of cost analysis. To this end, we already defined the abstract semantics of arithmetic operators on the domain and we are now currently working to extend the domain definition to work with bivariate functions, to allow for more complex cost functions.

3. We defined a new approach to consider relationships (both implicit or explicit) between the variables of a program. We did this by creating a disjunctive non-convex non-relational domain, generic with respect to the type of variables of the program (since it is parametric on a base abstract domain).

   Instead of defining an abstraction for the values assumed by single variables (as it is usually the case in Abstract Interpretation: for example, numerical domains abstract the value of a single numeric variable), we defined a structure which abstracts the *tuple* of values which compose the whole state of the program (i.e., one value for each variable). Each abstract tuple can then be seen as a valid set of concrete states of the program. The non-convexity comes from the disjunctive nature of the domain, since an abstract element is made by a *set* of abstract tuples. The parametric nature of the domain, instead, comes from the fact that we did not fix *a-priori* the way in which the single elements of the tuples are abstracted (i.e., how the single variables are approximated). However, since we wanted to focus on game physics simulations (where variables are floating point values), we also defined a novel specific abstraction for such values, exploiting the basic concept of *Intervals* mixed with fixed partitioning of the value space (to improve performance) and with offsets inside the interval (to improve precision).

   We applied this domain (called *Parametric Hypercubes* because each tuple of values resembles an hypercube in the variables values space) to a small but significant case study coming from games (a bouncing ball, both in 2D and 3D) and the results have been encouraging, both in terms of performance and precision. Regarding the practical applicability of our domain, note that: (i) physics simulations are not only an important part of games software but

are also the basic constituent of many other applications (hybrid systems, ballistics, weather forecasts, environmental simulations and so on), meaning that this domain could potentially be useful in such fields as well; (ii) the parametric nature of the domain gives the user great freedom in specialising it, making it suitable for completely different kind of applications than physics simulations, as shown in Section 5.10.

1.6 Thesis Overview

The rest of the thesis is structured in five chapters. Each chapter begins with an introductory paragraph and the list of its contents. The three central chapters (Chapters 3, 4, 5) contain our contributions. In all three cases, we designed new abstract domains which solve a particular verification problem, as explained in Sections 1.4 and 1.5. For this reason, the three chapters follow the same general structure, that is:

- *Introduction*, a section where the problem tackled by the chapter is informally introduced and the existing literature on the subject is briefly presented. In particular, this section will usually start with an extensive treatment of the general *context* to which the problem belongs, followed by a recap of the *state-of-the-art* and our *contribution* to such field.
- *Case Study*, a section which explains the case study (one or more) chosen to exhibit the features of our proposed solution to the problem.
- *Notation*, a section where we define specific notation for the problem at hand (if necessary) in addition to the general notation introduced in Chapter 2.
- *Language Syntax*, a section where we present the programs syntax supported by our approach. Such syntax is usually inspired by the case studies of the chapter.
- *Concrete Domain and Semantics*, a section which formally defines the concrete domain (i.e., the lattice of concrete objects we aim to abstract), and the concrete semantics (i.e., the effect of operations which manipulate the concrete objects).
- *Abstract Domain and Semantics*, a section where the abstract domain (i.e., the lattice of abstract elements which approximate the concrete ones) and the abstract counterparts of the concrete operations are presented. The characterization of an abstract domain requires the definition of: the abstract elements which compose the lattice, the partial order between elements, the top and bottom elements, the lub and glb operators, the widening operator (to guarantee convergence of the analysis, when the height of the domain is infinite), and finally the abstraction and concretization functions.

- *Experimental Results*, a section containing the empirical evaluation of the proposed abstract domain(s) to show if and how our solution solves the problem of the chapter.
- *Related Work*, a section which explores in detail the existing literature concerning the problem tackled, comparing it to our solution. The purpose of this section is to better understand if and how we reached the goal of the chapter.
- *Discussion*, a section which recaps the contribution of the chapter.

The results of Chapters 3, 4, 5 have been published in the proceedings of international conferences with program committee [45, 46, 48]. Moreover, an article based on the contents of Chapter 3 has been accepted for publication in the journal SPE (Software: Practice and Experience) [47]; an article based on the contents of Chapter 4 has been submitted to another journal and is currently being reviewed. A part of Chapter 2 (the survey on product operators in Abstract Interpretation) has also been published [42].

Chapter 2 introduces some of the necessary mathematical background. It is mostly intended to define the terminology and notation that will be used throughout the thesis.

Chapter 3 presents a framework of abstractions for string-valued variables. We start by defining the concrete semantics of strings on a selected subset of string operators. Then, we move on to the definition of five abstract domains for the static analysis of programs manipulating strings. The domains are the following: (i) *character inclusion*, which tracks the characters certainly or maybe included in a string; (ii) *prefix*, which tracks how a string begins; (iii) *suffix*, which (dually to prefix) tracks how a string ends; (iv) *bricks*, which represents a string through the composition of simplified regular expressions; (v) *string graphs*, which exploits the data structure of type graphs to represent strings through trees with backward arcs. Each domain has a different level of precision and complexity. For every domain, we prove the fundamental theoretical properties, which guarantee the soundness of the analysis, and we define the abstract semantics of the operators selected before. We show the main features of such framework by applying each domain to the analysis of some case studies, thus showing the differences in precision. Finally, we discuss the relationships between the domains of the framework, creating a lattice of abstract domains.

Chapter 4 defines the *Trapezoid Step Functions* (TSF) domain to abstract continuous functions, with the purpose of analyzing hybrid systems where the behaviour of the physical environment does not depend on the discrete control. After briefly recalling the definition of IVSF, the domain presented by Bouissou and Martel [24], we generalize their approach: instead of assigning

two numbers (for a lower and an upper-bound of sensor-values) to each time interval, the introduced abstract domain assigns two linear functions to each time interval. We present all the required lattice operations and for every one of them we give a proof of correctness. We put particular effort into the definition of a sound abstraction function, which has to keep into account the computability on a finite precision machine, where the floating point representation induces rounding errors. We apply the domain to the analysis of two case studies, and we show the increase in precision (without loss of performance) with respect to the approach of IVSF. We also give the semantics of arithmetic operations on functions, to allow for possible different applications (like cost analysis) of our domain.

Chapter 5 presents a novel disjunctive non-relational abstract domain (called *Parametric Hypercubes*) for the static analysis of physics simulations. The interest in such kind of applications is due to the massive use of physics in game software, a field where static analysis is still not effectively applied. To successfully analyze this kind of programs, the abstraction must have a way to consider relationships between variables, since every variable of a physics simulation strongly influences the life of other variables. We chose to implicitly express these relationships by tracking disjunctive information. More precisely, we consider the variables space (where each axis represents the values of a certain variable) and we divide it in hypercubes of fixed size. The state of the program is associated to a set of such hypercubes. When executing a statement, we produce a new set of hypercubes, by applying the operation separately on each single hypercube. We present the syntax supported by our abstract semantics, and we apply the analysis to a representative case study (a bouncing ball). We show how this domain allows for interactive debugging, letting the developer find the appropriate subset of starting values which make the program correct (with respect to a certain property). The first experimental results are satisfactory, both in terms of precision and performance. Note also that, even though we focused on floating-point variables, our approach can be instanced in applications other than game software, because it is parametric in the abstract domain used to abstract the single variables: we conclude this chapter by showing an example of a different application where our domain proves to be useful as well.

Chapter 6 reports the conclusions and the perspectives for future work.

2

Abstract Interpretation Background

In this chapter we introduce the mathematical background that will be used throughout the thesis. In particular, we introduce some basic notation, and some well-known theoretical results on lattices, fixpoints, and Abstract Interpretation theory. The main "classical" references to Abstract Interpretation are [50, 53, 55, 132]. An Abstract Interpretation-based program analysis can be tuned at different granularities, changing the trade-off between efficiency and precision. In some cases, it is also possible to refine the results through the combination of different domains: at the end of this chapter we investigate the various kinds of product operators between abstract domains [1].

Contents

[1] This part of the chapter is derived from [42].

2.1 Sets and Sequences

Let S, A, B be sets. $|S|$ denotes the cardinality of S and A/B denotes the set A where the elements of B have been removed. The set of all subsets of a set S will be denoted by $\wp(S)$. Set inclusion, union and intersection are respectively denoted $\subseteq, \cup, \cap$. The sets of all natural, integer, rational, and real numbers will be denoted, respectively, by $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$.

Given a possibly infinite set of symbols Σ, we denote by Σ^* the family of finite-length strings (i.e., ordered sequences) from symbols in Σ (that is, $\Sigma^* = \{s_1 \cdots s_n : \forall i \in [1..n] : s_i \in \Sigma\}$), including the empty string ϵ. Given a sequence s, we use the notation $s[i \cdots j]$ to indicate the subsequence starting at $s[i]$ and ending at $s[j]$ (extremes included).

Given two lists l_1, l_2 of any kind, let $concatList(l_1, l_2)$ be the function that returns their concatenation.

2.2 Interval Arithmetic

The concept of interval arithmetics was presented by R. E. Moore in [129]. A closed compact interval $[\underline{a}, \overline{a}]$ is defined as the closed set of real values of the form

$$[\underline{a}, \overline{a}] = \{c \in \mathbb{R} | \underline{a} \leq c \leq \overline{a}\}$$

where $\underline{a}$ is called the lower bound and $\overline{a}$ the upper bound.

The width of the interval is defined as the difference between its bounds $(\overline{a} - \underline{a})$. We say that $c \in R$ is in $[\underline{a}, \overline{a}]$, and denote it $c \in [\underline{a}, \overline{a}]$, if $\underline{a} \leq c \leq \overline{a}$. Moreover, we say that $[\underline{b}, \overline{b}]$ is included in $[\underline{a}, \overline{a}]$ (or that $[\underline{b}, \overline{b}]$ is a subinterval of $[\underline{a}, \overline{a}]$), and denote it $[\underline{b}, \overline{b}] \subseteq [\underline{a}, \overline{a}]$, if it holds that $\underline{a} \leq \underline{b}$ and $\overline{b} \leq \overline{a}$. Further, $[\underline{a}, \overline{a}]$ and $[\underline{b}, \overline{b}]$ are equal, denoted $[\underline{a}, \overline{a}] = [\underline{b}, \overline{b}]$, if $[\underline{a}, \overline{a}] \subseteq [\underline{b}, \overline{b}]$ and $[\underline{b}, \overline{b}] \subseteq [\underline{a}, \overline{a}]$.

Unary and binary arithmetic operators on intervals, for $n \in \mathbb{N}$, are defined as follows:

$$\begin{aligned}
[\underline{a}, \overline{a}] \mathbin{\widehat{+}} [\underline{b}, \overline{b}] &= [\underline{a} + \underline{b}, \overline{a} + \overline{b}] \\
[\underline{a}, \overline{a}] \mathbin{\widehat{-}} [\underline{b}, \overline{b}] &= [\underline{a} - \overline{b}, \overline{a} - \underline{b}] \\
[\underline{a}, \overline{a}] \mathbin{\widehat{\times}} [\underline{b}, \overline{b}] &= [\min((\overline{a} \times \overline{b}), (\underline{a} \times \overline{b}), (\overline{a} \times \underline{b}), (\underline{a} \times \underline{b})), \\
&\qquad \max((\overline{a} \times \overline{b}), (\underline{a} \times \overline{b}), (\overline{a} \times \underline{b}), (\underline{a} \times \underline{b}))] \\
[\underline{a}, \overline{a}] \mathbin{\widehat{/}} [\underline{b}, \overline{b}] &= \begin{cases} [\underline{a}, \overline{a}] \mathbin{\widehat{\times}} [1/\overline{b}, 1/\underline{b}] & \text{if } \underline{b} \times \overline{b} > 0 \\ \text{undefined} & \text{if } 0 \in [\underline{b}, \overline{b}] \end{cases}
\end{aligned}$$

2.3 Preorders, Partial and Total Orders

A **preorder** $\sqsubseteq_P$ over a set P is a binary relation which is reflexive (i.e., $\forall x \in P : x \sqsubseteq_P x$) and transitive (i.e., $\forall x, y, z \in P : x \sqsubseteq_P y \wedge y \sqsubseteq_P z \Rightarrow x \sqsubseteq_P z$). For $x \in P$, the downward closure of x, written $\downarrow x$, is defined by

$$\downarrow x := \{y \in P : y \sqsubseteq_P x\}$$

If $\sqsubseteq_P$ is also antisymmetric (i.e., $\forall x, y \in P : x \sqsubseteq_P y \wedge y \sqsubseteq_P x \Rightarrow x = y$), then it is called **partial order**. $\sqsubseteq_P$ is a **total order** if, in addition, for each $x, y \in P$, either $x \sqsubseteq_P y$ or $y \sqsubseteq_P x$. A set P equipped with a partial (resp., total) order $\sqsubseteq_P$ is said to be partially ordered (resp., totally ordered), and sometimes written $\langle P, \sqsubseteq_P \rangle$. Partially ordered sets are also called *posets*. A subset S of a poset $\langle P, \sqsubseteq_P \rangle$ is said to be a *chain* if it is totally ordered with respect to $\sqsubseteq_P$.

Given a poset $\langle P, \sqsubseteq_P \rangle$ and $S \subseteq P$, $y \in P$ is an upper bound for S if and only if $x \sqsubseteq_P y$ for each $x \in S$. An upper bound y for S is the **least upper bound** (or lub) of S if and only if for every other upper bound y' of S it holds $y \sqsubseteq_P y'$. The lub, when it exists, is unique. Lower bounds and greatest lower bounds (glb) are defined dually. $\langle P, \sqsubseteq_P \rangle$ is said to be *bounded* if it has a minimum and a maximum element (respectively, $\bot_P$ and $\top_P$). A *directed set* is a poset in which any two elements, and hence any finite subset, has an upper bound in the set. A *complete partial order* (abbreviated as cpo) is a poset such that every increasing chain has a least upper bound.

A poset $\langle P, \sqsubseteq_P \rangle$ is said to satisfy the ascending chain condition (**ACC**) [133] if every ascending chain $p_1 \sqsubseteq_P p_2 \sqsubseteq_P \ldots$ of elements of P is eventually stationary, that is, there is some positive integer n such that $\forall m > n : p_m = p_n$. Similarly, $\langle P, \sqsubseteq_P \rangle$ is said to satisfy the descending chain condition (DCC) if there is no infinite descending chain. Every finite poset satisfies both ACC and DCC.

2.4 Lattices

A poset $\langle L, \sqsubseteq_L \rangle$ such that, for each $x, y \in L$, both $lub\{x, y\}$ and $glb\{x, y\}$ exist, is called a **lattice**. In this case, lub and glb are also called, respectively, the *join* and the *meet* operations of the lattice. A poset where only the glb operation is well-defined is called a meet-semilattice. A **complete lattice** is a lattice $\langle L, \sqsubseteq_L \rangle$ such that every subset of L has both a least upper bound and a greatest lower bound. A complete lattice $\langle L, \sqsubseteq_L, \sqcup_L, \sqcap_L, \top_L, \bot_L \rangle$, with partial ordering $\sqsubseteq_L$, lub $\sqcup_L$, glb $\sqcap_L$, greatest element (top) $\top_L$, and least element (bottom) $\bot_L$, is denoted $\widehat{L}$. Any finite lattice is complete. A complete lattice is always a cpo: it has both a least element $\bot_L := \bigsqcup \emptyset$ and a greatest element $\top_L := \bigsqcup L$; also, each set $S \subseteq L$ has a greatest lower bound $\bigsqcap S := \bigsqcup \{X \in L \mid \forall Y \in S : X \sqsubseteq_L Y\}$. An important example of complete lattice is the power-set $\langle \wp(S), \subseteq, \cup, \cap, \emptyset, S \rangle$ of any set S.

2.5 Functions

A function is a relation r such that if $(x, y_1) \in r$ and $(x, y_2) \in r$, then $y_1 = y_2$. In other words, a function is a relation that relates each element of the domain to *at most* one element of the co-domain. Thus, given an element $x \in dom(r)$, we denote the element in the co-domain by $r(x)$. In order to define functions, we use the λ notation: by $f = \lambda x.E$, we denote a function f that relates the evaluation of the expression E (which depends on x) to the element x of its domain.

By the notation $f : X \mapsto Y$ we mean that the domain of the function f is included in X, and its co-domain is included in Y. Let $f : X \mapsto Y$ and $g : Y \mapsto Z$, then $g \circ f : X \mapsto Z$ represents the composition of functions f and g, i.e. $g(f(x))$.

Given two posets $\langle X; \sqsubseteq_X \rangle$ and $\langle Y; \sqsubseteq_Y \rangle$, a function $f : X \mapsto Y$ is:

- *monotonic* if it preserves the order of the elements, i.e. $\forall x_1, x_2 \in X : x_1 \sqsubseteq_X x_2 \Rightarrow f(x_1) \sqsubseteq_Y f(x_2)$
- *join preserving* if it preserves least upper bounds, i.e. $\forall x_1, x_2 \in X : f(x_1 \sqcup_X x_2) = f(x_1) \sqcup_Y f(x_2)$, where $\sqcup_X, \sqcup_Y$ are the least upper bound operators of the two posets
- *complete join preserving* if it preserves least upper bounds for arbitrary subsets of X, i.e. $\forall X_1 \subseteq X$ such that $\bigsqcup_X X_1$ exists, then $f(\bigsqcup_X X_1) = \bigsqcup_Y f(X_1)$
- *continuous* if it preserves the least upper bound of increasing chains, i.e. for all chains $C \subseteq X$ we have that $f(\bigsqcup_X C) = \bigsqcup_Y \{f(c) : c \in C\}$

Similarly, a function $f : X \mapsto Y$ is said to be *meet preserving* if it preserves the greatest lower bound of two elements and *complete meet preserving* if it preserves the greatest lower bound for any subset of X.

2.6 Fixpoints

Let f be a function on a poset $\langle X; \sqsubseteq_X \rangle$. The sets of pre-fixpoints, fixpoints, post-fixpoints of f are respectively:

- $prefp(f) = \{x \in X : x \sqsubseteq_X f(x)\}$
- $fp(f) = \{x \in X : x = f(x)\}$
- $postfp(f) = \{x \in X : f(x) \sqsubseteq_X x\}$

In particular, $\bot$ and $\top$ are, respectively, a pre-fixpoint and a post-fixpoint for all operators.

The *least fixpoint* of a function f is denoted $lfp(f)$ and is such as $lfp(f) \in fp(f) \wedge \forall p \in fp(f) : lfp(f) \sqsubseteq_X p$. If f has a least fixpoint, this is unique. Similarly,

the *greatest fixpoint* of a function f is denoted $gfp(f)$ and is such as $gfp(f) \in fp(f) \wedge \forall p \in fp(f) : p \sqsubseteq_X gfp(f)$. If f has a greatest fixpoint, this is unique. Moreover, we denote by $lfp_x^{\sqsubseteq} f$ the least fixpoint of f that is greater than x with respect to the order $\sqsubseteq$ and by $gfp_x^{\sqsubseteq} f$ the greatest fixpoint of f smaller than x with respect the order $\sqsubseteq$.

A monotonic function defined over a complete lattice admits a least and greatest fixpoint: this is guaranteed by Tarski's theorem presented in [143]. This fundamental theorem states that the set of fixpoints $fp(f)$ of a monotonic function f is a complete lattice:

Theorem 2.6.1 (Tarski's theorem). *Let $\langle L, \sqsubseteq_L, \sqcup_L, \sqcap_L, \top_L, \bot_L \rangle$ be a complete lattice. Let $f : L \mapsto L$ be a monotonic function on this lattice. Then, the set of fixpoints is a non-empty complete lattice, and:*

$$lfp_{\bot}^{\sqsubseteq_L} f = \bigsqcap_L \{x \in L : f(x) \sqsubseteq_L x\}$$
$$gfp_{\bot}^{\sqsubseteq_L} f = \bigsqcup_L \{x \in L : x \sqsubseteq_L f(x)\}$$

Since such theorem is not constructive (i.e., it does not explain how to compute the least and greatest fixpoint, it just states their existence), an alternative characterization of the least fixpoint for monotonic functions defined over a complete lattice can be given using the theorem presented in [54]. In this paper, Cousot & Cousot give a constructive proof of Tarski's theorem without using the continuity hypothesis.

Theorem 2.6.2 (Constructive version of Tarski's theorem). *Let $\langle L, \sqsubseteq_L, \sqcup_L, \sqcap_L, \top_L, \bot_L \rangle$ be a complete lattice. Let $f : L \mapsto L$ be a monotonic function on this lattice. Define the following sequence:*

$$f^0 = \bot_L$$
$$f^\delta = f(f^{\delta-1}) \text{ for every successor ordinal } \delta$$
$$f^\delta = \bigsqcup_{\alpha<\delta} f^\alpha \text{ for every limit ordinal } \delta$$

Then the ascending chain $\{f^i : 0 \leq i \leq \delta\}$ (where δ is an ordinal) is ultimately stationary for some $\rho \in \mathbb{N}$ that is $f^\rho = lfp_{\bot}^{\leq} f$.

The set of fixed points of f is shown to be the image of X by preclosure operations defined by means of limits of stationary transfinite iteration sequences. This characterizion of fixed points by iterative schemes leads to practical computation or approximation procedures and it allows the use of transfinite induction for proving properties of these fixed points.

2.7 Traces

Given a set S, a *trace* τ is a partial function $\mathbb{N} \mapsto S$ such that

$$\forall i \in \mathbb{N} : i \notin dom(\tau) \Rightarrow \forall j > i : j \notin dom(\tau)$$

This means that, if a trace is undefined for a certain $n \in \mathbb{N}$, then it is also undefined for all the successors of n: the domain of all non-empty traces is a segment of $\mathbb{N}$. The empty trace (i.e. the trace τ such that $dom(\tau) = \emptyset$) is denoted by ϵ_τ. Let S be a generic set of elements: we denote by $S^{\overrightarrow{+}}$ the set of all the finite traces composed by elements in S.

The length of a trace τ is formally defined as $len(\tau) = i+1 : i \in dom(\tau) \wedge i+1 \notin dom(\tau)$. If $\tau = \epsilon_\tau$, then $len(\tau) = 0$.

A trace is often represented as a sequence of states, i.e. $\sigma_0 \rightarrow \sigma_1 \rightarrow \dots$ corresponds to the trace $\{(0, \sigma_0), (1, \sigma_1), \dots\}$.

We represent with $S^{\overrightarrow{+}}_{\overset{T}{\rightarrow}}$ the set of traces in $S^{\overrightarrow{+}}$ which cannot be extended further with respect to the transition $\overset{T}{\rightarrow}$. Formally: $S^{\overrightarrow{+}}_{\overset{T}{\rightarrow}} = \{\sigma_0 \rightarrow \dots \rightarrow \sigma_i : \sigma_0 \rightarrow \dots \rightarrow \sigma_i \in S^{\overrightarrow{+}} \wedge \nexists \sigma_j \in S : \sigma_i \overset{T}{\rightarrow} \sigma_j\}$.

Two traces τ_1 and τ_2 can be concatenated and the result (written as $\tau_1 \rightarrow_\tau \tau_2$) represents the trace $\tau_1 \cup \{i \mapsto \sigma : \exists j \in dom(\tau_2) : i = j + len(\tau_1) \wedge \sigma = \tau_2(j)\}$.

Given a set of initial elements S_0 and a transition relation $\overset{T}{\rightarrow} \subseteq \Sigma \times \Sigma$, the partial trace semantics [55] builds up all the traces that can be obtained by starting from traces containing only a single element from S_0 and then iteratively applying the transition relation until a fixpoint is reached.

Definition 2.7.1 (Partial trace semantics [55])**.** *Let Σ be a set of states, $S_0 \subseteq \Sigma$ a set of initial elements, and $\overset{T}{\rightarrow} \subseteq \Sigma \times \Sigma$ a transition relation. Let $f : [\wp(\Sigma) \rightarrow [\Sigma^{\overrightarrow{+}} \rightarrow \Sigma^{\overrightarrow{+}}]]$ be the function defined as:*

$$F(S_0) = \lambda T.\{0 \mapsto \sigma_0 : \sigma_0 \in S_0\} \cup \\ \cup\{\sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \rightarrow \sigma_i : \sigma_0 \rightarrow \dots \rightarrow \sigma_{i-1} \in T \wedge \sigma_{i-1} \overset{T}{\rightarrow} \sigma_i\}$$

The partial trace semantics is defined as:

$$\mathbb{PT}[\![S_0]\!] = lfp^{\subseteq}_{\emptyset} F(S_0)$$

2.8 Galois Connections

Abstract Interpretation has been widely applied as a general technique for the sound approximation of the semantics of computer programs. In particular, abstract domains (to represent data) and semantics (to represent data operations) approximate

the concrete computation. When analyzing a program and trying to prove some properties on it, the quality of the result is determined by the abstract domain choice, since there is always a trade-off between accuracy and efficiency of the analysis.

As already explained in Chapter 1, the main idea of Abstract Interpretation is to define the semantics of a program as the fixpoint of a monotonic function. The concrete semantics belongs to a concrete semantic domain C which is a partially ordered set $\langle C; \sqsubseteq_C \rangle$. In such a setting, the partial order $\sqsubseteq_C$ formalizes the loss of information. The abstract semantics also belongs to a partial order $\langle A; \sqsubseteq_A \rangle$, which is ordered by the abstract version $\sqsubseteq_A$ of the concrete approximation order $\sqsubseteq_C$.

Definition 2.8.1 (Galois connection, [53]). *Let $\langle C; \sqsubseteq_C \rangle$ and $\langle A; \sqsubseteq_A \rangle$ be two posets. Two functions $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ form a Galois connection if and only if*

$$\forall c \in C : \exists a \in A : \alpha(c) \sqsubseteq_A a \Rightarrow c \sqsubseteq_C \gamma(a)$$

We denote this fact by writing $\langle C; \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A; \sqsubseteq_A \rangle$.

Theorem 2.8.1 ([53]). *Let $\langle C; \sqsubseteq_C \rangle$ and $\langle A; \sqsubseteq_A \rangle$ be two partial orders and let $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ be two maps such that:*

- *α and γ are monotone*
- *$\alpha \circ \gamma$ is reductive (i.e., $\forall a \in A : \alpha \circ \gamma(a) \sqsubseteq_A a$)*
- *$\gamma \circ \alpha$ is extensive (i.e., $\forall c \in C : c \sqsubseteq_C \gamma \circ \alpha(c)$)*

Then, it holds that $\langle C; \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A; \sqsubseteq_A \rangle$.

Usually, we call the left part of the Galois connection as the *concrete poset*, and the right one as the *abstract poset*. Similarly, γ is called the concretization function and α is the abstraction function. Galois connections enjoy several properties (composition, uniqueness of the adjoint, preservation of bounds, etc.) which are explained in [56]. A Galois connection can be induced by an abstraction function that is complete $\sqcup_A$ preserving, or dually by a concretization function that is complete $\sqcap_C$ preserving (where $\sqcup_A$ and $\sqcap_C$ are respectively the upper bound operator on the abstract lattice and the lower bound operator on the concrete lattice), as proved by Proposition 7 of [56].

Theorem 2.8.2 (Galois connection induced by lub preserving maps). *Let $\alpha : C \mapsto A$ be a complete join preserving map between posets $\langle C; \sqsubseteq_C \rangle$ and $\langle A; \sqsubseteq_A \rangle$. Define:*

$$\gamma = \lambda a. \sqcup_C \{c : \alpha(c) \sqsubseteq_A a\}$$

If γ is well-defined, then:

$$\langle C; \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A; \sqsubseteq_A \rangle$$

Theorem 2.8.3 (Galois connection induced by glb preserving maps)**.** *Let $\gamma : A \mapsto C$ be a complete meet preserving map between posets $\langle C; \sqsubseteq_C \rangle$ and $\langle A; \sqsubseteq_A \rangle$. Define:*

$$\alpha = \lambda c. \sqcap_A \{a : c \sqsubseteq_C \gamma(a)\}$$

If α is well-defined, then:

$$\langle C; \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A; \sqsubseteq_A \rangle$$

An interesting property of Galois connections is that they are compositional, i.e. the composition of two Galois connections is still a Galois connection.

Theorem 2.8.4 (Composition of Galois connections)**.** *Suppose that $\langle A; \sqsubseteq_A \rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle B; \sqsubseteq_B \rangle$ and $\langle B; \sqsubseteq_B \rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle C; \sqsubseteq_C \rangle$. Then:*

$$\langle A; \sqsubseteq_A \rangle \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} \langle C; \sqsubseteq_C \rangle$$

2.9 Soundness and Completeness

The process of abstraction induces loss of information with respect to the concrete domain and semantics. For this reason, the result of the application of the abstract semantics does not necessarily coincide with the result of the concrete one. However, as already hinted in Section 1.2.1, we require that properties which are verified by the abstract process, are verified also in the concrete one: the analysis must be *sound*. Instead, we accept that the abstract result includes more information (i.e., it is less precise) than the concrete one: the analysis can be *incomplete*.

We are now going to formally define the concepts of soundness and completeness of an abstract interpretation. We are also going to define a metrics to compare the precision of different abstract domains related to the same concrete domain.

Soundness

It is very important to prove the correctness of the abstract semantics with respect to the concrete one: the concretization of the results of the abstract semantics must *over-approximate* the output of the concrete semantics.

Definition 2.9.1 (Soundness)**.** *Let $\mathbb{S} : C \mapsto C$ and $\overline{\mathbb{S}} : A \mapsto A$ be the concrete and the abstract semantics respectively, where the two posets C, A form a Galois connection ($\langle C; \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A; \sqsubseteq_A \rangle$). The abstract semantics $\overline{\mathbb{S}}$ is sound iff for all the prefixpoints $\mathtt{p} \in P \subseteq A$ of $\overline{\mathbb{S}}$, we have that:*

$$\gamma \circ \overline{\mathbb{S}}[\![\mathtt{p}]\!] \sqsupseteq_C \mathbb{S}[\![\gamma(\mathtt{p})]\!]$$

The soundness of an abstract semantics can be proved in many ways [57], each relying a specific subset of properties of transfer functions, concrete and abstract lattices, concretization and abstraction functions. Note that we showed the definition of soundness based on the computational process of the concrete domain, but we could have expressed it also based on the computational process of the abstract domain.

Completeness

We would like to understand which conditions are required for an abstract interpretation to be complete, that is when the concrete and abstract processes of calculus preserve the same precision. When defining the soundness, we could base our definition both on the abstract process and the concrete one; in the case of completeness, however, the two characterizations are not equivalent. Following [117], we report here the most known notion of completeness, which compares the computational results in the abstract domain:

Definition 2.9.2 (Complete abstraction [55])**.** *Let C, A be two posets which form a Galois connection ($\langle C; \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A; \sqsubseteq_A \rangle$), and let $\mathbb{S} : C \mapsto C$ be a concrete function. Then, the abstract function $\overline{\mathbb{S}} : A \mapsto A$ is complete for $\mathbb{S}$, on the abstract domain $\alpha(C)$, if:*

$$\alpha \circ \mathbb{S} = \overline{\mathbb{S}} \circ \alpha$$

This notion (also called *backward* completeness, or $\mathcal{B}$-completeness) requires that the result of the abstract and concrete computations are the same in the abstract domain. If we decided to compare the results of the two computations in the concrete domain, we would obtain another notion of completeness (called *forward* completeness, or $\mathcal{F}$-completeness), which holds iff $\mathbb{S} \circ \gamma = \gamma \circ \overline{\mathbb{S}}$. This notion is less known with respect to $\mathcal{B}$-completeness, which is considered the standard notion of completeness in abstract interpretation [82, 131].

When abstract domains (and their corresponding abstract operations) do not respect the completeness property, it can be useful to measure their precision relatively to each other. In fact, note that many Galois connections can be defined with the same concrete domain as left part but different abstract domains as right part, where each abstract domain tracks a different kind of information on the concrete objects. Such abstract domains can be compared with regard to the precision of the abstract representation. In particular, we define that, given three partial orders $\langle A_1; \sqsubseteq_{A_1} \rangle, \langle A_2; \sqsubseteq_{A_2} \rangle, \langle C; \sqsubseteq_C \rangle$ and the maps $\alpha_{A_1} : C \mapsto A_1, \gamma_{A_1} : A_1 \mapsto C, \alpha_{A_2} : C \mapsto A_2, \gamma_{A_2} : A_2 \mapsto C$ satisfying the properties required by Theorem 2.8.1, the abstract domain A_1 is *more precise* than A_2 if and only if it holds that

$$\gamma_{A_1}(\alpha_{A_1}(x)) \subseteq \gamma_{A_2}(\alpha_{A_2}(x)) \; \forall x \in C$$

This definition compares the precision of the abstraction in itself, but it does not involve the abstract operations associated to the domains (the semantics). Again, we

could informally estimate the relative degree of precision of the abstract semantics of different domains, by executing the analysis on a reference set of benchmarks programs, and then comparing their results on such benchmarks.

2.10 Fixpoint Approximation

The concrete and abstract semantics of an Abstract Interpretation-based analysis are defined as the fixpoint computation of monotonic functions.

If the abstract domain enjoys the ACC condition, then the abstract semantics computation is guaranteed to terminate in a finite time. If not, the convergence of the analysis must be forced through the use of a widening operator. Informally, a widening is a particular kind of join for which every increasing sequence is stationary after a finite number of steps, by extrapolating an approximation of the sequence limit. Formally:

Definition 2.10.1 (Widening [43]). *Given an ascending chain $d_0 \leq d_1 \leq d_2 \leq \ldots$ in a poset $\langle C; \leq \rangle$, a widening operator $\nabla : C \mapsto C$ is an upper bound operator such that the chain $w_0 = d_0, w_1 = w_0 \nabla d_1, \ldots, w_i = w_{i-1} \nabla d_i$ is ultimately stationary, i.e. $\exists j \in \mathbb{N} : \forall k \in \mathbb{N} : k > j \Rightarrow w_j = w_k$.*

The analysis obtained using the widening operator is surely convergent and still sound, even if the results are more approximated.

Theorem 2.10.1 (Widening soundness). *Let $\langle C; \sqsubseteq_C \rangle$ and $\langle A; \sqsubseteq_A \rangle$ be two complete lattices, $\alpha : C \mapsto A$ and $\gamma : A \mapsto C$ be two functions such that $\langle C; \sqsubseteq_C \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A; \sqsubseteq_A \rangle$. Let $\mathbb{S} : C \mapsto C$ and $\overline{\mathbb{S}} : A \mapsto A$ be two monotonic function such that $lfp(\mathbb{S}) \sqsubseteq_C \gamma \circ lfp(\overline{\mathbb{S}})$.*

Then, the sequence defined by

$$a_0 = \bot$$
$$a_i = \begin{cases} \overline{\mathbb{S}}(a_{i-1}) & \text{if } \overline{\mathbb{S}}(a_{i-1}) \sqsubseteq_A a_{i-1} \\ \overline{\mathbb{S}}(a_{i-1}) \nabla a_{i-1} & \text{otherwise} \end{cases}$$

is ultimately stationary and its limit a_k is a post-fixpoint of $\mathbb{S}$. Hence, it soundly approximates the concrete semantics, i.e. $lfp(\mathbb{S}) \sqsubseteq_C \gamma \circ lfp(\overline{\mathbb{S}}) \sqsubseteq_C \gamma(a_k)$.

2.11 Product Operators

An interesting feature of the Abstract Interpretation theory is the possibility to combine different domains in the same analysis. In fact, the Abstract Interpretation framework offers some standard ways to compose abstract domains, ensuring the

preservation of the theoretical properties needed to guarantee the soundness of the analysis. These compositional methods are called *domain refinements*. A systematic treatment of abstract domain refinements has been given in [76, 79], where a generic refinement is defined to be a lower closure operator on the lattice of abstract interpretations of a given concrete domain. These kinds of operators on abstract domains provide high-level facilities to tune a program analysis in terms of accuracy and cost. Two of the most well-known domain refinements are the *disjunctive completion* [55, 58, 77, 80, 106] and the *reduced product* [55], but they are not the only ones. The reduced product can be seen as the most precise refinement of the simple Cartesian product. Moreover, the reduced cardinal power is introduced by [55]. While the other domain refinements have been, since their introduction, widely used and explored, the reduced cardinal power has seen less further developments since 1979, with the notable exceptions of [81] and, successively, its application inside the static program analyzer ASTRÉE.

In this section we aim at giving a survey of the different product operators introduced in the literature by providing a uniform terminology, an analysis of their complexity, and of the implementation effort they require. In particular, we introduce the three main ways of combining various abstract domains in the Abstract Interpretation theory (namely, the Cartesian product, the reduced product, and the reduced cardinal power). For the sake of simplicity, we will focus on the combination of *two* abstract domains. Therefore, we suppose that two abstract domains $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$ are given, and that they are equipped with lattice operators: $\langle \overline{\mathsf{A}}, \leq_{\overline{\mathsf{A}}}, \sqcup_{\overline{\mathsf{A}}}, \sqcap_{\overline{\mathsf{A}}} \rangle$ and $\langle \overline{\mathsf{B}}, \leq_{\overline{\mathsf{B}}}, \sqcup_{\overline{\mathsf{B}}}, \sqcap_{\overline{\mathsf{B}}} \rangle$.

In addition, let C be the concrete domain. We suppose that this domain is equipped with lattice operators as well: $\langle \mathsf{C}, \leq_{\mathsf{C}}, \sqcup_{\mathsf{C}}, \sqcap_{\mathsf{C}} \rangle$. We suppose that both $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$ are sound abstractions of C, that is, they form a Galois connection: $\langle \mathsf{C}, \leq_{\mathsf{C}} \rangle \underset{\alpha_{\overline{\mathsf{A}}}}{\overset{\gamma_{\overline{\mathsf{A}}}}{\leftrightarrows}} \langle \overline{\mathsf{A}}, \leq_{\overline{\mathsf{A}}} \rangle$ and $\langle \mathsf{C}, \leq_{\mathsf{C}} \rangle \underset{\alpha_{\overline{\mathsf{B}}}}{\overset{\gamma_{\overline{\mathsf{B}}}}{\leftrightarrows}} \langle \overline{\mathsf{B}}, \leq_{\overline{\mathsf{B}}} \rangle$, where $\alpha_{\overline{\mathsf{A}}}, \gamma_{\overline{\mathsf{A}}}$ and $\alpha_{\overline{\mathsf{B}}}, \gamma_{\overline{\mathsf{B}}}$ are the abstraction and concretization functions of $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$, respectively. [2]

Finally, abstract domains provide abstract semantic transformers. Formally, we suppose that $\overline{\mathsf{A}}$ provides $\mathbb{S}_{\overline{\mathsf{A}}} : \overline{\mathsf{A}} \to \overline{\mathsf{A}}$, and $\overline{\mathsf{B}}$ provides $\mathbb{S}_{\overline{\mathsf{B}}} : \overline{\mathsf{B}} \to \overline{\mathsf{B}}$. These are sound approximation of the concrete semantics $\mathbb{S}_{\mathsf{C}} : \mathsf{C} \to \mathsf{C}$. Formally, $\forall \overline{\mathsf{a}} \in \overline{\mathsf{A}} : \mathbb{S}_{\mathsf{C}}[\![\gamma_{\overline{\mathsf{A}}}(\overline{\mathsf{a}})]\!] \leq_{\mathsf{C}} \gamma_{\overline{\mathsf{A}}}(\mathbb{S}_{\overline{\mathsf{A}}}[\![\overline{\mathsf{a}}]\!])$ and $\forall \overline{\mathsf{b}} \in \overline{\mathsf{B}} : \mathbb{S}_{\mathsf{C}}[\![\gamma_{\overline{\mathsf{B}}}(\overline{\mathsf{b}})]\!] \leq_{\mathsf{C}} \gamma_{\overline{\mathsf{B}}}(\mathbb{S}_{\overline{\mathsf{B}}}[\![\overline{\mathsf{b}}]\!])$.

2.11.1 Cartesian Product

The elements of this domain are elements in the Cartesian product of the two domains, and the operators are defined as the component-wise application of the operators of the two domains.

[2] There exist other approaches which can be used as well (e.g., when the best abstraction function does not exist [62]). However, the Galois connection-based approach is definitely the most commonly used [56].

Formally, let $\mathfrak{C} = \overline{\mathsf{A}} \times \overline{\mathsf{B}}$ be the Cartesian product. The partial order is defined as the conjunction of the partial orders of the two domains ($(\overline{\mathsf{a}}_1, \overline{\mathsf{b}}_1) \leq_{\mathfrak{C}} (\overline{\mathsf{a}}_2, \overline{\mathsf{b}}_2) \Leftrightarrow \overline{\mathsf{a}}_1 \leq_{\overline{\mathsf{A}}} \overline{\mathsf{a}}_2 \wedge \overline{\mathsf{b}}_1 \leq_{\overline{\mathsf{B}}} \overline{\mathsf{b}}_2$). Similarly, the least upper bound and the greatest lower bound operators are defined as the component-wise application of the operators of the two domains ($(\overline{\mathsf{a}}_1, \overline{\mathsf{b}}_1) \sqcup_{\mathfrak{C}} (\overline{\mathsf{a}}_2, \overline{\mathsf{b}}_2) = (\overline{\mathsf{a}}_1 \sqcup_{\overline{\mathsf{A}}} \overline{\mathsf{a}}_2, \overline{\mathsf{b}}_1 \sqcup_{\overline{\mathsf{B}}} \overline{\mathsf{b}}_2)$ and $(\overline{\mathsf{a}}_1, \overline{\mathsf{b}}_1) \sqcap_{\mathfrak{C}} (\overline{\mathsf{a}}_2, \overline{\mathsf{b}}_2) = (\overline{\mathsf{a}}_1 \sqcap_{\overline{\mathsf{A}}} \overline{\mathsf{a}}_2, \overline{\mathsf{b}}_1 \sqcap_{\overline{\mathsf{B}}} \overline{\mathsf{b}}_2)$, respectively). This way, we obtain that the Cartesian product $\langle \mathfrak{C}, \leq_{\mathfrak{C}}, \sqcup_{\mathfrak{C}}, \sqcap_{\mathfrak{C}} \rangle$ forms a lattice. The pairwise approach to combine operators holds also in the case of widening. In fact, given the widening operators ∇_A and ∇_B on the domains A and B, respectively, the operator $\nabla_{A \times B}((a_1, b_1), (a_2, b_2)) = (a_1 \nabla_A a_2, b_1 \nabla_B b_2)$ is a widening operator on $\mathfrak{C}$ [43].

In addition, the abstraction function $\alpha_{\mathfrak{C}}$ consists in the component-wise application of the abstraction functions of the two domains ($\alpha_{\mathfrak{C}}(\mathsf{c}) = (\alpha_{\overline{\mathsf{A}}}(\mathsf{c}), \alpha_{\overline{\mathsf{B}}}(\mathsf{c}))$), while the concretization function $\gamma_{\mathfrak{C}}$ consists in the intersection of the results obtained by the concretization functions of the two domains on the corresponding component ($\gamma_{\mathfrak{C}}(\overline{\mathsf{a}}, \overline{\mathsf{b}}) = \gamma_{\overline{\mathsf{A}}}(\overline{\mathsf{a}}) \sqcap_{\mathsf{C}} \gamma_{\overline{\mathsf{B}}}(\overline{\mathsf{b}})$). Then, the Cartesian product forms a Galois connection with the concrete domain (formally, $\langle \mathsf{C}, \leq_{\mathsf{C}} \rangle \underset{\alpha_{\mathfrak{C}}}{\overset{\gamma_{\mathfrak{C}}}{\leftrightarrows}} \langle \mathfrak{C}, \leq_{\mathfrak{C}} \rangle$).

Finally, also the semantic operator $\mathbb{S}_{\mathfrak{C}} : \mathfrak{C} \to \mathfrak{C}$ is defined as the component-wise application of the abstract semantics of the two domains (formally, $\mathbb{S}_{\mathfrak{C}}[\![(\overline{\mathsf{a}}, \overline{\mathsf{b}})]\!] = (\mathbb{S}_{\overline{\mathsf{A}}}[\![\overline{\mathsf{a}}]\!], \mathbb{S}_{\overline{\mathsf{B}}}[\![\overline{\mathsf{b}}]\!])$). This way, the semantics of the Cartesian product is a sound over-approximation of the concrete semantics ($\forall (\overline{\mathsf{a}}, \overline{\mathsf{b}}) \in \mathfrak{C} : \mathbb{S}_{\mathsf{C}}[\![\gamma_{\mathfrak{C}}(\overline{\mathsf{a}}, \overline{\mathsf{b}})]\!] \leq_{\mathsf{C}} \gamma_{\mathfrak{C}}(\mathbb{S}_{\mathfrak{C}}[\![\overline{\mathsf{a}}, \overline{\mathsf{b}}]\!])$).

As pointed out by Patrick Cousot [49], "the Cartesian product discovers in one shot the information found separately by the component analyses", but "we do not learn more by performing all analyses simultaneously than by performing them one after another and finally taking their conjunctions".

In addition, the Cartesian product may contain several abstract elements that represent the same information. For instance, consider the Cartesian product of the Interval and the Parity domains, and in particular the elements $([2..4], \mathsf{O})$, $([2..3], \mathsf{O})$, $([3..4], \mathsf{O})$, and $([3..3], \mathsf{O})$, where O represents the odd element of the Parity domain. All these elements concretize to the singleton $\{3\}$, but some of them are not minimal[3].

Complexity

When applying lattice or semantic operators, the complexity of the operator defined on $\mathfrak{C}$ is exactly the sum of the complexity of the corresponding operators on $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$. Instead, the height of the lattice of $\mathfrak{C}$ (that is important to estimate the complexity of computing a fixpoint using this domain) is the multiplication of the heights of $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$.

[3] An abstract element $\overline{\mathsf{a}}$ is minimal w.r.t. a property $\mathsf{c} \in \mathsf{C}$ if and only if (i) $\gamma(\overline{\mathsf{a}}) \geq_{\mathsf{C}} \mathsf{c}$ and (ii) $\nexists \overline{\mathsf{a}}' : \gamma(\overline{\mathsf{a}}') \geq_{\mathsf{C}} \mathsf{c} \wedge \overline{\mathsf{a}}' < \overline{\mathsf{a}}$.

Implementation

Given the implementations of $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$, the implementation of $\mathfrak{C}$ is completely straightforward, and it could be used to combine any existing abstract domain in a completely generic way. In fact, the implementation only requires the existence of the operators, and there is no need to develop anything specific on such domains.

2.11.2 Reduced Product

Even if the Cartesian product is a quite effective way to cheaply combine two domains in terms of both formalization and implementation, it is clear that one may want to let the information flow among the two domains to mutually refine them. Already in one of the foundative papers of Abstract Interpretation [55], Patrick and Radhia Cousot introduced the reduced product exactly with the purpose of refining the information tracked by $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$. In particular, when we have an abstract state that is non-minimal, we can take the smallest element which represents the same information by *reducing* it. A reduction improves the precision of the abstract representation with respect to the order in the Cartesian product without affecting its concrete meaning. Intuitively, a reduction exploits the information tracked by one of the two domains involved in the product to refine the information tracked by the other one (and vice versa). Let (a, b) be an element of a reduced product (where a and b belong respectively to the two domains combined in the product: $a \in \overline{\mathsf{A}}$, $b \in \overline{\mathsf{B}}$). Let c_1 be the set of concrete values associated to a and c_2 be the set of concrete values associated to b. Then, the element (a, b) represents the set of concrete elements $c_1 \cap c_2$. The reduction tries to find the smallest element (a', b') such that the concretizations of a' and b' are subsets of those of a, b (respectively), but their intersection remains the same as the original one ($c_1 \cap c_2$).

The lattice and semantic structures of the reduced product are exactly the same as those of the Cartesian product. In addition, a reduction operator aimed at refining the information tracked by the two domains is introduced, and it is used after each lattice or semantic operator application. Formally, the reduction operator $\rho : \mathfrak{C} \to \mathfrak{C}$ is defined by $\rho(\overline{\mathsf{c}}) = \bigsqcap_{\mathfrak{C}} \{\overline{\mathsf{c}}' \in \mathfrak{C} : \gamma_{\mathfrak{C}}(\overline{\mathsf{c}}) \leq_{\mathfrak{C}} \gamma_{\mathfrak{C}}(\overline{\mathsf{c}}')\}$. Nevertheless, such definition is not computable in general, and often one wants to have a relaxed version of this operator that is not expensive to compute. In general, a reduction operator has to satisfy the following two properties: (i) $\rho(\overline{\mathsf{c}}) \leq_{\mathfrak{C}} \overline{\mathsf{c}}$ (the result of its application is a more precise abstract element); (ii) $\gamma(\rho(\overline{\mathsf{c}})) = \gamma(\overline{\mathsf{c}})$ (an abstract element and its reduction represent the same property).

Consider again the example of the product of the Interval and Parity domains. A simple reduction operator may increase by one the lower bound (or decrease by one the upper bound) of the interval if the bound does not respect the information tracked by the Parity domain (e.g., it is odd while the parity tracks that the value is even). This way, the reduction of $([2..4], \mathsf{O})$, $([2..3], \mathsf{O})$, and $([3..4], \mathsf{O})$ yields in all cases the abstract value $([3..3], \mathsf{O})$. Note that the reduction operator does not

always obtain the minimal information. For instance, if we reduce $([1..1], \mathsf{E})$ (where E represents the even element of the Parity domain), we would obtain $(\bot_I, \mathsf{E})$ (where $\bot_I$ is the bottom element of the Intervals domain), that could be further reduced to $(\bot_I, \bot_P)$ (where $\bot_P$ is the bottom element of the Parity domain). Therefore, the reduction operator usually requires to compute a fixpoint [49]. As two other examples, consider the reduced product of Intervals and Congruences [4] domains. Firstly, the reduction of the abstract value $([2..2], 3)$ produces the abstract value $(\bot, 3)$ which is not a minimal element. We need to iterate the reduction to obtain $(\bot, \bot)$. Secondly, the reduction of $([4..5], 2)$ produces the abstract value $([4..4], 2)$ which can be reduced again to $([4..4], 4)$.

Observe that the widening operator on the reduced product cannot be derived "for free" as refinement on the the widening operators of the components. As proved in [43], this is true only under the (quite strict) condition that $\forall a_1, a_2 \in A, \forall b_1, b_2 \in B, (a_1 \nabla_A a_2, b_1 \nabla_B b_2) \in \rho(A \times B)$, where $\rho(A \times B)$ represents the elements of the reduced product. This property does not often hold in practice. A far simpler (and naive) solution consists in applying the widening component-wise and refraining from reducing the result before feeding it back as left argument of the next iteration' s widening. This subsumes the above condition (as the reduction becomes idempotent on the iterates), but it also allows converging when the condition does not hold.

Complexity

In addition to the complexity of the Cartesian product, the reduced product requires to compute the reduction operator. Therefore, the complexity of an operator of the reduced product is the sum of the complexity of the operators defined on $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$ and of the reduction operator. Since this operator may require computing a fixpoint, the final cost of a generic operator could be rather expensive. Therefore, usually it is more convenient to define a reduction operator that refines only partially the information tracked by the two domains [114].

Implementation

The implementation of the reduction operator has to be specific for the domains we are refining. Therefore, while the Cartesian product was completely generic and automatic, the reduced product requires one to define and implement how two domains let the information flow among them. This means that each time we want to combine two domains in a reduced product we have to implement such operator. On the other hand, all the other lattice and semantic operators are defined exactly

[4] An abstract element of the Congruence domain is defined as two integer numbers (a, b) which represent the set of concrete values $\{x : x \in a\mathbb{Z} + b\}$. Here we consider a simpler version, where an abstract element is made by only one integer a, which abstracts all multiples of such value: $\{x : x \in a\mathbb{Z}\}$.

as in the Cartesian product, except that they have to call the reduction operator at the end, but this can be implemented generically w.r.t. the combined domains.

Granger Product

Granger [88] proposed an elegant solution to compute an approximation of the reduction operator. Granger based his new product on the definition of two operators $\rho_1 : \mathfrak{C} \to \overline{\mathsf{A}}$ and $\rho_2 : \mathfrak{C} \to \overline{\mathsf{B}}$. The idea is that each operator refines one of the two domains involved in the product. The final reduction is obtained by iteratively applying ρ_1 and ρ_2. In order to have a sound reduction operator, ρ_1 and ρ_2 have to satisfy the following conditions:

- $\rho_1(\overline{\mathsf{a}}, \overline{\mathsf{b}}) \leq_{\overline{\mathsf{A}}} \overline{\mathsf{a}} \wedge \gamma_{\mathfrak{C}}(\rho_1(\overline{\mathsf{a}}, \overline{\mathsf{b}}), \overline{\mathsf{b}}) = \gamma_{\mathfrak{C}}(\overline{\mathsf{a}}, \overline{\mathsf{b}})$
- $\rho_2(\overline{\mathsf{a}}, \overline{\mathsf{b}}) \leq_{\overline{\mathsf{B}}} \overline{\mathsf{b}} \wedge \gamma_{\mathfrak{C}}(\overline{\mathsf{a}}, \rho_2(\overline{\mathsf{a}}, \overline{\mathsf{b}})) = \gamma_{\mathfrak{C}}(\overline{\mathsf{a}}, \overline{\mathsf{b}})$

The intuition behind Granger's product is that dealing with only one flow of information at a time is simpler. Each of the two operators ρ_1, ρ_2 tries to descend in one of the lattices: given the abstract element made by the pair $(\overline{\mathsf{a}}, \overline{\mathsf{b}})$, the ρ_1 operator uses the information from $\overline{\mathsf{b}}$ to go down the lattice of $\overline{\mathsf{A}}$, while the ρ_2 operator uses the information from $\overline{\mathsf{a}}$ to go down the lattice of $\overline{\mathsf{B}}$. After each application of ρ_1 or ρ_2 we get a smaller element. The descent is iteratively repeated until the operators cannot recover any more precision: the reduction operator $\rho(\overline{\mathsf{a}}, \overline{\mathsf{b}})$ is then defined as the fixpoint of the decreasing iteration sequence obtained by applying ρ_1 and ρ_2. This is defined by the sequence $(\overline{\mathsf{a}}^n, \overline{\mathsf{b}}^n)_{n \in \mathbb{N}}$ as follows:

$$(\overline{\mathsf{a}}^0, \overline{\mathsf{b}}^0) = (\overline{\mathsf{a}}, \overline{\mathsf{b}})$$
$$(\overline{\mathsf{a}}^{n+1}, \overline{\mathsf{b}}^{n+1}) = (\rho_1(\overline{\mathsf{a}}^n, \overline{\mathsf{b}}^n), \rho_2(\overline{\mathsf{a}}^n, \overline{\mathsf{b}}^n))$$

The Granger product has exactly the same complexity we discussed for the reduced product. The main practical advantage of the Granger product is that one only needs to define and implement ρ_1 and ρ_2, that is, how the information flows from one domain to the other in one step. Then the reduction operator relying on the fixpoint computation comes for free.

Open Product

Cortesi et al. [41] proposed a further refinement of the Cartesian product. Its purpose is to let the domains interact with each other during *and* after operations by making explicit the domains' interaction through (abstract) queries. The open product is orthogonal to Granger's product and the two proposals can be combined, by incorporating Granger's idea of refinement inside the open product. The open product is orthogonal also to other methods such as down-set completion, and tensor product.

2.11.3 Reduced Cardinal Power

The reduced cardinal power was introduced by Cousot and Cousot in [55], but the literature concerning it has been relatively poor on both the theoretical and the practical level. The main feature of the cardinal power is that it allows one to track disjunctive information over the abstract values of the analysis. For instance, given the Interval and the Parity domain, one could track information like "when `x` is odd, `y` is in [0..10]". Some examples of the application of the cardinal power are the example 10.2.0.2 of [55], and examples 3 and 4 of [61]. In addition, a detailed explanation with various examples has been proposed by Giacobazzi and Ranzato [81]. Let us look at the example in [61], where the following slice of code is analyzed (typical of data transfer protocols where even and odd numbered packets contain data of different types):

```
1 n := 10; i := 0; A := new int[n];
2 while (i < n) do {
3     A[i] := 0;
4     i := i + 1;
5     A[i] := -16;
6     i := i + 1;
7 }
```

To analyze it, the authors combine *Parity* (where the lattice is made by the abstract elements $\bot, o, e, \top$) and *Intervals*. The reduced cardinal power of Intervals by Parity tracks abstract properties of the form $(o \to i_o, e \to i_e)$, which means that the interval associated to some variable is i_o (resp., i_e) when the parity associated to another variable (which could be the same) is o (resp., e). First of all, the authors show a non-relational analysis of the listing above, where they use:

- the reduced product of Parity and Intervals for simple variables;
- the reduced cardinal power of Parity by Interval for array elements (hence ignoring their relationship to indexes)

For example $(o \to \bot, e \to [-16, 0])$ means that the indexed array elements must be even with value included between -16 and 0. The result of this analysis is: $i : (e, [10, 10])$ (variable `i` is even and has value 10), $n : (e, [10, 10])$ (variable n is even and has value 10), and $A : (o \to \bot, e \to [-16, 0])$, which represents that array elements are abstracted by $(o \to \bot, e \to [-16, 0])$ (i.e., they are even and with values in $[-16, 0]$). The precision of this analysis can be greatly improved by using again the reduced cardinal power of Intervals by Parity, but this time relating the parity of an *index* of the array to the interval of the *elements* of the array at that index. The new result is $A : (o \to [-16, -16], e \to [0, 0])$, which means that the array elements at odd indexes are equal to -16 while those at even indexes are 0.

The reduced cardinal power has been formalized as follows in [55]. Given two abstract domains $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$, the cardinal power $\mathfrak{P} = \overline{\mathsf{B}}^{\overline{\mathsf{A}}}$ with base $\overline{\mathsf{B}}$ and exponent $\overline{\mathsf{A}}$ is the set of all isotone maps $\mathfrak{P} : \overline{\mathsf{A}} \to \overline{\mathsf{B}}$. Roughly, the combination of two abstract domains in $\overline{\mathsf{B}}^{\overline{\mathsf{A}}}$ means that a state in $\overline{\mathsf{A}}$ implies the abstract state of $\overline{\mathsf{B}}$ it is in relation with. The partial ordering $\leq_{\mathfrak{P}}$ is defined by $\overline{\mathsf{f}} \leq_{\mathfrak{P}} \overline{\mathsf{g}} \Leftrightarrow \forall \overline{\mathsf{x}} \in \overline{\mathsf{A}} : \overline{\mathsf{f}}(\overline{\mathsf{x}}) \leq_{\overline{\mathsf{B}}} \overline{\mathsf{g}}(\overline{\mathsf{x}})$. Similarly, the least upper bound and greatest lower bound operators are defined as the pointwise application of the operators of $\overline{\mathsf{B}}$. This way, $\langle \mathfrak{P}, \leq_{\mathfrak{P}}, \sqcup_{\mathfrak{P}}, \sqcap_{\mathfrak{P}} \rangle$ forms a lattice.

Let f_1, f_2 be two abstract elements in $\mathfrak{P}$. Then, the widening operator $\nabla_{\mathfrak{P}}$ on (f_1, f_2) can be defined as:

$$\forall x \in \overline{\mathsf{A}} : \nabla_{\mathfrak{P}}(f_1, f_2)(x) = \nabla_{\overline{\mathsf{B}}}(f_1(x), f_2(x))$$

Observe that the operator above can be effectively applied only if $\overline{\mathsf{A}}$ is finite.

By defining $\alpha_{\mathfrak{P}}(\mathsf{c}) = \lambda \overline{\mathsf{x}}.\alpha_{\overline{\mathsf{B}}}(\mathsf{c} \sqcap_{\mathsf{C}} \gamma_{\overline{\mathsf{A}}}(\overline{\mathsf{x}}))$ and $\gamma_{\mathfrak{P}}(\overline{\mathsf{p}})$ consequently, we have that $\langle \mathsf{C}, \leq_{\mathsf{C}} \rangle \underset{\alpha_{\mathfrak{P}}}{\overset{\gamma_{\mathfrak{P}}}{\leftrightarrows}} \langle \mathfrak{P}, \leq_{\mathfrak{P}} \rangle$. We refer the interested reader to [81] (and in particular to Theorem 3.6 and Proposition 3.7, where $\odot$ corresponds to $\sqcap_{\mathsf{C}}$) for more details and formal proofs.

A correctness result was presented in [55] as well (Theorem 10.2.0.1). In this work, the authors focused on a collecting semantics defined by a lattice of assertions which is a Boolean algebra. Afterwards, Cousot and Cousot did not broaden their theoretical definition to a more general setting.

Let us recall the example used in [55] to show the expressiveness of this domain.

```
1 x := 100; b := true;
2 while b do {
3     x := x - 1;
4     b := (x > 0);
5 }
```

The exponent of the cardinal power we use to analyze this example is the Boolean domain for variable `b`, while the base is the Sign domain for variable `x` tracking values $+, -, 0$ as well as $0+$ (meaning that the values are ≥ 0), $0-$ (meaning that the values are ≤ 0), $\neq 0$ (meaning that the values are different from 0). This way, we track that when variable `b` has a particular Boolean value, then the sign of variable `x` has a particular sign. Immediately before entering the `while` loop, we know that `b` = `true` $\Rightarrow$ `x` $= +$, while `b` = `false` $\Rightarrow$ `x` $= \bot$. After the application of the semantics of statement 3, we will have that `b` = `true` $\Rightarrow$ `x` $= 0+$, and `b` = `false` $\Rightarrow$ `x` $= \bot$, because the value of `b` is unchanged (it is certainly true) while the value of `x` has been decreased by one (so it could become equal to zero or remain greater than zero). After line 4, we obtain that `b` = `true` $\Rightarrow$ `x` $= +$, and `b` = `false` $\Rightarrow$ `x` $= 0$, since the new condition `x` > 0 is assigned to `b`. In fact,

`b` equals to `true` implies that `x` must be greater than zero. In addition, we knew that the value of `x` was ≥ 0. Then, if `b` is now `false`, we are sure that `x` will be equal to zero (but not less than zero). The fixpoint computation over the while loop stabilizes immediately (because, if we enter the loop again, we know that `b` is true and, as a consequence, `x` is positive, thus returning to the same conditions of the first iteration), and so we obtain that at the end of the program we have that $\texttt{b} = \texttt{true} \Rightarrow \texttt{x} = \bot$, and $\texttt{b} = \texttt{false} \Rightarrow \texttt{x} = 0$, since we have to assume the negation of `b` to terminate the execution of the `while` loop.

The cardinal power effectiveness is compromised when $\overline{\mathsf{A}}$ is infinite (i.e., intervals in $\mathbb{Z}$), and can become costly when $\overline{\mathsf{A}}$ is finite but non-trivial (i.e., intervals of machine integers). For this reason, some restricted forms of cardinal power can be used, where only a finite subset of $\overline{\mathsf{A}}$ is represented.

Summarizing, the main difficulties in constructing a reduced cardinal power domain are: (i) the choice of elements in $\overline{\mathsf{A}}$ to use; and (ii) the efficient design of abstract operators.

Complexity

Each time a lattice or semantic operator has to be applied to the abstract state, the cardinal power requires to apply it to all the elements of the base. Consider the cardinal power $\overline{\mathsf{B}}^{\overline{\mathsf{A}}}$. In a state of our domain, we will track a state of $\overline{\mathsf{B}}$ for each possible state of $\overline{\mathsf{A}}$. Let n be the number of states of $\overline{\mathsf{A}}$, a the cost of an operator on $\overline{\mathsf{A}}$ and b the cost on $\overline{\mathsf{B}}$. Then the overall cost over $\overline{\mathsf{B}}^{\overline{\mathsf{A}}}$ is $n * (a + b)$, since, for any element in $\overline{\mathsf{A}}$ we have to apply the operator both on $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$.

Let h_a and h_b be the height of the lattice of $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$, respectively. Then the height of $\overline{\mathsf{B}}^{\overline{\mathsf{A}}}$ is $h_b^{h_a}$.

It is then clear that the cardinal power causes a significant increase in the complexity of the analysis w.r.t. the complexity of the two original analyses. If there is already practical evidence that the reduction operator in the reduced product may induce an analysis that is too complex [114], it is even more important to carefully choose the two domains combined in a cardinal power. Nevertheless, particular instances of the cardinal power are already used to analyze industrial software, and in particular in ASTRÉE [18]. ASTRÉE exploits the Boolean relation domain, which applies the cardinal power using the values of some particular Boolean program variables as exponent. In this way, the analysis tracks precise disjunctive information w.r.t. these variables. In addition, ASTRÉE contains trace partitioning [118]. This can be seen as a cardinal power in which the exponent is a set of manually provided tokens on which the analysis tracks disjunctive information. There are various types of tokens the user can provide: particular abstract values of a variable, the begin of an `if` statement, etc. It is proved that, in practice, if an expert user provides the *right* tokens, the resulting analysis can be quite precise preserving its performances at the same time.

Implementation

The implementation can be rather simple when using a programming language providing functional constructs. In fact, the most part of the cardinal power (namely, elements of the domain, and lattice and semantic operators) can be defined as the functional point-wise application of the operators on the base and the exponent. Instead, the implementation of the cardinal power may be more verbose using an imperative programming language, but we do not expect it represents a significant challenge.

Reduced Relative Power

Giacobazzi and Ranzato generalized the reduced cardinal power [81]. For this purpose, the authors introduce the operation of *reduced relative power* on abstract domains. As it happened with the cardinal power, the reduced relative power is based on two domains $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$ (respectively, the exponent and the base) and is defined in a general and standard Abstract Interpretation setting. Its formal definition is $\overline{\mathsf{A}} \xrightarrow{\odot} \overline{\mathsf{B}}$, where $\odot$ is a generic operator used to combine concrete denotations. It is called "reduced *relative* power" because it is parametric with respect to $\odot$. The operator $\odot$ should be thought of as a kind of combinator of concrete denotations: the glb is a typical example, but another less restrictive combinator could be needed for some non-trivial applications. An example comes from the field of logic program semantics. The reduced relative power can be used to systematically derive new declarative semantics for logic programs by composing the domains of interpretation of some well-known semantics. In this case a concrete domain of sets of program execution traces is endowed with an operator of trace-unfolding that does not behave like a meet-operation (in particular, it is not even commutative). For more details, see Section 7 of [81]. The definition of the reduced relative power is as follows. $\overline{\mathsf{A}} \xrightarrow{\odot} \overline{\mathsf{B}}$ consists of all the monotone functions from $\overline{\mathsf{A}}$ to $\overline{\mathsf{B}}$ having the shape $\lambda\overline{\mathsf{x}}.\alpha_B(\mathsf{d} \odot \gamma_A(\overline{\mathsf{x}}))$, where: (i) d ranges over concrete values, (ii) γ_A is the concretization function of $\overline{\mathsf{A}}$ and (iii) α_B is the abstraction function of $\overline{\mathsf{B}}$. These monotone functions establish a dependency between the values of $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$, and for this reason are called *dependencies*. Intuitively, a dependency encodes how the abstract domain $\overline{\mathsf{B}}$ is able to represent the "reaction" of the concrete value d whenever it is combined via $\odot$ with an object described by $\overline{\mathsf{A}}$.

2.11.4 Examples

In this section, we discuss the application of the Cartesian product, the reduced product, and the cardinal power to some examples dealing with arrays. This way, we show the main features and limits of each combination of domains. The two abstract domains we will combine are Intervals [53] and a relational domain that tracks constraints of the form $\mathtt{x} < \mathtt{y} + c$.

Cartesian Product

As a first example, we consider a quite standard program that initializes to 0 all the elements of a given array.

```
1 for(i = 0; i < arr.length; i++)
2     arr[i] = 0;
```

We want to prove that the array accesses are safe, that is, $\texttt{i} \geq 0$ and $\texttt{i} < \texttt{arr.length}$, in particular when we perform $\texttt{arr}[\texttt{i}] = \texttt{0}$.

If we run the analysis using only Intervals, we obtain that $\texttt{i} = [0.. + \infty]$. In fact, this domain cannot infer any information from the loop guard $\texttt{i} < \texttt{arr.length}$ since it does not have any upper bound for arr.length. This result suffices to prove the first part of our property ($\texttt{i} \geq 0$) but not the second part ($\texttt{i} < \texttt{arr.length}$). If we run the analysis using only a relational domain, we obtain that $\texttt{i} < \texttt{arr.length} + 0$ when we analyze the statement $\texttt{arr}[\texttt{i}] = \texttt{0}$ thanks to the loop guard. In this way, we can prove the second part of the property, but not the first part.

Therefore, the two domains alone cannot prove the property of interest, while the Cartesian product can. In fact, it runs the two analyses in parallel, and at the end it takes from both domains the most precise result they get regarding the property to verify. Combining the two results, the entire property is proved to hold.

Reduced Product

Let us introduce a more complex example. It receives as input an integer variable k, and it creates an array with one element if $\texttt{k} \leq 0$, and of k elements otherwise. Then it initializes the first k elements to zero.

```
1 if(k <= 0)
2     arr = new Int[1];
3 else
4     arr = new Int[k];
5 for(i = 0; i < k; i++)
6     arr[i] = 0;
```

As before, we want to prove that when we perform $\texttt{arr}[\texttt{i}] = \texttt{0}$ we have that $\texttt{i} \geq 0$ and $\texttt{i} < \texttt{arr.length}$. In particular, the critical property is the second one, since the first one is already proved by Intervals as explained before.

If we analyze this example with the Cartesian product defined before, we obtain that (i) $(\{\texttt{arr.length} \mapsto [1..1], \texttt{k} \mapsto [-\infty..0]\}, \emptyset)$ in the then branch, and (ii) $(\{\texttt{arr.length} \mapsto [1..+\infty], \texttt{k} \mapsto [1..+\infty]\}, \{\texttt{arr.length} < \texttt{k}+1, \texttt{k} < \texttt{arr.length}+1\})$ in the else branch. Then, when we compute the upper bound of these two states, we obtain only $(\{\texttt{arr.length} \mapsto [1.. + \infty], \texttt{k} \mapsto [-\infty.. + \infty]\}, \emptyset)$. This leads to infer that $(\{\texttt{arr.length} \mapsto [1.. + \infty], \texttt{k} \mapsto [-\infty.. + \infty], \texttt{i} \mapsto [0.. + \infty], \}, \{\texttt{i} < \texttt{k} + 0\})$ inside the loop, but this cannot prove that $\texttt{i} < \texttt{arr.length}$.

We now define a specific reduction operator that refines the information tracked by the relational domain with Intervals. In particular, if Intervals track that $\mathtt{x} \mapsto [a..b], \mathtt{y} \mapsto [c..d]$ and we have that $b \neq +\infty \wedge c \neq -\infty$, then in the relational domain we introduce the constraint $\mathtt{x} < \mathtt{y} + j$ where $j = b - c + 1$. Thanks to this reduction operator, we infer that, in the `then` branch, $(\{\mathtt{arr.length} \mapsto [1..1], \mathtt{k} \mapsto [-\infty..0]\}, \mathtt{k} < \mathtt{arr.length} + 0)$. Thank to this reduction, when we join the two abstract states after the `if` statement we obtain that $(\{\mathtt{arr.length} \mapsto [1.. + \infty], \mathtt{k} \mapsto [-\infty.. + \infty]\}, \{\mathtt{k} < \mathtt{arr.length} + 1\})$. This leads to infer (when we analyze $\mathtt{arr}[\mathtt{i}] = 0$) that $(\{\mathtt{arr.length} \mapsto [1.. + \infty], \mathtt{k} \mapsto [-\infty.. + \infty], \mathtt{i} \mapsto [0.. + \infty], \}, \{\mathtt{k} < \mathtt{arr.length} + 1, \mathtt{i} < \mathtt{k} + 0\})$, and the information tracked by the relational domain proves that $\mathtt{i} < \mathtt{arr.length}$.

Reduced Cardinal Power

We slightly modify the previous example. In particular, we create an array of one element if $\mathtt{k} \leq 2$, and we initialize all the elements in the array from the third to the $(\mathtt{k} - 1)$-th element.

```
1 if(k <= 2)
2     arr = new Int[1];
3 else
4     arr = new Int[k];
5 for(i = 3; i < k; i++)
6     arr[i] = 0;
```

As in the previous example, the main challenge is to prove the second part of the property (that is, $\mathtt{i} < \mathtt{arr.length}$) when executing $\mathtt{arr}[\mathtt{i}] = 0$.

First of all, we show that the reduced product of Intervals and our relational domain is not in position to prove this property. In the `then` branch, the Interval domain tracks that $\mathtt{k} \in [-\infty..2]$ and $\mathtt{arr.length} \in [1..1]$. This information yields the strict lower bound relationship $\mathtt{k} < \mathtt{arr.length} + 2$ through the reduction operator we previously introduced. The abstract state associated to the `then` branch is $(\{\mathtt{k} \to [-\infty..2], \mathtt{arr.length} \to [1..1]\}, \{\mathtt{k} < \mathtt{arr.length} + 2\})$, while in the `else` branch we obtain $(\{\mathtt{k} \to [3..+\infty], \mathtt{arr.length} \to [3..+\infty]\}, \{\mathtt{arr.length} < \mathtt{k}+1, \mathtt{k} < \mathtt{arr.length} + 1\})$. When we compute the join between these two states, we obtain $(\{\mathtt{k} \to [-\infty.. + \infty], \mathtt{arr.length} \to [1.. + \infty]\}, \{\mathtt{k} < \mathtt{arr.length} + 2\})$. In fact, the join of the constraints $\mathtt{k} < \mathtt{arr.length} + 2$ and $\mathtt{k} < \mathtt{arr.length} + 1$ results in $\mathtt{k} < \mathtt{arr.length} + 2$. Finally, inside the `for` loop we know (from the Interval domain and its widening operator) that $\mathtt{i} \to [3..+\infty]$. Moreover, the loop guard implies that, when we perform $\mathtt{arr}[\mathtt{i}] = 0$, $\mathtt{i} < \mathtt{k}$ holds. From $\mathtt{i} < \mathtt{k}$ and $\mathtt{k} < \mathtt{arr.length} + 2$, we obtain that $\mathtt{i} < \mathtt{arr.length} + 1$, that is weaker than the property of interest $\mathtt{i} < \mathtt{arr.length}$.

Now consider the reduced cardinal power of Intervals on `k` as exponent and our relational domain as base. The `then` branch is associated to the abstract

state $[-\infty..2] \Rightarrow \{\texttt{k} < \texttt{arr.length} + 2\}$, and the `else` branch to $[3.. + \infty] \Rightarrow \{\texttt{arr.length} < \texttt{k}+1, \texttt{k} < \texttt{arr.length}+1\}$. The join between these two states simply creates a new abstract state which contains both informations, that is, $[-\infty..2] \Rightarrow \{\texttt{k} < \texttt{arr.length} + 2\}$ and $[3.. + \infty] \Rightarrow \{\texttt{arr.length} < \texttt{k} + 1, \texttt{k} < \texttt{arr.length} + 1\}$. When we enter the while loop, we have to consider the two cases separately:

- in the first case, $\texttt{k} = [-\infty..2]$. Then, the loop guard $\texttt{i} < \texttt{k}$ is surely evaluated to false: the loop is not executed, so we do not need to verify the property about array accesses. Therefore, when $\texttt{i} < \texttt{k}$ holds, we have that $[-\infty..2] \Rightarrow \bot$. This way, when we analyze $\texttt{arr}[\texttt{i}] = \texttt{0}$, we can discard this case;
- in the second case, we have that $\texttt{k} = [3.. + \infty]$ and $\texttt{i} < \texttt{k}$. Then, from the abstract state obtained after the `if` statements and assuming the loop guard, we know that $[3..+\infty] \Rightarrow \{\texttt{arr.length} < \texttt{k}+1, \texttt{k} < \texttt{arr.length}+1, \texttt{i} < \texttt{k}\}$. By combining $\texttt{k} < \texttt{arr.length} + 1$ and $\texttt{i} < \texttt{k}$, we obtain $\texttt{i} < \texttt{k} \leq \texttt{arr.length} \Rightarrow \texttt{i} < \texttt{arr.length}$, which is exactly the property we wanted to prove.

3

A Generic Framework for String Analysis

In this chapter we focus on the first of the three goals of our thesis, i.e. creating a unifying approach for string analysis.

Strings are widely used in modern programming languages in various scenarios. For instance, strings are used to build up SQL queries that are then executed. Malformed strings may lead to subtle bugs, as well as non-sanitized strings may rise security issues in an application. For these reasons, the application of static analysis to compute safety properties over string values at compile time is particularly appealing. Here we propose a generic approach for the static analysis of string values based on Abstract Interpretation. In particular, we design a suite of abstract semantics for strings, where each abstract domain tracks a different kind of information. We discuss the trade-off between efficiency and accuracy when using such domains to catch the properties of interest. In this way, the analysis can be tuned at different levels of precision and efficiency, and it can address specific properties.

This chapter closely follows the generic structure discussed in Section 1.6. In Section 3.1 we briefly explain the issues faced by string analysis and our contribution in solving them. Section 3.2 introduces two case studies which will be used throughout all the chapter to show the application of our techniques. Section 3.3 defines some string-specific notation. Section 3.4 defines the syntax of the string operators we will consider in the rest of the chapter. Section 3.5 introduces their concrete semantics, while in Section 3.6 the five abstract domains of our framework are formalized and used to analyze the case studies. In Section 3.7 more experimental results are presented. Finally, Section 3.8 discusses in depth the related work and Section 3.9 concludes.

Contents

[0]This chapter is partially derived from [44, 45, 47].

3.1 Introduction

Context

String analysis is a static analysis technique that determines the values that a string variable can hold at specific points in a program. This information is often useful to help program understanding, to detect and fix programming errors and security vulnerabilities, and to solve certain program verification problems. The great importance of string analysis is due to the current widespread use of strings in computer programs. Their applications vary from providing an output to a user to the construction of programs executed through reflection. For instance, in Java they are widely used to build up SQL queries, or to access information about the classes through reflection. Errors in such applications can cause a lot of damage: for example, when dealing with SQL queries, what happens if we execute the query "`DELETE FROM Table WHERE ID =` " + `id` when `id` is equal to "`10 OR TRUE`"? The content of `Table` would be permanently erased. It is clear that a wrong manipulation of strings could lead not only to subtle run-time errors, but to dramatic and permanent effects too [93].

Let us make a detailed overview of some contexts in which strings play a starring role.

- **Dynamic creation of SQL queries**. SQL (Structured Query Language) is a special-purpose programming language designed for managing data held in a relational database management system (RDBMS). Originally based upon relational algebra and tuple relational calculus, SQL consists of a data definition language and a data manipulation language. The scope of SQL includes data insert, query, update and delete, schema creation and modification, and data access control. The most common operation in SQL is the declarative SELECT statement. SELECT retrieves data from one or more tables, or expressions. A simple example of a query is: "`SELECT` $*$ `FROM books WHERE price` $>$ `100.0`

`ORDER BY title`", which retrieves all the books which cost more than 100, in ascending order by their title. SQL can be used both in a static or dynamic way. By static SQL we mean SQL code written once in the development phase when database and query structures are known. Static SQL is usually targeted at a specific database and in many cases gets stored in stored procedures. Many applications (especially Enterprise Applications) reach a stage where some dynamic data manipulation is required and static SQL techniques no longer suffice (for example, custom reports and filters designed by an application user, or when the databases structure itself is dynamic). A dynamic SQL statement is constructed at execution time, for which different conditions generate different SQL statements. Then, the query is sent to the database for execution (through the `sp_executesql` system stored procedure or through the `EXECUTE`() operator). Note that these SQL strings are not parsed for errors, because they are generated at execution time, and they may introduce security vulnerabilities into the database. Also, SQL strings can be hard to debug. However, sometimes they are perfect for certain scenarios. For all these reasons, a strong need for static analysis of programs generating dynamic SQL queries arises. Since SQL queries are built using string variables, this automatically translates into a need for string analysis.

- **Server-side scripting**. Server-side programming is one of the key technologies that support today's WWW environment. It makes possible to generate Web pages dynamically according to a user's request and to customize pages for each user. PHP is one of the most popular server-side scripting languages used to generate Web pages dynamically, even though many others exist (ASP.Net, Lua, Perl, Javascript, and so on). Unfortunately, the flexibility obtained by server-side programming makes it much harder to guarantee validity and security of dynamically generated pages. For example, it is well known that inappropriate treatment of input data causes vulnerabilities called cross-site scripting, which may cause leakage of critical information such as HTTP cookies. It is critical for a server-side program to prevent this kind of vulnerability and to guarantee security. By applying static program analysis to Web pages generated dynamically by a server-side program we can perform a static checking of properties of such pages. The approximation obtained by the analysis has many applications in checking the validity and security of a server-side program. Two well-known examples of applications are HTML validation (against the HTML specification) and the detection of cross-site scripting vulnerabilities in a server-side program. A string analyzer that approximates the string output of a program would then be useful to verify the soundness of such programs.

- **Dynamic generation of XML documents** Many interesting programming formalisms deal explicitly with XML documents. Examples range from

domain-specific languages, such as XSLT and XQuery, to general-purpose languages, such as Java in which XML documents may be handled by special frameworks or simply as plain text. Also, XML documents are often generated dynamically by programs. A common example is XHTML documents being generated by interactive Web services in response to requests from clients. Typically, there are no static guarantees that the generated documents are valid according to the DTD (Document Type Definition) for XHTML. In fact, a quick study of the outputs from many large commercial Web services shows that most generated documents are in fact invalid [36]. This is not a huge problem, since the browsers interpreting this output are able to render invalid documents. Increasingly, however, Web services will generate output in other XML languages for less tolerant clients, many of whom will themselves be Web services. Thus, it is certainly an interesting question to statically guarantee validity of dynamically generated XML: it is necessary to obtain a formal model of sets of XML documents or fragments, typically to represent conservative approximations of the possible results at specific program points. Several such models have been proposed, mainly based on the observation that formal tree languages capture many desired properties since XML documents are essentially trees [130, 100].

- **Reflection mechanism**. Reflection is the ability of a computer program to examine and modify the structure and behaviour (specifically the values, meta-data, properties and functions) of an object at runtime. Reflection is most commonly used in high-level virtual machine programming languages like Smalltalk and scripting languages and also in manifestly typed or statically typed programming languages such as Java, ML, Haskell, C# and Scala. In object oriented programming languages such as Java, reflection allows inspection of classes, interfaces, fields and methods at runtime without knowing the names of the interfaces, fields, methods at compile time. It also allows instantiation of new objects and invocation of methods. Reflection can also be used to adapt a given program to different situations dynamically. A language supporting reflection provides a number of features available at runtime that would otherwise be very difficult to accomplish in a lower-level language. Among these features, there are the abilities to: (i) convert a string matching the symbolic name of a class or function into a reference to or invocation of that class or function; and (ii) evaluate a string as if it were a source code statement at runtime. Since strings play a fundamental role in reflection (to get information from classes and invoking operation on them knowing only their name in the form of a string), a precise static analysis which determines the possible values of strings can have a positive impact on the verification of programs using reflection.

Consider also that, according to the Open Web Application Security Project

(OWASP)'s list that identifies the top ten most serious web application vulnerabilities [1], two of the top three vulnerabilities are:

1. *Injection flaws*, such as SQL, OS, and LDAP injection, which occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing unauthorized data.

2. Cross Site Scripting (*XSS flaws*), which occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victim' s browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

Also, in the list of other important security risks to consider, we find the Malicious File Execution (MFE), which in 2007 even appeared in the top three of the list. *MFE vulnerabilities* occur if developers directly use or concatenate potentially hostile input with file or stream functions, or improperly trust input files. On many platforms, frameworks allow the use of external object references, such as URLs or file system references. When the data is insufficiently checked, this can lead to arbitrary remote and hostile content being included, executed or invoked by the web server.

All these vulnerabilities involve string manipulation operations and they occur due to inadequate sanitization or inappropriate use of input strings provided by users.

State of the art

The interest on approaches that automatically analyze and discover bugs on strings is constantly arising. The state-of-the-art in this field is however still limited: techniques that rely on automata and use regular expressions are precise but slow, and they do not scale up [98, 99, 142, 150], while many other approaches are focused on particular properties or classes of programs [27, 85, 89, 122, 124, 127]. In Section 3.8 we will explore in more detail the existing literature.

As genericity and scalability are the main advantages of the Abstract Interpretation approach (since it allows to define analyses at different levels of precision and efficiency), in this chapter we investigate Abstract Interpretation as an alternative solution to the problem of string analysis.

Contribution and methodology

The main contribution of this chapter is the formalization of a *unifying* generic Abstract Interpretation-based framework for string analysis, and its instantiations with *five different domains* that track distinct types of information. In this way, we can tune the analysis at diversified levels of *accuracy*, yielding to faster and rougher, or slower but more precise string analyses.

```
1 var query = "SELECT $ ||
2   (RETAIL/100) FROM INVENTORY WHERE ";
3 if (l != null)
4   query = query + "WHOLESALE > " + l + " AND ";
5
6 var per = "SELECT TYPECODE, TYPEDESC FROM
7   TYPES WHERE NAME = 'fish' OR NAME = 'meat'";
8 query = query + "TYPE IN (" + per + ");";
9 return query;
```

(a) The first case study, `prog1`

```
1 var x = "a";
2 while(cond)
3   x = "0" + x + "1";
4 return x;
```

(b) The second case study, `prog2`

Figure 3.1: The case studies

The methodology is inspired by the approach adopted for numerical domains for static analysis of software [62, 87, 123]. The interface of a numerical domain is nowadays standard: each domain has to define the semantics of arithmetic expressions and Boolean conditions. Similarly, we consider a limited set of basic string operators supported by all the mainstream programming languages. The concrete semantics of these operators is approximated in different ways by the five different abstract domains. In addition, after 30 years of practice with numerical domains, it is clear that a monolithic domain precise on any program and property (e.g., Polyhedra [62]) gives up in terms of efficiency, while to achieve scalability we need specific approximations on a given property (e.g., Pentagons [114]) or class of programs (e.g., ASTRÉE [60]). With this scenario in mind, we develop several domains inside the same framework to tune the analysis at different levels of precision and efficiency w.r.t. the analyzed class of programs and property. Other abstractions are possible and welcomed, and we expect our framework to be generic enough to support them.

3.2 Case Studies

In this chapter, we chose to use the two case studies reported in Figures 3.1(a) and 3.1(b).

Table 3.1: Shortcuts of string constants in `prog1`

Name	String constant
s_1	"SELECT '$' || (RETAIL/100) FROM INVENTORY WHERE "
s_2	"WHOLESALE > "
s_3	" AND "
s_4	"SELECT TYPECODE, TYPEDESC FROM TYPES WHERE NAME = 'fish' OR NAME = 'meat'"
s_5	"TYPE IN ("
s_6	");"

The first case study, `prog1`, is taken from [85]: a `Java` servlet program generates and manipulates `SQL` queries as string data. In particular, consider a front-end Java servlet for a grocery store, with an SQL-driven database back-end. The database table INVENTORY contains a list of all items in the store. This table has three columns: RETAIL, WHOLESALE, and TYPE, among others. The RETAIL and WHOLESALE columns are both of type integer, indicating their respective costs in cents. The TYPE column is an integer, representing the product type-codes of the items in the table. In the grocery store database, there is another table TYPES used to look up type-codes. This table contains the columns TYPECODE, TYPEDESC, and NAME, of the types integer, varchar (a string), and varchar, respectively. The program constructs the string `query` to hold an SQL SELECT statement to return the prices of all the perishable items, and return the query for its execution. In the code, || is the concatenation operator, and the clause TYPE IN (...) checks whether the type-code TYPE matches any of the type-codes of the perishable items.

We are interested in checking if the SQL query resulting by the execution of such code is always well formed. For the sake of readability, we will use some shortcuts to identify the string constants of this program, as reported in Table 3.1.

The second program, `prog2`, modifies a string inside a `while` loop. The string `x` initially gets assigned with a string made by a single character (an a). At each iteration of the loop, a concatenation takes place: the string "0" is concatenated to the previous value of `x` and to the string "1". This means that the initial a character gets surrounded by a "0" and a "1" at each iteration. The number of 0s and 1s in the final value of the string `x` depends on the number of iterations executed. For example, after 4 iterations the value of `x` would be $0000a1111$. Generalizing, this program produces strings of the form "$0^n a 1^n$", where n is the number of iterations executed (which could also be zero, in which case `x` would simply have value "a"). However, the condition of the loop cannot be statically evaluated, so it impossible to know at compile time how many iterations will be executed. This example is quite standard in string analysis and is very useful to compare the precision of various approaches, because of the difficulty added by the unknown condition of the loop.

$$V \in \mathcal{V}, s, s_1, s_2 \in \mathcal{S}, b, e \in \mathbb{N}, c \in \mathcal{C}$$
$$E := V | \texttt{new String}(s) | \texttt{concat}(s_1, s_2) | \texttt{substring}(b, e, s)$$
$$B := \texttt{contains}(c, E) | B \texttt{ and } B | \texttt{not } B | B \texttt{ or } B$$
$$P := V = E | \texttt{if}(B) \texttt{ then } P \texttt{ else } P | \texttt{while}(B)\ P | P; P$$

Figure 3.2: Syntax

3.3 Notation

In addition to the notation presented in Chapter 2, we introduce here some additional definitions specific for this chapter.

We will omit the quotation marks (“”) when writing strings and the context is not ambiguous (e.g., *abc* instead of “*abc*”). Similarly, we will omit the apices ($'$) when writing characters (e.g., a instead of $'a'$).

Let $char(s)$ be a function that returns the set of characters contained in the string s in input, while $charAt_{\mathsf{i}}(s)$ is a function that returns the character at index i in s.

Let $trunc(s, n)$ be a function which, given a string s and a positive number n, returns the truncation of s at index n, i.e. all characters from index n onwards are removed from the string. Note that, after the application of $trunc(s, n)$, the resulting string is made by n characters.

3.4 Language Syntax

Let $\mathcal{V}$ be a finite set of variables, and $\mathcal{C}, \mathcal{S}$ the set of all characters and strings, respectively. Figure 3.2 defines the language supported by our analysis.

We focus on programs dealing with operations over string-valued variables (even though numerical variables could be easily added to the syntax, and included in the analysis working in cooperation with an already existent abstract domain). Therefore, we consider expressions built through some of the most common string operators. The problem is that different languages define different operators on strings, and usually each language supports a huge set of such operators: in Java 1.6 the `String` class contains 65 methods and 15 constructors, `System.Text` in .Net contains about 12 classes that work with Unicode strings, and PHP provides 111 string functions. Considering all these operators would be quite overwhelming, and in addition the most part of them perform similar actions using slightly different data. We decided then to restrict our focus on a small but representative set of common operators. We chose these operators analyzing some case studies, and they are supported by all the mainstream programming languages. In particular, we chose to

support the creation of a string (`new String(str)`), the concatenation of two strings (`concat(s1, s2)`), the creation of a substring from a string ($\texttt{substring}_{\texttt{b}}^{\texttt{e}}(\texttt{s})$), and the boolean check of character inclusion in a string ($\texttt{contains}_{\texttt{c}}(\texttt{s})$) [1]. Another common operation is the reading of some input from the user with the `readLine()` statement, but we do not include this operator because its abstract semantics is the same in any abstract domain we could define, i.e. it simply returns the top element $\top$ of the considered domain. Boolean conditions can be combined as usual with logical operators (and, or, not). As for statements, we support the assignment of an string to a variable, `if − then − else`, `while` loops, and concatenation. The resulting syntax is simple and limited, but other operators can be easily added to our semantics. For each operator, this would mean to define its concrete semantics, and its approximations on the different domains we will introduce.

An informal description about the strings operations supported by our analysis follows:

- `new String(str)` (where `str` is a sequence of characters) creates a new constant string;
- `concat(s1, s2)` (where `s1` and `s2` are strings) concatenates two strings. Note that the concatenation operation can also be written with the + operator: `concat(s1, s2)` is the same as `s1 + s2`.
- $\texttt{substring}_{\texttt{b}}^{\texttt{e}}(\texttt{s})$ (where `s` is a string, and `b` and `e` are integer values representing the first and last index to use for the substring creation) extracts a substring from a given string;
- $\texttt{contains}_{\texttt{c}}(\texttt{s})$ (where `s` is a string and `c` is a character) checks if a string contains a specific character.

In Tables 3.2 we present the syntax of the corresponding string operations in three commonly used programming languages, i.e. Java, C#, PHP.

3.5 Concrete Domain and Semantics

Given an alphabet K (that is, a finite set of characters), we define strings as *sequences of characters*. Formally, following the notation of Chapter 2:

$$\mathsf{S} = \mathsf{K}^*$$

A string variable in our program could have different values in different executions, and the goal of the Abstract Interpretation approach is to approximate all

[1] Note that here we considered the operator $\texttt{contains}_{\texttt{c}}$ which checks if a certain *character* is contained in a string, but in [44] we presented the semantics of the extended version of this operator, i.e., $\texttt{contains}_{\texttt{seq}}$ which checks if a certain *sequence of characters* `seq` is contained in a string. In [44] we also presented two additional operators, i.e. $\texttt{indexOf}_{\texttt{c}}$ and $\texttt{lastIndexOf}_{\texttt{c}}$.

Table 3.2: String operators in Java , C# and PHP

Operator	Java	C#	PHP
`new String(str)`	new String(str) or "str"	new String(str) or "str"	"str"
`concat(s1, s2)`	s1.concat(s2) or s1 + s2	string.concat(s1,s2) or s1+s2	s1 . s2
$\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}(\texttt{s})$	s.substring(b, e)	s.substring(b, e)	substr(s, b, e-b)
$\texttt{contains}_{\mathsf{c}}(\texttt{s})$	s.contains(c)	s.contains(c)	preg _ match(c, s)

Table 3.3: Concrete semantics

$$
\begin{aligned}
\mathbb{S}[\![\texttt{new String(str)}]\!]() &= \{\texttt{str}\} \\
\mathbb{S}[\![\texttt{concat}]\!](\mathsf{S}_1, \mathsf{S}_2) &= \{\mathsf{s}_1\mathsf{s}_2 : \mathsf{s}_1 \in \mathsf{S}_1 \wedge \mathsf{s}_2 \in \mathsf{S}_2\} \\
\mathbb{S}[\![\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}]\!](\mathsf{S}_1) &= \{\mathsf{c}_\mathsf{b}..\mathsf{c}_\mathsf{e} : \mathsf{c}_1..\mathsf{c}_\mathsf{n} \in \mathsf{S}_1 \wedge \mathsf{n} \geq \mathsf{e} \wedge \mathsf{b} \leq \mathsf{e}\} \\
\mathbb{B}[\![\texttt{contains}_{\mathsf{c}}]\!](\mathsf{S}_1) &= \begin{cases} \mathsf{true} & \text{if } \forall \mathsf{s} \in \mathsf{S}_1 : \mathsf{c} \in \mathit{char}(\mathsf{s}) \\ \mathsf{false} & \text{if } \forall \mathsf{s} \in \mathsf{S}_1 : \mathsf{c} \notin \mathit{char}(\mathsf{s}) \\ \top_{\mathsf{B}} & \text{otherwise} \end{cases}
\end{aligned}
$$

these values (potentially infinite) in a finite, computable, and efficient manner. For this reason, the concrete domain is simply made of sets of strings: the concrete lattice is then the power-set of the strings set S. As explained in Section 2.4, when the lattice is the power-set of a set, the other operators immediately follow: the partial order is the set inclusion $\subseteq$, the least upper bound corresponds to set union $\cup$, the greatest lower bound corresponds to set intersection $\cap$, the top element $\top$ is the set itself, while the bottom element $\bot$ is $\emptyset$. The complete definition of the lattice $\mathcal{S}$ is then:

$$\mathcal{S} = \langle \wp(\mathsf{S}), \subseteq, \cup, \cap, \mathsf{S}, \emptyset \rangle$$

We can now define the concrete semantics of the language introduced in Section 3.4. For the statements operations and the logic combination of boolean conditions we refer to the usual semantics of the classical Abstract Interpretation framework. Then, we have to specify only the semantics (both concrete and abstract) of operators dealing with strings. We formalize the concrete semantics in Table 3.3.

For the first three statements (creation, concatenation, substring), we define the semantics $\mathbb{S}$ that, given the statement and eventually some sets of concrete string values in S (containing the values of the arguments of the statement), returns a set of concrete strings resulting from that operation. In particular:

- `new String(str)` returns a singleton containing `str`;

- `concat` returns all the possible concatenations of a string taken from the first set and a string taken from the second set (we denote by $\mathsf{s}_1\mathsf{s}_2$ the concatenation of strings s_1 and s_2);

- $\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}$ returns all the substrings from the `b`-th to `e`-th character of the given strings. Note that if one of the strings is too short, there is no substring for it in the resulting set, since this would cause a runtime error.

For $\texttt{contains}_{\mathsf{c}}$ we define a particular semantics $\mathbb{B} : \wp(\mathsf{S}) \rightarrow \{\mathsf{true}, \mathsf{false}, \top_{\mathsf{B}}\}$. Given a set of strings, the semantics of this operator returns true if all the strings contain the character `c`, false if none contains this character, and $\top_{\mathsf{B}}$ otherwise. This special boolean value represents a situation in which the boolean condition may be evaluated to true some times, and to false other times, depending on the string in S_1 we are considering. Therefore, we define a partial order $\geq_{\mathsf{B}}$ over these values such that (i) $\forall b \in \{\mathsf{true}, \mathsf{false}, \top_{\mathsf{B}}\} : \top_{\mathsf{B}} \geq_{\mathsf{B}} b$, (ii) $\mathsf{true} \geq_{\mathsf{B}} \mathsf{true}$, and (iii) $\mathsf{false} \geq_{\mathsf{B}} \mathsf{false}$.

3.6 Abstract Domains and Semantics

Abstract Interpretation is based on the consideration that it is impossible to track both sound and complete information about all possible executions of a program at compile time: it is thus necessary to introduce some kind of *approximation*. The approximation level (and, consequently, the precision and performance of the analysis) depends on the abstract domain used to analyze the program. In fact, given a specific concrete object, there are many possible approximations of it, which depend on the features you focus on: this means that you can build more than one abstract domain for the same object. Each abstract domain will preserve a different kind of information and it could be useful to verify a specific property, or to satisfy some performance constraints.

In our case, we want to create a framework of abstractions for strings. For this purpose, we have to understand first what is the *relevant* information contained in a string. A string can be defined as "*an ordered sequence of characters*". Then, the main features of a string are essentially two: the *characters* which compose it, and the *order* with which they are positioned. Each domain of our framework will consider a different subset of this concrete information. The first level of approximation we will introduce is an abstract representation in which we maintain all the information we have about characters inclusion but nothing about order (Section 3.6.1). This approximation would behave well in programs which use string operators like `contains`. The second kind of abstract representation we will define keeps some information about the order (in a limited part of the string, i.e. beginning or end) but not about inclusion in itself. This representation (Section 3.6.2) could be particularly useful for programs which use the `substring` operator. Finally, we will present two abstractions that track information on both character inclusion and

order: the abstract domain presented in Section 3.6.3 is inspired by regular expressions, while the abstract representation of Section 3.6.4 is based on a data structure resembling trees with backward arcs.

The domains introduced in Sections 3.6.3 and 3.6.4 are strictly more precise than the ones presented in Sections 3.6.1 and 3.6.2, but they are less efficient as well. Nevertheless, in some contexts the less precise domains would be precise enough to prove some properties of interest, while in other cases we would need the more complex domains. Therefore, one can tune the analysis at different levels of precision and efficiency by choosing different domains.

In the following sections we will present the five domains of our framework. For each domain, we will follow the same presentation order (already sketched in Section 1.6) and define:

- lattice elements
- partial order with top and bottom elements
- lub and glb operators
- abstraction and concretization functions [2]
- widening operator (if required)

For each domain we will include the following lemmas and theorems (together with their proofs):

- form of the domain (complete lattice, sup-semi lattice, cpo, etc.)
- correctness of lub and glb operators
- Galois connection between abstraction and concretization function

Then, for each domain we will define the abstract semantics of the string operators introduced in Section 3.4 approximating their concrete semantics presented in Section 3.5. Each abstract operation is formally proved to be sound with respect to the Galois connection of the domain. Lastly, each domain is applied to the two case studies of Section 3.2.

[2]Note that we will explicitly define only the concretization function, while the abstraction function will be implicitly characterized by the Galois connection, in order to guarantee the soundness of the Galois connection itself through Theorem 2.8.3. However, an informal (and more intuitive) characterization of the abstraction function is contained in the definition of the abstract semantics of the `new String` operator.

3.6.1 Character Inclusion

The first abstract domain approximates strings with the characters we know the strings *surely* contain, and ones that they *could* contain. This information could be particularly useful if the indices extrapolated from a string with operators like `indexOf(c)` could be used to cut the string (because it is interesting to know if the index is invalid, i.e., -1).

In this domain, denoted by $\overline{\mathcal{CI}}$, a string will be represented by a pair of sets, the set of *certainly* contained characters $\overline{\mathsf{C}}$ and the set of *maybe* contained characters $\overline{\mathsf{MC}}$:

$$\overline{\mathcal{CI}} = \{(\overline{\mathsf{C}}, \overline{\mathsf{MC}}) : \overline{\mathsf{C}}, \overline{\mathsf{MC}} \in \wp(\mathsf{K}) \wedge \overline{\mathsf{C}} \subseteq \overline{\mathsf{MC}}\} \cup \bot_{\overline{\mathcal{CI}}}$$

Partial Order

The partial order $\leq_{\overline{\mathcal{CI}}}$ on $\overline{\mathcal{CI}}$ is defined by

$$(\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1) \leq_{\overline{\mathcal{CI}}} (\overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_2) \Leftrightarrow (\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1) = \bot_{\overline{\mathcal{CI}}} \vee (\overline{\mathsf{C}}_1 \supseteq \overline{\mathsf{C}}_2 \wedge \overline{\mathsf{MC}}_1 \subseteq \overline{\mathsf{MC}}_2)$$

This is because the more information we have on the string (that is, the more characters are certainly contained and the fewer characters are maybe contained), the fewer strings we are representing. Consequently, the top element of the lattice is:

$$\top_{\overline{\mathcal{CI}}} = (\emptyset, \mathsf{K})$$

while the bottom element of the lattice is defined as:

$$\bot_{\overline{\mathcal{CI}}} = \{(\overline{\mathsf{C}}, \overline{\mathsf{MC}}) : \overline{\mathsf{C}} \not\subseteq \overline{\mathsf{MC}}\}$$

in order to represent a computational error. In fact, any character that is certainly contained in the string *must* belong also to the set of maybe contained characters. Thus, with this definition of $\bot_{\overline{\mathcal{CI}}}$, we ensure that its concretization will be the empty set. The partial order $\leq_{\overline{\mathcal{CI}}}$ is specifically built to guarantee that $\bot_{\overline{\mathcal{CI}}} \leq_{\overline{\mathcal{CI}}} A \; \forall A \in \overline{\mathcal{CI}}$, in order to respect the definition of bottom element.

Least Upper Bound and Greatest Lower Bound

The definition of these two operators is induced by the definition of the partial order. Formally, the least upper bound is defined by:

$$\sqcup_{\overline{\mathcal{CI}}}((\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1), (\overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_2)) = (\overline{\mathsf{C}}_1 \cap \overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_1 \cup \overline{\mathsf{MC}}_2)$$

Similarly, the greatest lower bound, instead, is defined by:

$$\sqcap_{\overline{\mathcal{CI}}}((\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1), (\overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_2)) = \begin{cases} (\overline{\mathsf{C}}_1 \cup \overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_1 \cap \overline{\mathsf{MC}}_2) & \text{if } \overline{\mathsf{C}}_1 \subseteq \overline{\mathsf{MC}}_2 \wedge \overline{\mathsf{C}}_2 \subseteq \overline{\mathsf{MC}}_1 \\ \bot_{\overline{\mathcal{CI}}} & \text{otherwise} \end{cases}$$

Lemma 3.6.1. *$\sqcup_{\overline{\mathcal{CI}}}$ is the least upper bound operator and $\sqcap_{\overline{\mathcal{CI}}}$ is the greatest lower bound operator.*

Proof. The fact that $\sqcup_{\overline{\mathcal{CI}}}$ and $\sqcap_{\overline{\mathcal{CI}}}$ are, respectively, the least upper bound and the greatest lower bound operator follows from basic properties of set union and intersection.

□

Lemma 3.6.2. *The abstract domain $\overline{\mathcal{CI}}$ is a complete lattice.*

Proof. The proof follows straightforwardly from the fact that, for any set C, $\langle \wp(\mathsf{C}), \subseteq \rangle$ and $\langle \wp(\mathsf{C}), \supseteq \rangle$ are both complete lattices.

□

Abstraction and Concretization Functions

The concretization function maps an abstract element to a set of strings. Given an abstract element $(\overline{\mathsf{C}}, \overline{\mathsf{MC}})$, the resulting strings will have to (i) contain at least all the characters in $\overline{\mathsf{C}}$, and (ii) contain at most the characters in $\overline{\mathsf{MC}}$. This is defined as follows:

$$\gamma_{\overline{\mathcal{CI}}}(\overline{\mathsf{C}}, \overline{\mathsf{MC}}) = \{s : c_1 \in \overline{\mathsf{C}} \Rightarrow c_1 \in s \wedge c_2 \in s \Rightarrow c_2 \in \overline{\mathsf{MC}}\}$$

Theorem 3.6.3. *Let the abstraction function $\alpha_{\mathcal{CI}}$ be defined by $\alpha_{\mathcal{CI}} = \lambda \mathsf{Y}.\ \sqcap_{\mathcal{CI}} \{(\overline{\mathsf{C}}, \overline{\mathsf{MC}}) : \mathsf{Y} \subseteq \gamma_{\mathcal{CI}}((\overline{\mathsf{C}}, \overline{\mathsf{MC}}))\}$.*
Then $\langle \wp(\mathsf{S}), \subseteq \rangle \underset{\alpha_{\mathcal{CI}}}{\overset{\gamma_{\mathcal{CI}}}{\leftrightarrows}} \langle \mathcal{CI}, \leq_{\mathcal{CI}} \rangle$.

Proof. By Theorem 2.8.3 we only need to prove that $\gamma_{\mathcal{CI}}$ is a complete meet morphism. Formally, we have to prove that $\gamma_{\mathcal{CI}}(\underset{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}}{\sqcap_{\mathcal{CI}}} (\overline{\mathsf{C}}, \overline{\mathsf{MC}})) = \bigcap_{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}} \gamma_{\mathcal{CI}}(\overline{\mathsf{C}}, \overline{\mathsf{MC}})$.

$\gamma_{\mathcal{CI}}(\underset{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}}{\sqcap_{\mathcal{CI}}} (\overline{\mathsf{C}}, \overline{\mathsf{MC}}))$
by Definition of $\sqcap_{\mathcal{CI}}$
$= \gamma_{\mathcal{CI}}(\bigcup_{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}} \overline{\mathsf{C}}, \bigcap_{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}} \overline{\mathsf{MC}})$
by Definition of $\gamma_{\mathcal{CI}}$
$= \{s : c_1 \in \bigcup_{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}} \overline{\mathsf{C}} \Rightarrow c_1 \in s \wedge c_2 \in s \Rightarrow c_2 \in \bigcap_{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}} \overline{\mathsf{MC}}\}$
by logic rules of set theory
$= \{s : \forall(\overline{\mathsf{C}}, \overline{\mathsf{MC}}) \in \overline{\mathsf{X}} : c_1 \in \overline{\mathsf{C}} \Rightarrow c_1 \in s \wedge c_2 \in s \Rightarrow c_2 \in \overline{\mathsf{MC}}\}$
by Definition of $\cap$
$= \bigcap_{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}} \{s : c_1 \in \overline{\mathsf{C}} \Rightarrow c_1 \in s \wedge c_2 \in s \Rightarrow c_2 \in \overline{\mathsf{MC}}\}$
by Definition of $\gamma_{\mathcal{CI}}$
$= \bigcap_{(\overline{\mathsf{C}},\overline{\mathsf{MC}})\in\overline{\mathsf{X}}} \gamma_{\mathcal{CI}}(\overline{\mathsf{C}}, \overline{\mathsf{MC}})$

□

Table 3.4: The abstract semantics of $\overline{\mathcal{CI}}$

$$\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{new String(str)}]\!]() = (\mathit{char}(\mathsf{str}), \mathit{char}(\mathsf{str}))$$
$$\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{concat}]\!]((\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1), (\overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_2)) = (\overline{\mathsf{C}}_1 \cup \overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_1 \cup \overline{\mathsf{MC}}_2)$$
$$\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}]\!]((\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1)) = (\emptyset, \overline{\mathsf{MC}}_1)$$
$$\overline{\mathbb{B}_{\mathcal{CI}}}[\![\texttt{contains}_{\mathsf{c}}]\!]((\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1)) = \begin{cases} \mathsf{true} & \text{if } \mathsf{c} \in \overline{\mathsf{C}}_1 \\ \mathsf{false} & \text{if } \mathsf{c} \notin \overline{\mathsf{MC}}_1 \\ \top_{\mathsf{B}} & \text{otherwise} \end{cases}$$

Widening Operator

The widening operator $\nabla_{\overline{\mathcal{CI}}} : (\overline{\mathcal{CI}} \times \overline{\mathcal{CI}}) \to \overline{\mathcal{CI}}$ is defined by:

$$(\overline{\mathsf{C}_1}, \overline{\mathsf{MC}_1}) \nabla_{\overline{\mathcal{CI}}} (\overline{\mathsf{C}_2}, \overline{\mathsf{MC}_2}) = (\overline{\mathsf{C}_1}, \overline{\mathsf{MC}_1}) \sqcup_{\overline{\mathcal{CI}}} (\overline{\mathsf{C}_2}, \overline{\mathsf{MC}_2})$$

because in domains with finite height the least upper bound operator is also a widening operator since it converges in finite time. Our domain has finite height, since the height of the power-set lattice of a set S based on $\subseteq$ or $\supseteq$ is $|\mathsf{S}| + 1$, and we always consider only finite alphabets.

Semantics

Table 3.4 defines the abstract semantics (in the abstract domain $\overline{\mathcal{CI}}$) of the operators introduced in Section 3.4. We denote by $\overline{\mathbb{S}_{\mathcal{CI}}}$ and $\overline{\mathbb{B}_{\mathcal{CI}}}$ the abstract counterparts of $\mathbb{S}$ and $\mathbb{B}$, respectively.

The abstract operations are defined as follows:

- When we evaluate a string constant (`new String(str)`), we know that the characters that are surely or maybe included are exactly the ones that appear in the string `str`.
- The `concat` operator takes in input two strings and concatenates them. If a character appears (or could appear) in one of the two input strings, then it will appear (or it could appear) in the resulting string too. For this reason, we employ set union.
- The `substring` operator returns a new string that is a substring of the string s in input. The $\overline{\mathsf{MC}}_1$ set remains the same, while the only sound approximation of the certainly contained characters is $\emptyset$, because we do not know the position of the certainly contained characters inside s.
- The `contains` operator returns true if and only if the string in input (let it be s) contains the specified character (c). Its semantics is quite precise, as it checks if a character is surely contained or not contained respectively through $\overline{\mathsf{C}}_1$ and $\overline{\mathsf{MC}}_1$.

We now prove the soundness of the abstract operations defined above.

Theorem 3.6.4 (Soundness of the abstract semantics)**.** *$\overline{\mathbb{S}_{\mathcal{CI}}}$ and $\overline{\mathbb{B}_{\mathcal{CI}}}$ are sound over-approximations of $\mathbb{S}$ and $\mathbb{B}$, respectively. Formally, $\gamma_{\overline{\mathcal{CI}}}(\overline{\mathbb{S}_{\mathcal{CI}}}[\![\mathsf{s}]\!](\overline{\mathsf{IC}})) \supseteq \{\mathbb{S}[\![\mathsf{s}]\!](\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{CI}}}(\overline{\mathsf{IC}})\}$ and $\overline{\mathbb{B}_{\mathcal{CI}}}[\![\mathsf{s}]\!](\overline{\mathsf{IC}}) \geq_{\mathsf{B}} \{\mathbb{B}[\![\mathsf{s}]\!](\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{CI}}}(\overline{\mathsf{IC}})\}$.*

Proof. We prove the soundness separately for each operator.

- $\gamma_{\overline{\mathcal{CI}}}(\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{new String(str)}]\!]()) \supseteq \{\mathbb{S}[\![\texttt{new String(str)}]\!]()\}$ follows immediately from the definition of $\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{new String(str)}]\!]()$ and of $\gamma_{\overline{\mathcal{CI}}}$.

- Consider the binary operator `concat`. Let $\overline{\mathsf{a}}_1 = (\overline{\mathsf{C}}_1, \overline{\mathsf{MC}}_1), \overline{\mathsf{a}}_2 = (\overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_2)$ be two abstract states. We have to prove that $\gamma_{\overline{\mathcal{CI}}}(\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{concat}]\!](\overline{\mathsf{a}}_1, \overline{\mathsf{a}}_2)) \supseteq \{\mathbb{S}[\![\texttt{concat}]\!](\mathsf{c}_1, \mathsf{c}_2) : \mathsf{c}_1 \in \gamma_{\overline{\mathcal{CI}}}(\overline{\mathsf{a}}_1) \wedge \mathsf{c}_2 \in \gamma_{\overline{\mathcal{CI}}}(\overline{\mathsf{a}}_2)\}$. A generic element $\mathsf{c}_1 \in \gamma_{\overline{\mathcal{CI}}}(\overline{\mathsf{a}}_1)$ is a string which contains at least one occurrence of each character of $\overline{\mathsf{C}}_1$ and which characters all belong to $\overline{\mathsf{MC}}_1$; the same goes for $\mathsf{c}_2 \in \gamma(\overline{\mathsf{a}}_2)$. The concatenation of c_1 and c_2 then, by definition of $\mathbb{S}$, produces a string which contains at least one occurrence of each character of $\overline{\mathsf{C}}_1$ and of $\overline{\mathsf{C}}_2$, and which characters all belong to $\overline{\mathsf{MC}}_1$ or $\overline{\mathsf{MC}}_2$. Then, this string belongs to $\gamma_{\overline{\mathcal{CI}}}(\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{concat}]\!](\overline{\mathsf{a}}_1, \overline{\mathsf{a}}_2))$, because $\overline{\mathbb{S}_{\mathcal{CI}}}[\![\texttt{concat}]\!](\overline{\mathsf{a}}_1, \overline{\mathsf{a}}_2) = (\overline{\mathsf{C}}_1 \cup \overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_1 \cup \overline{\mathsf{MC}}_2)$ by definition of $\overline{\mathbb{S}_{\mathcal{CI}}}$. Then $\gamma_{\overline{\mathcal{CI}}}(\overline{\mathsf{C}}_1 \cup \overline{\mathsf{C}}_2, \overline{\mathsf{MC}}_1 \cup \overline{\mathsf{MC}}_2)$ contains, by definition of $\gamma_{\overline{\mathcal{CI}}}$, all strings which contain at least one occurrence of each character of $\overline{\mathsf{C}}_1 \cup \overline{\mathsf{C}}_2$, and which characters all belong to $\overline{\mathsf{MC}}_1 \cup \overline{\mathsf{MC}}_2$.

- Consider the unary operator $\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}$. Our theorem trivially holds since the abstract semantics returns the top element of $\mathcal{CI}$, that concretizes to all the possible strings. This trivially overapproximates any possible result of the concrete semantics.

- Consider the unary operator $\texttt{contains}_{\mathsf{c}}$ and let $\overline{\mathsf{a}} = (\overline{\mathsf{C}}, \overline{\mathsf{MC}})$ be an abstract state. Considering the character c, we have three cases:

  - If $\mathsf{c} \in \overline{\mathsf{C}}$, all the strings belonging to $\gamma_{\overline{\mathcal{CI}}}(\overline{\mathsf{a}})$ contain at least one occurrence of c by definition of $\gamma_{\overline{\mathcal{CI}}}$. Then, the concrete semantics returns always `true` on this set. On the other hand, the abstract semantics on $\overline{\mathsf{a}}$ returns the $\overline{\mathsf{true}}$ value of the boolean domain, so it soundly approximates the concrete semantics.
  - If $\mathsf{c} \in \overline{\mathsf{MC}}$ and $\mathsf{c} \notin \overline{\mathsf{C}}$, then the abstract semantics returns $\top_{\mathsf{B}}$ that trivially approximates any possible result of the concrete semantics.
  - If $\mathsf{c} \notin \overline{\mathsf{C}} \wedge \mathsf{c} \notin \overline{\mathsf{MC}}$, no string belonging to $\gamma(\overline{\mathsf{a}})$ will contain the character. The concrete semantics will therefore return always `false`, and the abstract semantics on $\overline{\mathsf{a}}$ returns the $\overline{\mathsf{false}}$ value of the boolean domain as well.

□

#I	Var	$\overline{\mathcal{CI}}$
1	query	$\alpha_{\mathcal{CI}}(\mathsf{s}_1)$
3	l	$(\emptyset, \mathsf{K})$
4	query	$(\pi_1(\alpha_{\mathcal{CI}}(\mathsf{s}_1)) \cup \pi_1(\alpha_{\mathcal{CI}}(\mathsf{s}_2)) \cup \pi_1(\alpha_{\mathcal{CI}}(\mathsf{s}_3)), \mathsf{K})$
5	query	$(\pi_1(\alpha_{\mathcal{CI}}(\mathsf{s}_1)), \mathsf{K})$
6	per	$\alpha_{\mathcal{CI}}(\mathsf{s}_4)$
8	query	$(\pi_1(\alpha_{\mathcal{CI}}(\mathsf{s}_1)) \cup \pi_1(\alpha_{\mathcal{CI}}(\mathsf{s}_4)) \cup \pi_1(\alpha_{\mathcal{CI}}(\mathsf{s}_5)) \cup \pi_1(\alpha_{\mathcal{CI}}(\mathsf{s}_6)), \mathsf{K})$

(a) Analysis of prog1

#I	Var	$\overline{\mathcal{CI}}$
1	x	$(\{a\}, \{a\})$
3	x	$(\{0, a, 1\}, \{0, a, 1\})$
4	x	$(\{a\}, \{0, a, 1\})$

(b) Analysis of prog2

Figure 3.3: The results of $\overline{\mathcal{CI}}$

Case studies

Consider now the two case studies introduced in Section 3.2.

The results of the analysis of prog1 using $\overline{\mathcal{CI}}$ are depicted in Figure 3.3(a). At the beginning, the variable query is related to a state that contains the abstraction of s_1 [3]. The value of l is unknown, so we must compute the least upper bound between the abstract values of query after instructions 1 and 4. The set $\overline{\mathsf{C}}$ of query after instruction 4 contains all the character of $\mathsf{s}_1, \mathsf{s}_2$ and s_3, because they are all concatenated; the $\overline{\mathsf{MC}}$ set instead is K because of the concatenation with l. Then, after the if statement (line 5) the abstract value of query contains the abstraction of s_1 in $\overline{\mathsf{C}}$, and K in $\overline{\mathsf{MC}}$ (because of l). The variable per is related (line 6) to a state that contains the abstraction of s_4. At line 8, query is concatenated to $\mathsf{s}_4, \mathsf{s}_5$ and s_6. Then, at the end of the given code, query surely contains the characters of s_1, s_4, s_5, and s_6, and it may contain any character, since we possibly concatenated in query an input string (the l variable).

As for prog2, in Figure 3.3(b) we see that after instruction 1 x surely contains the character 'a'. After the first iteration of the loop (line 3), x surely contains 'a', '0' and '1'. At line 4 we report the least upper bound between the value of x *before* entering the loop (line 1) and the value *after* the loop (line 4): variable x surely contains the character 'a', and it also may contain the characters '0' and '1'. This is the final result of the program. In fact, we do not know the value of cond, so we cannot know beforehand how many iterations will be done by the loop. In such cases, we have to use the widening to reach the convergence. Here the analysis converges immediately after the second iteration, since the abstract value obtained after two iterations (that is, $(\{0, a, 1\}, \{0, a, 1\})$) is the same as the one obtained after one iteration.

[3] The abstraction function $\alpha_{\mathcal{CI}}(\mathsf{s})$ corresponds to the abstract semantics of the new String(s) operator: given a concrete string s, it returns the pair of sets $(char(\mathsf{s}), char(\mathsf{s}))$.

3.6.2 Prefix and Suffix

In this section, we start by defining a domain that abstracts strings through their *prefix*; after that, we will also define its mirror image, i.e. a domain which abstracts strings through their *suffix*.

We represent a prefix by a sequence of characters followed by an asterisk $*$. The asterisk represents any string (the empty string ϵ included). For example, $abc*$ represents all the strings which begin with abc, including abc itself. Since the asterisk $*$ at the end of the representation is always present, we do not include it in the domain and consider abstract elements made only of sequence of characters. Formally:

$$\overline{\mathcal{PR}} = \mathsf{K}^* \cup \bot_{\overline{\mathcal{PR}}}$$

Partial Order

The partial order is defined by:

$$\overline{\mathsf{p}}_1 \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}_2 \Leftrightarrow \overline{\mathsf{p}}_1 = \bot_{\overline{\mathcal{PR}}} \vee (len(\overline{\mathsf{p}}_2) \leq len(\overline{\mathsf{p}}_1) \wedge (\forall \mathsf{i} \in [0, len(\overline{\mathsf{p}}_2) - 1] : \overline{\mathsf{p}}_2[\mathsf{i}] = \overline{\mathsf{p}}_1[\mathsf{i}]))$$

An abstract string $\overline{\mathsf{S}}$ is $\leq_{\overline{\mathcal{PR}}}$ than another abstract string $\overline{\mathsf{T}}$ if $\overline{\mathsf{T}}$ is a prefix of $\overline{\mathsf{S}}$ or if $\overline{\mathsf{S}}$ is the bottom $\bot_{\overline{\mathcal{PR}}}$ of the domain. The top element is $*$, since $*$ is the empty prefix, which is prefix of any other prefix. Instead the bottom value is the special element $\bot_{\overline{\mathcal{PR}}}$.

The definition of bottom as a special element descends from the fact that the domain has an infinite height. In fact, given any prefix, we can always add a character at the end of it, thus obtaining a new prefix, longer (and smaller according to the order $\leq_{\overline{\mathcal{PR}}}$) than the first one. For this reason, we cannot find a valid element of the domain which is smaller than any other element.

However, the domain respects the ascending chain condition (ACC), and we do not need to define a widening operator to ensure the convergence of the analysis. In fact, given a certain prefix $\overline{\mathsf{p}}$, where $len(\overline{\mathsf{p}}) = n$, the ascending chain starting at $\overline{\mathsf{p}}$ is

$$\overline{\mathsf{p}} \to \overline{\mathsf{p}}_1 \to \overline{\mathsf{p}}_2 \to \cdots \to \overline{\mathsf{p}}_n$$

where $\overline{\mathsf{p}}_1 = trunc(\overline{\mathsf{p}}, n-1)$ (that is, $\overline{\mathsf{p}}_1$ corresponds to $\overline{\mathsf{p}}$ without its last character), $\overline{\mathsf{p}}_2 = trunc(\overline{\mathsf{p}}_1, n-2)$, $\overline{\mathsf{p}}_3 = trunc(\overline{\mathsf{p}}_2, n-3)$, and so on, until we reach $\overline{\mathsf{p}}_n = trunc(\overline{\mathsf{p}}_{n-1}, n-n) = trunc(\overline{\mathsf{p}}_{n-1}, 0) = \epsilon$. $\overline{\mathsf{p}}_n$ corresponds to an empty prefix: it is $*$, which represents any string, the top of our domain. Thus, given any prefix $\overline{\mathsf{p}}$ of length n (which is finite), the ascending chain starting at $\overline{\mathsf{p}}$ has finite length $n+1$.

Least Upper Bound and Greatest Lower Bound

Given two prefixes, their least upper bound $\sqcup_{\overline{\mathcal{PR}}}$ is their longest common prefix. If the two prefixes do not have anything in common, the least upper bound is $*$ (the

prefix is empty). Instead, the greatest lower bound operator is defined by:

$$\sqcap_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}}_1, \overline{\mathsf{p}}_2) = \begin{cases} \overline{\mathsf{p}}_1 & \text{if } \overline{\mathsf{p}}_1 \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}_2 \\ \overline{\mathsf{p}}_2 & \text{if } \overline{\mathsf{p}}_2 \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}_1 \\ \bot_{\overline{\mathcal{PR}}} & \text{otherwise} \end{cases}$$

Lemma 3.6.5. *$\sqcup_{\overline{\mathcal{PR}}}$ is the least upper bound operator.*

Proof. Let $\overline{\mathsf{p}} = \overline{\mathsf{p}}_1 \sqcup_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}_2$ be the least upper bound of $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$. Then we have to prove the two following conditions:

1. $\overline{\mathsf{p}}_1 \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}} \wedge \overline{\mathsf{p}}_2 \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}$ straightforwardly holds, since $\overline{\mathsf{p}}$ is the longest common prefix between $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$ by definition of $\sqcup_{\overline{\mathcal{PR}}}$, so it is a prefix of both. This implies $\overline{\mathsf{p}}_1 \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}} \wedge \overline{\mathsf{p}}_2 \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}$ by definition of $\leq_{\overline{\mathcal{PR}}}$.

2. $\overline{\mathsf{p}} \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}'$ $\forall$ upper bound $\overline{\mathsf{p}}'$ of $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$. By definition of the lattice structure of $\overline{\mathcal{PR}}$, $\overline{\mathsf{p}}'$ has to be a prefix of *both* $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$. Since $\overline{\mathsf{p}}$ is the *longest* common prefix between $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$ by definition of $\sqcup_{\overline{\mathcal{PR}}}$, we know for sure that $\overline{\mathsf{p}}'$ cannot be longer than $\overline{\mathsf{p}}$: $\overline{\mathsf{p}}'$ is then a prefix of $\overline{\mathsf{p}}$, and so we proved $\overline{\mathsf{p}} \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}'$ by definition of $\leq_{\overline{\mathcal{PR}}}$.

□

Lemma 3.6.6. *$\sqcap_{\overline{\mathcal{PR}}}$ is a greatest lower bound operator.*

Proof. Let $\overline{\mathsf{p}} = \overline{\mathsf{p}}_1 \sqcap_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}_2$ be the greatest lower bound of $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$. Then we have to prove the two following conditions:

1. $\overline{\mathsf{p}} \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}_1 \wedge \overline{\mathsf{p}} \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}_2$ comes straightforwardly from the definition of $\sqcap_{\overline{\mathcal{PR}}}$.

2. $\overline{\mathsf{p}}' \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}$ $\forall$ lower bound $\overline{\mathsf{p}}'$ of $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$. If $\overline{\mathsf{p}}' = \bot_{\overline{\mathcal{PR}}}$, by definition of $\leq_{\overline{\mathcal{PR}}}$ it surely holds that $\overline{\mathsf{p}}' \leq_{\overline{\mathcal{CI}}} \overline{\mathsf{p}}$. Otherwise, it must hold that *both* $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$ are prefixes of $\overline{\mathsf{p}}'$ by definition of the lattice structure of $\overline{\mathcal{PR}}$. Then, it holds that $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$ are one the prefix of the other one (since they are both prefixes of the same string $\overline{\mathsf{p}}'$). Suppose that $\overline{\mathsf{p}}_1$ is the prefix of $\overline{\mathsf{p}}_2$ (the other case is symmetrical): then, $\overline{\mathsf{p}}_2 \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}_1$ by definition of $\leq_{\overline{\mathcal{PR}}}$. If this is the case, by definition of $\sqcap_{\overline{\mathcal{PR}}}$, we also know that $\overline{\mathsf{p}} = \overline{\mathsf{p}}_2$. Since $\overline{\mathsf{p}}' \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}_2$ by hypothesis and $\overline{\mathsf{p}} = \overline{\mathsf{p}}_2$, we get that $\overline{\mathsf{p}}' \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}$ by definition of $\leq_{\overline{\mathcal{PR}}}$.

□

Lemma 3.6.7. *The abstract domain $\overline{\mathcal{PR}}$ is a lattice.*

Proof. The order based on prefixes is a partial order. Informally: (i) a string is always a prefix of itself (reflexivity); (ii) if a string is prefix of another one and viceversa, then the two strings have to be the same string (antisymmetry); (iii) if a string s_1 is prefix of another string s_2 and s_2 is prefix of another string s_3, then s_1 is also a prefix of s_3 (transitivity).

The fact that $\sqcup_{\overline{\mathcal{PR}}}$ and $\sqcap_{\overline{\mathcal{PR}}}$ are the least upper bound and the greatest lower bound operators is proved by the two previous Lemmas.

□

Note that $\overline{\mathcal{PR}}$ is a simple lattice, and not a *complete* lattice. In fact, in a complete lattice both the lub and glb of *any* set (finite and infinite) must exist. However, we cannot define the greatest lower bound for an infinite set of elements, because this domain may have infinite chains of finite prefixes, whose limit can only be an infinite trace and not the bottom element. The infinite trace is not in the domain, so the domain (as it is defined) cannot be complete since it does not contain the limit of infinite descending chains. However, this does not cause problems, because what really matters (for the convergence of the analysis) are the ascending chains, which, in this domain, are finite. In other words, we want to be sure that there is a limit when we deal with chains of increasing elements and this is guaranteed because, as we proved before, given any prefix $\overline{\mathsf{p}}$ of length n (which is finite), the ascending chain starting at $\overline{\mathsf{p}}$ has finite length $n+1$.

Abstraction and Concretization Functions

The concretization function $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}})$ is defined as follows:

$$\begin{cases} \emptyset & \text{if } \overline{\mathsf{p}} = \bot_{\overline{\mathcal{PR}}} \\ \{s : s \in \mathsf{K}^* \wedge len(s) \geq len(\overline{\mathsf{p}}) \wedge \ \forall i \in [0, len(\overline{\mathsf{p}}) - 1] : s[i] = \overline{\mathsf{p}}[i]\} & \text{otherwise} \end{cases}$$

The abstract value $\overline{\mathsf{p}}$ maps to the set of the strings which begin with the sequence of characters represented by $\overline{\mathsf{p}}$.

Theorem 3.6.8. *Let the abstraction function $\alpha_{\overline{\mathcal{PR}}}$ be defined by $\alpha_{\overline{\mathcal{PR}}} = \lambda \mathsf{Y}.\sqcap_{\overline{\mathcal{PR}}}\{\overline{\mathsf{p}} : \mathsf{Y} \subseteq \gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}})\}$.*

Then $\langle \wp(\mathsf{S}), \subseteq \rangle \underset{\alpha_{\overline{\mathcal{PR}}}}{\overset{\gamma_{\overline{\mathcal{PR}}}}{\leftrightarrows}} \langle \overline{\mathcal{PR}}, \leq_{\overline{\mathcal{PR}}} \rangle$.

Proof. By Theorem 2.8.3 we only need to prove that $\gamma_{\overline{\mathcal{PR}}}$ is a complete meet morphism. Formally, we have to prove that $\gamma_{\overline{\mathcal{PR}}}(\bigsqcap_{\substack{\mathcal{PR} \\ (\overline{\mathsf{p}}) \in \overline{\mathsf{X}}}} \overline{\mathsf{p}}) = \bigcap_{\overline{\mathsf{p}} \in \overline{\mathsf{X}}} \gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}})$.

By definition of $\bigsqcap_{\overline{\mathcal{PR}}}$, we can have only the two following cases:

1. $\exists \overline{\mathsf{p}}' \in \overline{\mathsf{X}} : \forall \overline{\mathsf{p}} \in \overline{\mathsf{X}} : \overline{\mathsf{p}}' \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}}$. Then we have the following inference chain:

$$\begin{aligned} & \gamma_{\overline{\mathcal{PR}}}(\bigsqcap_{\substack{\mathcal{PR} \\ (\overline{\mathsf{p}}) \in \overline{\mathsf{X}}}} \overline{\mathsf{p}}) \\ & \quad \text{by Definition of } \sqcap_{\overline{\mathcal{PR}}} \\ & = \gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}'}) \\ & \quad \text{by Definition of } \gamma_{\overline{\mathcal{PR}}} \\ & = \{s : s \in \mathsf{K}^* \wedge len(s) \geq len(\overline{\mathsf{p}}') \wedge \ \forall i \in [0, len(\overline{\mathsf{p}}') - 1] : s[i] = \overline{\mathsf{p}}'[i]\} \\ & \quad \text{by Definition of } \leq_{\overline{\mathcal{PR}}} \text{ since } \forall \overline{\mathsf{p}} \in \overline{\mathsf{X}} : \overline{\mathsf{p}}' \leq_{\overline{\mathcal{PR}}} \overline{\mathsf{p}} \\ & = \bigcap_{\overline{\mathsf{p}} \in \overline{\mathsf{X}}} \{s : s \in \mathsf{K}^* \wedge len(s) \geq len(\overline{\mathsf{p}}) \wedge \ \forall i \in [0, len(\overline{\mathsf{p}}) - 1] : s[i] = \overline{\mathsf{p}}[i]\} \\ & \quad \text{by Definition of } \gamma_{\overline{\mathcal{PR}}} \\ & = \bigcap_{\overline{\mathsf{p}} \in \overline{\mathsf{X}}} \gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}}) \end{aligned}$$

Table 3.5: The abstract semantics of $\overline{\mathcal{PR}}$

$$\overline{\mathbb{S}_{\mathcal{PR}}}[\![\texttt{new String(str)}]\!]() = \mathsf{str}$$
$$\overline{\mathbb{S}_{\mathcal{PR}}}[\![\texttt{concat}]\!](\overline{\mathsf{p}}_1, \overline{\mathsf{p}}_2) = \overline{\mathsf{p}}_1$$
$$\overline{\mathbb{S}_{\mathcal{PR}}}[\![\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}]\!](\overline{\mathsf{p}}) = \begin{cases} \overline{\mathsf{p}}[\mathsf{b} \cdots \mathsf{e}-1] & \text{if } \mathsf{e} \leq \mathit{len}(\overline{\mathsf{p}}) \\ \overline{\mathsf{p}}[\mathsf{b} \cdots \mathit{len}(\overline{\mathsf{p}})-1] & \text{if } \mathsf{e} > \mathit{len}(\overline{\mathsf{p}}) \wedge \mathsf{b} < \mathit{len}(\overline{\mathsf{p}}) \\ \epsilon & \text{otherwise} \end{cases}$$
$$\overline{\mathbb{B}_{\mathcal{PR}}}[\![\texttt{contains}_{\mathsf{c}}]\!](\overline{\mathsf{p}}) = \begin{cases} \mathsf{true} & \text{if } \mathsf{c} \in \mathit{char}(\overline{\mathsf{p}}) \\ \top_{\mathsf{B}} & \text{otherwise} \end{cases}$$

2. otherwise, $\gamma_{\overline{\mathcal{PR}}}(\bigsqcap_{\mathcal{PR}}\limits_{(\overline{\mathsf{p}})\in\overline{\mathsf{X}}} \overline{\mathsf{p}}) = \gamma_{\overline{\mathcal{PR}}}(\bot_{\overline{\mathcal{PR}}}) = \emptyset$. Then $\bigcap_{\overline{\mathsf{p}}\in\overline{\mathsf{X}}} \gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{a}}) = \emptyset$, since there is no concretized strings in common among abstract states that represent different prefixes.

□

In a similar way to $\overline{\mathcal{PR}}$, we can also track information about the *suffix* of a string. We introduce another abstract domain, $\overline{\mathcal{SU}}$, where a string is approximated by the *end* of a certain sequence of characters, while we do not track anything about the string *before* such suffix. The notation and all the operators of this domain are dual to those of $\overline{\mathcal{PR}}$ domain.

The domain definition is: $\overline{\mathcal{SU}} = \mathsf{K}^* \cup \bot_{\overline{\mathcal{SU}}}$. As for the partial order, $\overline{\mathsf{s}}_1 \leq_{\overline{\mathcal{SU}}} \overline{\mathsf{s}}_2$ if $\overline{\mathsf{s}}_2$ is a suffix of $\overline{\mathsf{s}}_1$ or if $\overline{\mathsf{s}}_1$ is the bottom value of the domain, $\bot_{\overline{\mathcal{SU}}}$. The top element $\top_{\overline{\mathcal{SU}}}$ is $*$, while the bottom value is the special element $\bot_{\overline{\mathcal{SU}}}$ (for the same reasons explained for $\bot_{\overline{\mathcal{PR}}}$). The least upper bound operator $\sqcup_{\overline{\mathcal{SU}}}$, dually to $\sqcup_{\overline{\mathcal{PR}}}$, is defined as the longest common suffix between the two suffixes in input. As for the greatest lower bound, if the two suffixes are not comparable with respect to the order $\leq_{\overline{\mathcal{SU}}}$ (e.g., $*a$ and $*b$), then the string sets they represent have nothing in common and their glb is thus $\bot_{\overline{\mathcal{SU}}}$. If they are comparable, the smaller element between the two is the greatest lower bound. $\overline{\mathcal{SU}}$ is a domain with infinite height, just like $\overline{\mathcal{PR}}$. In fact, given any suffix, we can always add a character at its beginning, thus obtaining a new suffix, longer (therefore smaller, according to the order $\leq_{\overline{\mathcal{SU}}}$) than the initial one. As it happened with $\overline{\mathcal{PR}}$, though, this domain respects the ACC condition, and it does not need a widening operator. The concretization function maps an abstract value $\overline{\mathsf{s}}$ to the set of the strings which end with the sequence of characters represented by $\overline{\mathsf{s}}$.

All the proofs for this domain are symmetrical to those presented for $\overline{\mathcal{PR}}$.

Table 3.6: The abstract semantics of $\overline{\mathcal{SU}}$

$$\overline{\mathbb{S}_{\mathcal{SU}}}[\![\texttt{new String(str)}]\!]() = \mathsf{str}$$
$$\overline{\mathbb{S}_{\mathcal{SU}}}[\![\texttt{concat}]\!](\bar{\mathsf{s}}_1, \bar{\mathsf{s}}_2) = \bar{\mathsf{s}}_2$$
$$\overline{\mathbb{S}_{\mathcal{SU}}}[\![\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}]\!](\bar{\mathsf{s}}) = \epsilon$$
$$\overline{\mathbb{B}_{\mathcal{SU}}}[\![\texttt{contains}_{\mathsf{c}}]\!](\bar{\mathsf{s}}) = \begin{cases} \mathsf{true} & \text{if } \mathsf{c} \in \mathit{char}(\bar{\mathsf{s}}) \\ \top_{\mathsf{B}} & \text{otherwise} \end{cases}$$

Semantics

Tables 3.5 and 3.6 define the abstract semantics on $\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$, respectively. Let us explain in detail the semantics of each operator:

- When we evaluate a constant string value (`new String(str)`), the most precise suffix and prefix are the string itself.
- When we concatenate two strings, we create a new string which starts with the first one and ends with the second one. Then, we consider as prefix and suffix of the resulting string the abstract value of the left and right operand, respectively.
- The semantics of $\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}$ is $\top_{\overline{\mathcal{SU}}}$ in $\overline{\mathcal{SU}}$, since we do not know how many characters there are *before* the suffix (b and e are relative to the beginning of the string). With $\overline{\mathcal{PR}}$, instead, we *do* know how the string begins, so we can be more precise if b (and eventually e) are smaller than the length of the prefix we have. We have to distinguish three different cases: (i) if $\mathsf{e} \leq \mathit{len}(\bar{\mathsf{p}})$, the substring is completely included in the known prefix; (ii) if $\mathsf{e} > \mathit{len}(\bar{\mathsf{p}})$ but $\mathsf{b} < \mathit{len}(\bar{\mathsf{p}})$, only the first part of the substring is in the prefix; (iii) if $\mathsf{b} \geq \mathit{len}(\bar{\mathsf{p}})$, the substring is completely further the prefix and we return $\top_{\overline{\mathcal{PR}}}$.
- The semantics of $\texttt{contains}_{\mathsf{c}}$ returns true iff c is contained in the prefix or in the suffix, and $\top_{\mathsf{B}}$ otherwise, since we have no information at all about which characters are after the prefix or before the suffix.

We now prove the soundness of the abstract operations defined above.

Theorem 3.6.9 (Soundness of the abstract semantics)**.** *$\overline{\mathbb{S}_{\mathcal{PR}}}$ and $\overline{\mathbb{B}_{\mathcal{PR}}}$ are a sound overapproximation of $\mathbb{S}$ and $\mathbb{B}$, respectively. Formally, $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathbb{S}_{\mathcal{PR}}}[\![\mathsf{s}]\!](\bar{\mathsf{p}})) \supseteq \{\mathbb{S}[\![\mathsf{s}]\!](\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{PR}}}(\bar{\mathsf{p}})\}$ and $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathbb{B}_{\mathcal{PR}}}[\![\mathsf{s}]\!](\bar{\mathsf{p}})) \geq_{\mathsf{B}} \{\mathbb{B}[\![\mathsf{s}]\!](\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{PR}}}(\bar{\mathsf{p}})\}$.*

Proof. We prove the soundness separately for each operator. We only prove the soundness for the $\overline{\mathcal{PR}}$ domain: the proof for $\overline{\mathcal{SU}}$ are simply their mirror image.

- $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathbb{S}_{\mathcal{PR}}}[\![\texttt{new String(str)}]\!]()) \supseteq \{\mathbb{S}[\![\texttt{new String(str)}]\!]()\}$ follows immediately from the definition of $\overline{\mathbb{S}_{\mathcal{PR}}}[\![\texttt{new String(str)}]\!]()$ and of $\gamma_{\overline{\mathcal{PR}}}$.

- Consider the binary operator `concat`. Let $\overline{\mathsf{p}}_1$ and $\overline{\mathsf{p}}_2$ be two prefixes. We have to prove that $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathbb{S}_{\mathcal{PR}}[\![\texttt{concat}]\!]}(\overline{\mathsf{p}}_1, \overline{\mathsf{p}}_2)) \supseteq \{\mathbb{S}[\![\texttt{concat}]\!](\mathsf{c}_1, \mathsf{c}_2) : \mathsf{c}_1 \in \gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}}_1) \wedge \mathsf{c}_2 \in \gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}}_2)\}$. A generic element $\mathsf{c}_1 \in \gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}}_1)$ is a string which starts with the prefix $\overline{\mathsf{p}}_1$; the same goes for $\mathsf{c}_2 \in \gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}}_2)$. The concatenation of c_1 and c_2 produces a string which starts with $\overline{\mathsf{p}}_1$ and afterwards contains $\overline{\mathsf{p}}_2$ (in an unknown position) by definition of $\mathbb{S}$. Then, this string belongs to $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathbb{S}_{\mathcal{PR}}[\![\texttt{concat}]\!]}(\overline{\mathsf{p}}_1, \overline{\mathsf{p}}_2))$, since $\overline{\mathbb{S}_{\mathcal{PR}}[\![\texttt{concat}]\!]}(\overline{\mathsf{p}}_1, \overline{\mathsf{p}}_2) = \overline{\mathsf{p}}_1$ by definition of $\overline{\mathbb{S}_{\mathcal{PR}}}$, and $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}}_1)$ returns all the strings which start with $\overline{\mathsf{p}}_1$.

- Consider the unary operator $\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}$ and let $\overline{\mathsf{p}}$ be an abstract state. A generic string $\mathsf{c} \in \gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}})$ is a string which starts with $\overline{\mathsf{p}}$ by definition of $\gamma_{\overline{\mathcal{PR}}}$. We may have only the following three cases:

  - if $\mathsf{e} \leq \mathit{len}(\overline{\mathsf{p}})$, the substring of c from the b-th character to the e-th character is completely known (since the prefix $\overline{\mathsf{p}}$ is longer than e characters) and the result of the concrete semantics applied to c is the substring from the bth to the $\mathsf{e}-1$th character. $\overline{\mathbb{S}_{\mathcal{PR}}[\![\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}]\!]}(\overline{\mathsf{p}})$ returns the prefix composed by the substring from $\overline{\mathsf{p}}[\mathsf{b}]$ to $\overline{\mathsf{p}}[\mathsf{e}-1]$ by definition of $\overline{\mathbb{S}_{\mathcal{PR}}}$. The concretization of this result returns all the strings starting with the substring from $\overline{\mathsf{p}}[\mathsf{b}]$ to $\overline{\mathsf{p}}[\mathsf{e}-1]$ by definition of $\gamma_{\overline{\mathcal{PR}}}$, thus it contains also such substring that is the result of the concrete semantics.
  - if $\mathsf{e} > \mathit{len}(\overline{\mathsf{p}}) \wedge \mathsf{b} < \mathit{len}(\overline{\mathsf{p}})$, since c starts with $\overline{\mathsf{p}}$ by definition of $\gamma_{\overline{\mathcal{PR}}}$, we surely know that the substring of c from the b-th character to the e-th character starts with the characters from $\overline{\mathsf{p}}[\mathsf{b}]$ to $\overline{\mathsf{p}}[\mathit{len}(\overline{\mathsf{p}})-1]$ by definition of $\mathbb{S}$. $\overline{\mathbb{S}_{\mathcal{PR}}[\![\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}]\!]}(\overline{\mathsf{p}})$ returns the prefix made by the characters from $\overline{\mathsf{p}}[\mathsf{b}]$ to $\overline{\mathsf{p}}[\mathit{len}(\overline{\mathsf{p}})-1]$, thus representing all strings starting with such characters by definition of $\gamma_{\overline{\mathcal{PR}}}$. Therefore, it surely contains also the resulting substring of c.
  - otherwise, $\overline{\mathbb{S}_{\mathcal{PR}}[\![\texttt{substring}_{\mathsf{b}}^{\mathsf{e}}]\!]}(\overline{\mathsf{p}})$ returns ϵ, that is, the top element of $\overline{\mathcal{PR}}$, that trivially overapproximates any possible result of the concrete semantics.

- Consider the unary operator $\texttt{contains}_{\mathsf{c}}$ and let $\overline{\mathsf{p}}$ be an abstract prefix. Regarding the character c, we have two possible cases:

  - If $\mathsf{c} \in \mathit{char}(\overline{\mathsf{p}})$, all the strings belonging to $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathsf{p}})$ contain at least one occurrence of c, because they start with the prefix $\overline{\mathsf{p}}$ by definition of $\gamma_{\overline{\mathcal{PR}}}$, and such prefix contains the character c. Then, the concrete semantics returns always `true`, and the abstract semantics returns the same result.
  - Otherwise, $\mathsf{c} \notin \mathit{char}(\overline{\mathsf{p}})$, and the abstract semantics returns $\top_{\mathsf{B}}$, that trivially overapproximates any possible result of the concrete semantics.

□

#I	Var	$\overline{\mathcal{PR}}$	$\overline{\mathcal{SU}}$
1	query	$\overline{s_1}$	$\overline{s_1}$
3	l	ϵ	ϵ
4	query	$\overline{s_1}$	$\overline{s_3}$
5	query	$\overline{s_1}$	" "
6	per	$\overline{s_4}$	$\overline{s_4}$
8	query	$\overline{s_1}$	$\overline{s_6}$

(a) Analysis of prog1

#I	Var	$\overline{\mathcal{PR}}$	$\overline{\mathcal{SU}}$
1	x	a	a
3	x	0	1
4	x	$\top$	$\top$

(b) Analysis of prog2

Figure 3.4: The results of $\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$

Case studies

The results of the analysis using the prefix and suffix domains on the two case studies are reported by Figure 3.4.

For prog1, at line 1, query contains the whole string s_1 as both prefix and suffix. l is an input, so its prefix and suffix are both empty. After the concatenation at line 4, the prefix will be equal to s_1, the suffix to s_3 because we keep the prefix of the first string being concatenated and the suffix of the last one. Since the value of l is unknown, we must compute the least upper bound between the abstract values of query after lines 1 and 4. Then, at line 5, the prefix is s_1 and the suffix is a space character (the longest common suffix between s_1 and s_3). The variable per is associated at line 6 to s_4 for both the prefix and the suffix. At the end of the analysis, from the concatenation of line 8 we get that the prefix of query is string s_1 and its suffix is s_6, although we lose information about what there is in the middle.

For prog2, before entering the loop we know that the prefix and suffix of x are both an 'a' character. After the first iteration of the loop we get that the prefix of x is '0' and its suffix is '1'. The least upper bound of such state with the state *before* the loop (prefix and suffix are both an 'a' character), unfortunately goes to $\top$ (the longest common prefixes and suffixes are empty). Then, we reached convergence after just one iteration (since the least upper bound of any element with the $\top$ value returns always the $\top$ value), but we lost all the information.

3.6.3 Bricks

The domains already introduced do not track precise information about the order of characters. In fact, in $\overline{\mathcal{CI}}$ (Section 3.6.1) each character of the abstract representation was completely unrelated with regard to the others, while in the $\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$ domains (Section 3.6.2) we also considered order, but limited at the beginning (or at the end) of the string. Instead, the abstract domain we will define in this section will consider both inclusion and order among characters, but not limited to the beginning or the end of the string. Therefore, the information tracked by this

domain could be adopted to prove more sophisticated properties than the previous domains (e.g., the well-formedness of SQL queries). Obviously, this comes at a price: this abstract domain (called $\overline{\mathcal{BR}}$) is more expressive than $\overline{\mathcal{CI}}$, $\overline{\mathcal{PR}}$, and $\overline{\mathcal{SU}}$. $\overline{\mathcal{BR}}$ is based on the idea of identifying a string through a regular expression, but full regular expressions are too much complex for our purposes, and thus we will approximate them.

In $\overline{\mathcal{BR}}$, a string is approximated by a sequence of *bricks*. A single brick is defined by

$$\overline{\mathcal{B}} = [\mathsf{S}_1]^{\mathsf{min},\mathsf{max}} : \mathsf{S}_1 \in \wp(\mathsf{S})$$

where min and max are two integer positive values or $+\infty$ [4], S is the set of all strings, and S_1 is a generic element of its powerset. A brick represents all the strings which can be built through concatenation of the given strings (a subset of S), taken between min and max times altogether. For instance, $[\{\text{“}mo\text{”}, \text{“}de\text{”}\}]^{1,2}$ corresponds to $\{mo, de, momo, dede, mode, demo\}$

Elements in $\overline{\mathcal{BR}}$ represent strings as ordered lists of bricks. For instance, $[\{\text{“}straw\text{”}\}]^{0,1}$ $[\{\text{“}berry\text{”}\}]^{1,1} = \{berry, strawberry\}$ since $[\{\text{“}straw\text{”}\}]^{0,1}$ concretizes to $\{\epsilon, \text{“}straw\text{”}\}$ and $[\{\text{“}berry\text{”}\}]^{1,1}$ to $\{\text{“}berry\text{”}\}$. Formally, concatenation between bricks is defined as:

$$\overline{\mathsf{B}}_1\overline{\mathsf{B}}_2 = \{\alpha\beta : \alpha \in strings(\overline{\mathsf{B}}_1) \wedge \beta \in strings(\overline{\mathsf{B}}_2)\}$$

where $strings(\overline{\mathsf{B}})$ represents all the strings which can be built from the single brick $\overline{\mathsf{B}}$.

Since a particular set of strings could be represented by more than one combination of bricks (for example, *abc* is represented by $[\{abc\}]^{1,1}$ but also by $[\{a\}]^{1,1}[\{b\}]^{1,1}$ $[\{c\}]^{1,1}$, etc...), we adopted a normalized form. The normalization algorithm is based on five normalizing rules. After presenting the concretization function, we will prove the soundness of these normalization rules. The normal representation can be seen as the fixpoint of the application of the five rules to a given representation. We call $\overline{normBricks}(\overline{\mathsf{L}})$ the function which, given a list of bricks $\overline{\mathsf{L}}$, returns its normalized version. The five normalization rules are as follows:

Rule 1 remove unnecessary bricks, i.e., bricks of the form: $[\emptyset]^{0,0}$, since they represent only the empty string, which is the neutral element of the concatenation operation.

Rule 2 merge successive bricks with the same indices, $\mathsf{min} = 1$ and $\mathsf{max} = 1$, in a new single brick where the indices remain the same ($\mathsf{min} = \mathsf{max} = 1$), and the strings set is the concatenation the two original strings sets (i.e., each string is made by the concatenation of one string from the first set and one from the second set, in this order). For example, the two bricks $\overline{\mathsf{B}_0} = [\{a, cd\}]^{(1,1)}$ and $\overline{\mathsf{B}_1} = [\{b, ef\}]^{(1,1)}$ become, after the application of the second rule, the new single brick $\overline{\mathsf{B}'} = [\{ab, aef, cdb, cdef\}]^{(1,1)}$.

[4] The order relationship on integers is enriched to consider also the value of $+\infty$.

Rule 3 transform a brick in which the number of applications is constant ($\mathsf{min} = \mathsf{max}$) into one in which the indices are 1 ($\mathsf{min} = \mathsf{max} = 1$). Formally, a brick of the form $\overline{\mathsf{B_0}} = [\mathsf{S_0}]^{(m,m)}$ becomes the brick $\overline{\mathsf{B}}' = [\mathsf{S_0}^m]^{(1,1)}$, where $\mathsf{S_0}^m$ represents the concatenation of $\mathsf{S_0}$ with itself for m times. For example, $\overline{\mathsf{B}} = [\{a, b\}]^{(2,2)}$ becomes $\overline{\mathsf{B}'} = [\{aa, ab, ba, bb\}]^{(1,1)}$.

Rule 4 merge two successive bricks in which the set of strings is the same ($\mathsf{S_i} = \mathsf{S_{i+1}}$) into a single one modifying the indices. Formally, the bricks $\overline{\mathsf{B_i}} = [\mathsf{S_i}]^{(m_1,M_1)}$ and $\overline{\mathsf{B_{i+1}}} = [\mathsf{S_i}]^{(m_2,M_2)}$ become the new single bricks $\overline{\mathsf{B}} = [\mathsf{S_i}]^{(m_1+m_2,M_1+M_2)}$.

Rule 5 break a single brick with $\mathsf{min} \geq 1 \wedge \mathsf{max} \neq \mathsf{min}$ into two simpler bricks. More precisely, a brick of the form $\overline{\mathsf{B_i}} = [\mathsf{S_i}]^{(min,max)}$, where $\mathsf{min} \geq 1 \wedge \mathsf{max} \neq \mathsf{min}$, becomes the concatenation of $\overline{\mathsf{B_{i1}}} = [\mathsf{S_i}^{min}]^{(1,1)}$ and $\overline{\mathsf{B_{i2}}} = [\mathsf{S_i}]^{(0,max-min)}$. A simple example is the following one: the brick $\overline{\mathsf{B}} = [\{a\}]^{(2,5)}$ becomes the concatenation of the two bricks $\overline{\mathsf{B_1}} = [\{aa\}]^{(1,1)}$ and $\overline{\mathsf{B_2}} = [\{a\}]^{(0,3)}$.

Let us present an example of the normalization process. Consider the bricks list $[\{a\}]^{(1,1)}[\{a, b\}]^{(2,3)}[\{a, b\}]^{(0,1)}$. First, we can apply the fourth rule to the second and third brick, merging them because their strings set is the same. We obtain the new bricks list $[\{a\}]^{(1,1)}[\{a, b\}]^{(2,4)}$. Now we can apply the fifth rule to the second brick ($[\{a, b\}]^{(2,4)}$), which gets split into the concatenation of two bricks: $[\{aa, ab, ba, bb\}]^{(1,1)}$ and $[\{a, b\}]^{(0,2)}$. The resulting bricks list is then: $[\{a\}]^{(1,1)}$ $[\{aa, ab, ba, bb\}]^{(1,1)}$ $[\{a, b\}]^{(0,2)}$. Finally, we can apply the second rule to the first two bricks, merging them because of their indices range $(1, 1)$. The final list of bricks is then: $[\{aaa, aab, aba, abb\}]^{(1,1)}[\{a, b\}]^{(0,2)}$. We cannot apply any more rules to such representation, therefore we have reached a normal state.

Note that, in a normalized element of $\overline{\mathcal{BR}}$:

- there cannot be two successive bricks with both $\mathsf{min} = \mathsf{max} = 1$. In fact, they would be merged into one single brick by the second rule;
- there cannot be a brick with $\mathsf{min} \geq 1 \wedge \mathsf{max} \geq \mathsf{min}$, since it would be simplified by the third (if $\mathsf{min} = \mathsf{max}$) or fifth (if $\mathsf{max} > \mathsf{min}$) rule;
- there cannot be bricks with indices $\mathsf{min} = \mathsf{max} = 0$, since they would be removed by the first rule.

Thus, every brick of the normalized list will be in the form $[\mathsf{T}]^{1,1}$ or $[\mathsf{T}]^{0,\mathsf{max}>0}$ (where T is a set of strings).

The abstract domain of bricks is defined by

$$\overline{\mathcal{BR}} = \overline{normBricks}(\overline{\mathcal{B}}^*)$$

that is, the set of all finite normalized sequences of bricks.

Comparison between Lists of Bricks

In the definition of lattice and semantics operators, we will often have to deal with various lists of bricks of different length. However, it is usually convenient to deal with lists of the same size to define effective operators. When dealing with two abstract elements, this means to augment the shorter list with some empty bricks ($E = [\emptyset]^{(0,0)}$). In fact, empty bricks represent the empty string, and adding empty bricks in *any* position of a bricks list will not change the set of strings represented by such bricks list.

A crucial question is *where* to insert the empty bricks in the shorter list. Let $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$ be two lists of bricks, and let $\overline{\mathsf{L}}_1$ be the shortest one. Let n_1 be the number of bricks of $\overline{\mathsf{L}}_1$, n_2 the number of bricks of $\overline{\mathsf{L}}_2$, and n be their difference ($n = n_2 - n_1$). Then, we have to add n empty bricks to $\overline{\mathsf{L}}_1$. The simplest solution would be to insert *all* n bricks at the beginning (or end) of $\overline{\mathsf{L}}_1$. However, this method often induces loss of precision, because it does not consider possible "similarities" between bricks from the two lists. Hence, we choose to adopt a different and more precise approach. The idea is that, for each brick of the shorter list, we check if the same brick appears in the other list. If so, we modify the shorter list by adding empty bricks such that the two equal bricks will appear in the same position in the two lists. If no pair of equal bricks is found, the algorithm works in a way that all n empty bricks are added at the beginning of the shorter list.

More formally, Algorithm 1 defines the procedure used to pad the shorter list with empty bricks. The purpose of such algorithm is to build a new list $\overline{\mathsf{L}}_{new}$ which has the same length of $\overline{\mathsf{L}}_2$ (assuming it is the longest one) and contains all bricks of $\overline{\mathsf{L}}_1$ plus some empty bricks E, trying to maximize the positional correspondences of equal bricks in $\overline{\mathsf{L}}_{new}$ and $\overline{\mathsf{L}}_2$. To do this, we process each brick b of $\overline{\mathsf{L}}_2$ (`for` loop at line 7) and, in the same position of $\overline{\mathsf{L}}_{new}$ we put:

- an empty brick E if $\overline{\mathsf{L}}_1$ is empty (i.e., we have already inserted all its bricks in $\overline{\mathsf{L}}_{new}$) or if b and the first brick of $\overline{\mathsf{L}}_1$ are different (lines 11-13);

- b itself, if the first brick of $\overline{\mathsf{L}}_1$ is equal to b. In this case, we also remove the first brick from $\overline{\mathsf{L}}_1$, to avoid inserting it multiple times in the new list. (lines 14-16)

When the empty bricks have all been added (i.e., $emptyBricksAdded \geq n$), we proceed to insert in $\overline{\mathsf{L}}_{new}$ all remaining bricks in $\overline{\mathsf{L}}_1$, one at a time (lines 8-10).

This padding is particularly useful in order to maximize the number of bricks in the two lists that are equals and at the same position. For instance, consider the case $\overline{\mathsf{L}}_1 = [b_0; b_1; b_2]$ and $\overline{\mathsf{L}}_2 = [b_3; b_0; b_1; b_4; b_5]$. The result of the padding is $\overline{\mathsf{L}}_{new} = [E; b_0; b_1; E; b_2]$. We managed to put b_0 and b_1 in the same position as they appear in $\overline{\mathsf{L}}_2$. Thanks to this feature, the lattice and semantic operator will be in position to obtain precise results traversing the list of bricks only once.

Algorithm 1 Algorithm for making two lists of bricks of the same size, by padding the shorter one with empty bricks, where $removeHead(L)$ is a helper function which removes the first value of the list L in input and $L.add(v)$ is a function which adds the value v at the end of the list L, and E represents the empty brick

```
1:  function padList(L̄1, L̄2)
2:      n1 ← length(L̄1)
3:      n2 ← length(L̄2)
4:      n ← n2 − n1
5:      L̄new ← List.empty
6:      emptyBricksAdded ← 0
7:      for i = 0 → n2 − 1 do
8:          if emptyBricksAdded ≥ n then
9:              L̄new ← L̄new.add(L̄1[0])
10:             removeHead(L̄1)
11:         else if empty(L̄1) ∨ L̄1[0]! = L̄2[i] then
12:             L̄new ← L̄new.add(E)
13:             emptyBricksAdded ← emptyBricksAdded + 1
14:         else
15:             L̄new ← L̄new.add(L̄1[0])
16:             removeHead(L̄1)
17:         end if
18:     end for
19:     return L̄new
20: end function
```

Partial Order

To define an order on *lists* of bricks, we have first to define a partial order on *single* bricks. $\leq_{\overline{\mathcal{B}}}$ is defined as follows:

$$[C_1]^{\mathsf{m}_1,\mathsf{M}_1} \leq_{\overline{\mathcal{B}}} [\overline{\mathsf{C}}_2]^{\mathsf{m}_2,\mathsf{M}_2}$$
$$\Updownarrow$$
$$(\overline{\mathsf{C}}_1 \subseteq \overline{\mathsf{C}}_2 \wedge \mathsf{m}_1 \geq \mathsf{m}_2 \wedge \mathsf{M}_1 \leq \mathsf{M}_2) \vee ([\overline{\mathsf{C}}_2]^{\mathsf{m}_2,\mathsf{M}_2} = \top_{\overline{\mathcal{B}}}) \vee ([\overline{\mathsf{C}}_1]^{\mathsf{m}_1,\mathsf{M}_1} = \bot_{\overline{\mathcal{B}}})$$

where $\top_{\overline{\mathcal{B}}}$ and $\bot_{\overline{\mathcal{B}}}$ are two special bricks, greater and smaller than any other brick, respectively. More precisely, given an alphabet of characters K, we define the top element of single bricks as the brick:

$$\top_{\overline{\mathcal{B}}} = [\mathsf{K}]^{(0,+\infty)}$$

which represents any possible string. Instead, the bottom element is defined by:

$$\bot_{\overline{\mathcal{B}}} = [\emptyset]^{(m,M)\neq(0,0)} \vee ([\overline{\mathsf{S}}]^{(m,M)} \wedge M < m)$$

This definition of bottom tries to capture the concept of computational error: in fact, the two possible definitions are both bricks which do not represent any concrete string. They are invalid bricks and they correspond to $\emptyset$. Note that bricks of the form $[\overline{\mathsf{S}}]^{(0,0)}$ (including the *empty brick* $E = [\emptyset]^{(0,0)}$) are all valid bricks which correspond only to the empty string ϵ. By definition of $\leq_{\overline{\mathcal{B}}}$ we are sure that $\bot_{\overline{\mathcal{B}}}$ is smaller than any other brick of the domain. We may also want to prove that no element of the domain $\overline{\mathcal{B}}$ (different from $\bot_{\overline{\mathcal{B}}}$) is smaller than bottom. To this purpose, let $[\overline{\mathsf{S}_1}]^{(a,b)}$ be a generic brick different from $\bot_{\overline{\mathcal{B}}}$ (then, we know for sure that $a \leq b$). We want to prove that $[\overline{\mathsf{S}_1}]^{(a,b)} \not\leq_{\overline{\mathcal{B}}} \bot_{\overline{\mathcal{B}}}$. Since bottom has two possible values, we consider the two cases separately:

- let $\bot_{\overline{\mathcal{B}}} = [\emptyset]^{(m,M)\neq(0,0)}$. Suppose by contradiction that $[\overline{\mathsf{S}_1}]^{(a,b)} \leq_{\overline{\mathcal{B}}} [\emptyset]^{(m,M)\neq(0,0)}$. By definition of $\leq_{\overline{\mathcal{B}}}$, $\overline{\mathsf{S}_1}$ must be equal to $\emptyset$. Then, at least one of the two indices a, b must be equal to zero (otherwise the brick would be equal to $[\emptyset]^{(m,M)\neq(0,0)}$, that is bottom, from which we supposed it to be different). If $a = 0$ we reach a contradiction, because it holds $a < m$ (since $a = 0$ and $m \neq 0$), while by definition of $\leq_{\overline{\mathcal{B}}}$ it should hold the opposite ($a \geq m$). Instead, if $b = 0$, then also a must be equal to 0 (otherwise it would hold $b < a$ and the brick would be equal to bottom, from which we supposed it to be different). However, we just saw how $a = 0$ brings to a contradiction. Thus, we proved that $[\overline{\mathsf{S}_1}]^{(a,b)} \not\leq_{\overline{\mathcal{B}}} [\emptyset]^{(m,M)\neq(0,0)}$.
- let $\bot_{\overline{\mathcal{B}}} = ([\overline{\mathsf{S}}]^{(c,d)} \wedge d < c)$. Remember that $a \leq b$ by hypothesis. Suppose by contradiction that $[\overline{\mathsf{S}_1}]^{(a,b)} \leq_{\overline{\mathcal{B}}} ([\overline{\mathsf{S}}]^{(c,d)}$. By definition of $\leq_{\overline{\mathcal{B}}}$, then, it must hold that $a \geq c \wedge b \leq d$. Consider $b \leq d$. Since $d < c$, it holds also that $b < c$ (by transitive property). Then, since $a \leq b$ and $b < c$, we obtain (again, by transitive property) that $a < c$. But this is in contradiction with $a \geq c$ which should be true if $[\overline{\mathsf{S}_1}]^{(a,b)} \leq_{\overline{\mathcal{B}}} ([\overline{\mathsf{S}}]^{(c,d)}$. Thus, $[\overline{\mathsf{S}_1}]^{(a,b)} \not\leq_{\overline{\mathcal{B}}} ([\overline{\mathsf{S}}]^{(c,d)}$.

We proved that no element of the domain $\overline{\mathcal{B}}$ (different from bottom) is smaller than bottom.

Now we can define the order relationship on elements of $\overline{\mathcal{BR}}$: given two lists $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$, we augment the shorter list using Algorithm 1 in order to have lists of the same size. Then, we proceed by extracting one brick from each list and comparing the two bricks, until we reach the end of the two lists.

Formally, given two lists $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$, we make them have the same size n by applying Algorithm 1, thus obtaining $\overline{\mathsf{L}}'_1$ and $\overline{\mathsf{L}}'_2$. Then:

$$\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2 \Leftrightarrow (\overline{\mathsf{L}}_2 = \top_{\overline{\mathcal{BR}}}) \vee (\overline{\mathsf{L}}_1 = \bot_{\overline{\mathcal{BR}}}) \vee (\forall \mathsf{i} \in [0, \mathsf{n}-1] : \overline{\mathsf{L}}'_1[\mathsf{i}] \leq_{\overline{\mathcal{B}}} \overline{\mathsf{L}}'_2[\mathsf{i}])$$

Note that, depending on the algorithm chosen to pad the shorter list, sometimes the relationship between two lists of bricks can be missed. As a simple example, consider $\overline{\mathsf{L}}_1 = [[\{a\}]^{(1,1)}; [\{b\}]^{(0,1)}]$ and $\overline{\mathsf{L}}_2 = [[\{a\}]^{(1,1)}]$. The first list represents the set of concrete strings $\{a, ab\}$, while the second list represents the singleton $\{a\}$. With Algorithm 1, the second list is transformed into $\overline{\mathsf{L}}'_2 = [[\{a\}]^{(1,1)}; E]$. Then, it holds that $\overline{\mathsf{L}}'_2 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_1$, which is what we expect, since $\{a\} \subseteq \{a, ab\}$. However, if we used a different algorithm which put all empty bricks at the beginning of the list, we would obtain $\overline{\mathsf{L}}''_2 = [E; [\{a\}]^{(1,1)}]$ and in this case $\overline{\mathsf{L}}''_2 \not\leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_1$: we would not be able to prove that $\overline{\mathsf{L}}_2$ represents a smaller set of concrete elements (i.e., is more precise) than $\overline{\mathsf{L}}_1$. This means that the choice of the algorithm to pad the shorter list is very important, since it has direct repercussions on the precision of the order. The experimental results obtained using Algorithm 1 are satisfactory, but it is still unclear if a better algorithm exists. However, the choice of the specific algorithm *must not* have the effect of inverting the order: given two lists $\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2$, it must not happen that $\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2$ when using a certain algorithm, but $\overline{\mathsf{L}}_2 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_1$ when using another one.

Lemma 3.6.10 ($\leq_{\overline{\mathcal{BR}}}$ is not inverted by the choice of the padding algorithm)**.** *Let $\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2$ be two lists of bricks and let A, B be two padding algorithms. Let $\overline{\mathsf{L}}_1^A, \overline{\mathsf{L}}_1^B$ be the lists obtained by applying such algorithms to $\overline{\mathsf{L}}_1$ (to make its length equal to that of $\overline{\mathsf{L}}_2$). Then, it never happens that*

$$\overline{\mathsf{L}}_1^A \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2 \wedge \overline{\mathsf{L}}_2 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_1^B$$

Proof. Let n_1, n_2 be the length of $\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2$ (respectively), where $n_1 < n_2$. Then, algorithms A, B will insert $n_2 - n_1$ empty bricks E in $\overline{\mathsf{L}}_1$: both $\overline{\mathsf{L}}_1^A, \overline{\mathsf{L}}_1^B$ have length n_2. Assume also that $\overline{\mathsf{L}}_1$ does not contain empty bricks (since they would be removed by the normalization). Let $i < n_2$ be an index such that

$$\overline{\mathsf{L}}_1^A[i] \neq E \wedge \overline{\mathsf{L}}_1^B[i] = E$$

Such index must exist, otherwise all the empty bricks in $\overline{\mathsf{L}}_1^B$ would correspond to empty bricks in $\overline{\mathsf{L}}_1^A$ and thus $\overline{\mathsf{L}}_1^A, \overline{\mathsf{L}}_1^B$ would be exactly the same list of bricks (i.e., the two algorithms would not have produced different results).

Suppose, by contradiction, that $\overline{\mathsf{L}}_1^A \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2 \wedge \overline{\mathsf{L}}_2 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_1^B$ holds. From the hypothesis $\overline{\mathsf{L}}_1^A \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2$ we get that $\overline{\mathsf{L}}_1^A[i] \leq_{\overline{\mathcal{B}}} \overline{\mathsf{L}}_2[i]$. From the hypothesis $\overline{\mathsf{L}}_2 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_1^B$ we get that $\overline{\mathsf{L}}_2[i] \leq_{\overline{\mathcal{B}}} \overline{\mathsf{L}}_1^B[i]$. By transitive property and considering we assumed $\overline{\mathsf{L}}_1^B[i] = E$, we obtain:

$$\overline{\mathsf{L}}_1^A[i] \leq_{\overline{\mathcal{B}}} \overline{\mathsf{L}}_2[i] \leq_{\overline{\mathcal{B}}} \overline{\mathsf{L}}_1^B[i] = E$$

Then, again by transitive property, we have that $\overline{\mathsf{L}}_1^A[i] \leq_{\overline{\mathcal{B}}} E$. Let $\overline{\mathsf{L}}_1^A[i] = [S]^{(m,M)}$. Since $E = [\emptyset]^{(0,0)}$, the order relationship $\leq_{\overline{\mathcal{B}}}$ implies that $S \subseteq \emptyset \wedge m \geq 0 \wedge M \leq 0$:

- the only set included in the empty set is the empty set itself;
- the index M must be equal to 0, because all bricks indices must be greater or equal than 0 but it must also hold that $M \leq 0$;
- since $M = 0$ and $m \geq 0$, also m must be equal to 0 (otherwise we would obtain $m > M$ and this would correspond to an invalid brick).

Combining these observations, we obtain that $\overline{\mathsf{L}}_1^A[i] = [\emptyset]^{(0,0)} = E$, which is not possible because we chose a specific index i such that $\overline{\mathsf{L}}_1^A[i] \neq E$: we reached a contradiction. □

Lemma 3.6.11 ($\leq_{\overline{\mathcal{BR}}}$ is a partial order)**.** *The order $\leq_{\overline{\mathcal{BR}}}$ is a partial order.*

Proof. We refer to [44] for the proofs that $\leq_{\overline{\mathcal{BR}}}$ is reflexive and transitive, and here we prove that it is antisymmetric. Formally, we must prove that, given two lists of bricks $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$ of the same length n (otherwise we add empty bricks inside the shorter one through Algorithm 1, without changing the represented set of strings), it holds:

$$\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2 \wedge \overline{\mathsf{L}}_2 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_1 \Rightarrow \overline{\mathsf{L}}_1 = \overline{\mathsf{L}}_2$$

This trivially holds if one of the two abstract states is $\top_{\overline{\mathcal{BR}}}$ or $\bot_{\overline{\mathcal{BR}}}$ by definition of $\leq_{\overline{\mathcal{BR}}}$. Otherwise, since $\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2$, we know that $\forall \mathsf{i} \in [0, \mathsf{n}-1] : \overline{\mathsf{L}}_1[\mathsf{i}] \leq_{\overline{\mathcal{B}}} \overline{\mathsf{L}}_2[\mathsf{i}]$ by definition of $\leq_{\overline{\mathcal{BR}}}$. But we also know that $\overline{\mathsf{L}}_2 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_1$, and this means that $\forall \mathsf{i} \in [0, \mathsf{n}-1] : \overline{\mathsf{L}}_2[\mathsf{i}] \leq_{\overline{\mathcal{B}}} \overline{\mathsf{L}}_1[\mathsf{i}]$. Consider then a generic pair of bricks $\overline{\mathsf{L}}_1[\mathsf{i}]$ and $\overline{\mathsf{L}}_2[\mathsf{i}]$. Neither of these two bricks can be equal to $\bot_{\overline{\mathcal{B}}}$, since otherwise the abstract state to which it belongs would be equal to $\bot_{\overline{\mathcal{BR}}}$ and we already excluded this case. If one brick is equal to $\top_{\overline{\mathcal{B}}}$, then also the other one must be too, otherwise our hypothesis would not hold. In this case, then, the two bricks are equal. Otherwise (neither brick is top nor bottom), let $\overline{\mathsf{L}}_1[\mathsf{i}] = [C_1]^{\mathsf{m}_1,\mathsf{M}_1}$ and $\overline{\mathsf{L}}_2[\mathsf{i}] = [C_2]^{\mathsf{m}_2,\mathsf{M}_2}$. Since $\overline{\mathsf{L}}_1[\mathsf{i}] \leq_{\overline{\mathcal{B}}} \overline{\mathsf{L}}_2[\mathsf{i}]$, it holds that $(\overline{\mathsf{C}}_1 \subseteq \overline{\mathsf{C}}_2 \wedge \mathsf{m}_1 \geq \mathsf{m}_2 \wedge \mathsf{M}_1 \leq \mathsf{M}_2)$. Also, since $\overline{\mathsf{L}}_2[\mathsf{i}] \leq_{\overline{\mathcal{B}}} \overline{\mathsf{L}}_1[\mathsf{i}]$, it holds that $(\overline{\mathsf{C}}_2 \subseteq \overline{\mathsf{C}}_1 \wedge \mathsf{m}_2 \geq \mathsf{m}_1 \wedge \mathsf{M}_2 \leq \mathsf{M}_1)$. Then: (i) from $\overline{\mathsf{C}}_1 \subseteq \overline{\mathsf{C}}_2$ and $\overline{\mathsf{C}}_2 \subseteq \overline{\mathsf{C}}_1$ it follows $\overline{\mathsf{C}}_1 = \overline{\mathsf{C}}_2$; (ii) from $\mathsf{m}_1 \geq \mathsf{m}_2$ and $\mathsf{m}_2 \geq \mathsf{m}_1$ it follows $\mathsf{m}_1 = \mathsf{m}_2$; (iii) from $\mathsf{M}_1 \leq \mathsf{M}_2$ and $\mathsf{M}_2 \leq \mathsf{M}_1$ it follows $\mathsf{M}_1 = \mathsf{M}_2$. This means that $\overline{\mathsf{L}}_1[\mathsf{i}] = \overline{\mathsf{L}}_2[\mathsf{i}]$, and this is valid for all $i \in [0, \mathsf{n}-1]$. This implies that $\overline{\mathsf{L}}_1 = \overline{\mathsf{L}}_2$. □

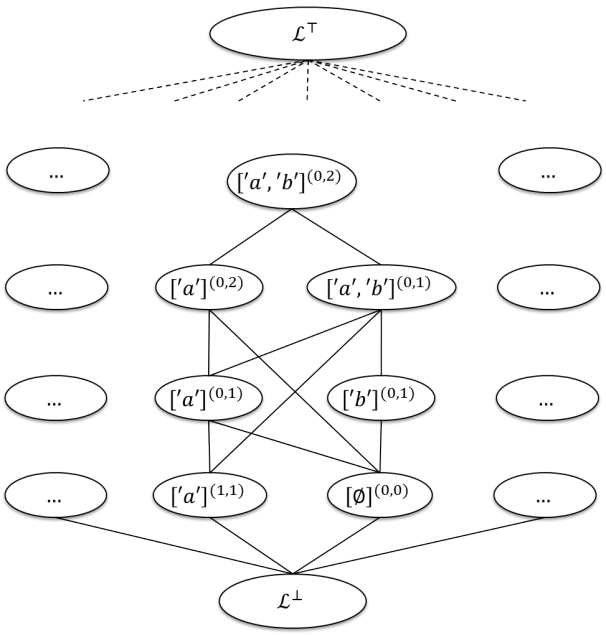


Figure 3.5: The abstract domain $\overline{\mathcal{BR}}$ with $\mathsf{K} = \{a, b\}$

The top element of $\overline{\mathcal{BR}}$ is then a list containing only one brick:

$$\top_{\overline{\mathcal{BR}}} = [\top_{\overline{\mathcal{B}}}]$$

Since $\top_{\overline{\mathcal{B}}}$ represents all the strings, $\top_{\overline{\mathcal{BR}}}$ does too. The bottom element instead is defined as:

$$\bot_{\overline{\mathcal{BR}}} = [\,] \vee ([b_1, \ldots, b_n] : \exists i : b_i = \bot_{\overline{\mathcal{B}}})$$

i.e., $\bot_{\overline{\mathcal{BR}}}$ is an empty list (it does not represent any string at all, not even the empty string) or any list which contains at least one invalid element ($\bot_{\overline{\mathcal{B}}}$). In fact, if the list contains the equivalent of a computational error, then the entire list is considered invalid.

The lattice of $\overline{\mathcal{BR}}$ is depicted in Figure 3.5. For visual clarity we only pictured lists of size one and we considered the alphabet $\mathsf{K} = \{a, b\}$.

Least Upper Bound and Greatest Lower Bound

As we did for the partial order, we define the least upper bound operator on single bricks at first:

$$\bigsqcup_{\overline{\mathcal{B}}}([\overline{\mathsf{S}}_1]^{(\mathsf{m}_1,\mathsf{M}_1)}, [\overline{\mathsf{S}}_2]^{(\mathsf{m}_2,\mathsf{M}_2)}) = [\overline{\mathsf{S}}_1 \cup \overline{\mathsf{S}}_2]^{(m,M)}$$

where $m = \min(\mathsf{m}_1, \mathsf{m}_2)$ and $M = \max(\mathsf{M}_1, \mathsf{M}_2)$. For example, the least upper bound between $[\{a, b\}]^{(1,3)}$ and $[\{a, c\}]^{(0,2)}$ is the brick $[\{a, b, c\}]^{(0,3)}$.

To compute the least upper bound between elements of $\overline{\mathcal{BR}}$ (lists of bricks), we proceed exactly as we did to define the partial order $\leq_{\overline{\mathcal{BR}}}$. Given two lists $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$, we make them have the same size n by using Algorithm 1, thus obtaining $\overline{\mathsf{L}}'_1$ and $\overline{\mathsf{L}}'_2$. Then, $\sqcup_{\overline{\mathcal{BR}}}$ is defined as follows:

$$\bigsqcup_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2) = \bigsqcup_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}'_1, \overline{\mathsf{L}}'_2) = \overline{\mathsf{L}}_R[0]\overline{\mathsf{L}}_R[1] \dots \overline{\mathsf{L}}_R[\mathsf{n}-1]$$

where $\forall \mathsf{i} \in [0, \mathsf{n}-1] : \overline{\mathsf{L}}_R[\mathsf{i}] = \bigsqcup_{\overline{\mathcal{B}}}(\overline{\mathsf{L}}'_1[\mathsf{i}], \overline{\mathsf{L}}'_2[\mathsf{i}])$.

The greatest lower bound operator works very similarly to the least upper bound one. The glb operator on single bricks is defined as follows:

$$\bigsqcap_{\overline{\mathcal{B}}}([\overline{\mathsf{S}}_1]^{(\mathsf{m}_1,\mathsf{M}_1)}, [\overline{\mathsf{S}}_2]^{(\mathsf{m}_2,\mathsf{M}_2)}) = [\overline{\mathsf{S}}_1 \cap \overline{\mathsf{S}}_2]^{(m,M)}$$

where $m = \max(\mathsf{m}_1, \mathsf{m}_2)$ and $M = \min(\mathsf{M}_1, \mathsf{M}_2)$. For example, the greatest lower bound between $[\{a,b\}]^{(1,3)}$ and $[\{a,c\}]^{(0,2)}$ is the brick $[\{a\}]^{(1,2)}$. Note that sometimes the result of the glb is an invalid brick (for example because the *max* index is smaller than the *min* one). To conclude the description of the glb operator, we have to define how it works with lists of bricks (the elements of $\overline{\mathcal{BR}}$). Given two lists $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$, we make them have the same size n by using Algorithm 1, thus obtaining $\overline{\mathsf{L}}'_1$ and $\overline{\mathsf{L}}'_2$. Then $\sqcap_{\overline{\mathcal{BR}}}$ is defined as follows:

$$\bigsqcap_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2) = \bigsqcap_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}'_1, \overline{\mathsf{L}}'_2) = \overline{\mathsf{L}}_R[0]\overline{\mathsf{L}}_R[1] \dots \overline{\mathsf{L}}_R[\mathsf{n}-1]$$

where $\forall \mathsf{i} \in [0, \mathsf{n}-1] : \overline{\mathsf{L}}_R[\mathsf{i}] = \bigsqcap_{\overline{\mathcal{B}}}(\overline{\mathsf{L}}'_1[\mathsf{i}], \overline{\mathsf{L}}'_2[\mathsf{i}])$. If any element of the sequence $\overline{\mathsf{L}}_R$ corresponds to $\bot_{\overline{\mathcal{B}}}$, then the entire resulting list should be set to $\bot_{\overline{\mathcal{BR}}}$.

Lemma 3.6.12. *$\sqcup_{\overline{\mathcal{BR}}}$ is the least upper bound operator.*

Proof. Let $\overline{\mathsf{L}}$ be $\overline{\mathsf{L}} = \overline{\mathsf{L}}_1 \sqcup_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2$. Then we have to prove the following two conditions:

1. $\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}} \wedge \overline{\mathsf{L}}_2 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}$. Let us suppose that $\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2, \overline{\mathsf{L}}$ are lists of the same size (one of them could be padded with empty bricks inside it, but empty bricks do not interfere with order comparisons). Then, for $\overline{\mathsf{L}}_1$ to be smaller than $\overline{\mathsf{L}}$, it must be that each brick of $\overline{\mathsf{L}}_1$ is smaller (in the single brick order) than the corresponding brick of $\overline{\mathsf{L}}$. Let $[\overline{\mathsf{S}}_1]^{(\mathsf{m}_1,\mathsf{M}_1)}$ be the brick of $\overline{\mathsf{L}}_1$ in a generic position i, and $[\overline{\mathsf{S}}_2]^{(\mathsf{m}_2,\mathsf{M}_2)}$ be the brick of $\overline{\mathsf{L}}_2$ in the same position. The brick of $\overline{\mathsf{L}}$ in such position will be, by definition of $\sqcup_{\overline{\mathcal{BR}}}$, $[\overline{\mathsf{S}}_1 \cup \overline{\mathsf{S}}_2]^{(\min(\mathsf{m}_1,\mathsf{m}_2),\max(\mathsf{M}_1,\mathsf{M}_2))}$. The brick of $\overline{\mathsf{L}}_1$ is smaller than the brick of $\overline{\mathsf{L}}$, because $\overline{\mathsf{S}}_1 \subseteq (\overline{\mathsf{S}}_1 \cup \overline{\mathsf{S}}_2) \wedge \mathsf{m}_1 \geq \min(\mathsf{m}_1, \mathsf{m}_2) \wedge \mathsf{M}_1 \leq \max(\mathsf{M}_1, \mathsf{M}_2)$. The same goes for the brick of $\overline{\mathsf{L}}_2$. Thus, $\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}} \wedge \overline{\mathsf{L}}_2 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}$.

2. $\overline{\mathsf{L}} \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}'$ $\forall$ upper bound $\overline{\mathsf{L}}'$ of $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$. As before, suppose that $\overline{\mathsf{L}}$, $\overline{\mathsf{L}}_1$, $\overline{\mathsf{L}}_2$, and $\overline{\mathsf{L}}'$ have all the same size (otherwise we pad them with empty bricks using Algorithm 1). Since $\overline{\mathsf{L}}'$ is an upper bound of $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$, this means that each brick of $\overline{\mathsf{L}}'$ is greater than the corresponding brick of both $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$. Let $[\overline{\mathsf{S}}_1]^{(\mathsf{m}_1,\mathsf{M}_1)}$ be the brick of $\overline{\mathsf{L}}_1$ in a generic position i, and $[\overline{\mathsf{S}}_2]^{(\mathsf{m}_2,\mathsf{M}_2)}$ be the brick of $\overline{\mathsf{L}}_2$ in the same position. Then, the corresponding brick of $\overline{\mathsf{L}}'$ (let it be $[\overline{\mathsf{S}}']^{(\mathsf{m}',\mathsf{M}')}$) must satisfy (to be an upper bound) the following requirements: (i) $\overline{\mathsf{S}}' \supseteq (\overline{\mathsf{S}}_1 \cup \overline{\mathsf{S}}_2)$, (ii) $\mathsf{m}' \leq \min(\mathsf{m}_1, \mathsf{m}_2)$, (iii) $\mathsf{M}' \geq \max(\mathsf{M}_1, \mathsf{M}_2)$. The brick of $\overline{\mathsf{L}}$ in the same position is defined as $[\overline{\mathsf{S}}_1 \cup \overline{\mathsf{S}}_2]^{(\min(\mathsf{m}_1,\mathsf{m}_2),\max(\mathsf{M}_1,\mathsf{M}_2))}$ and it is certainly smaller (or equal) than the brick of $\overline{\mathsf{L}}'$, for definition of $\leq_{\overline{\mathcal{B}}}$. Since this happens for every brick of $\overline{\mathsf{L}}'$ and $\overline{\mathsf{L}}$, it holds that $\overline{\mathsf{L}} \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}'$.

□

Lemma 3.6.13. *$\sqcap_{\overline{\mathcal{BR}}}$ is the greatest lower bound operator.*

Proof. The reasoning is symmetrical to that of the least upper bound (set intersection instead of union, min instead of max, and so on). In the special case where the glb corresponds to $\bot_{\overline{\mathcal{BR}}}$, it is immediate to prove the two conditions (since $\bot_{\overline{\mathcal{BR}}}$ is smaller than any other element of the domain, and if the glb is $\bot_{\overline{\mathcal{BR}}}$ it cannot exist any other valid lower bound).

□

Lemma 3.6.14. *$\overline{\mathcal{BR}}$ is a lattice.*

Proof. A lattice is a partially ordered set in which every two elements have a join (also called a least upper bound) and a meet (also called a greatest lower bound). We already proved that the order $\leq_{\overline{\mathcal{BR}}}$ is a partial order (Theorem 3.6.11). Theorems 3.6.12 and 3.6.13 proved that $\sqcup_{\overline{\mathcal{BR}}}$ and $\sqcap_{\overline{\mathcal{BR}}}$ are the least upper bound and the greatest lower bound operators, respectively. □

Since the domain is not a complete lattice and it does not respect ACC, we will later ensure the convergence of the analysis through the definition of a widening operator, in order to define a limit to ascending chains of elements.

Abstraction and Concretization Functions

The concretization function maps an abstract element (i.e., a list of bricks) to a concrete element (i.e., a set of strings). Each brick represents a certain set of strings. The list of bricks thus represents all the strings built through the concatenation of strings which can be made from the bricks of the list (taken in the correct order). More formally, we define the strings represented by a single brick as:

$$\gamma_{\overline{\mathcal{B}}}(\overline{\mathsf{B}}) = \gamma_{\overline{\mathcal{B}}}([\overline{\mathsf{S}}]^{(m,M)}) = \bigcup_{j=m}^{M} (\underbrace{\overline{\mathsf{S}}\overline{\mathsf{S}}\ldots\overline{\mathsf{S}}}_{\text{j times}})$$

where $\underbrace{\overline{\mathsf{S}}\overline{\mathsf{S}}\ldots\overline{\mathsf{S}}}_{\text{j times}} = \overline{\mathsf{S}}^j$ stands for the concatenation between sets of strings (in particular, we concatenate $\overline{\mathsf{S}}$ to itself j times). To account for the case in which $j = 0$, we impose $\overline{\mathsf{S}}^0 = \{\epsilon\}$.

Let us see an example to clarify this definition. Consider the brick $[\{a, b\}]^{(1,3)}$ and let $\overline{\mathsf{S}} = \{a, b\}$. Then, the concretization of such brick is the following one:

$$\begin{aligned}\gamma_{\overline{\mathcal{B}}}([\{a, b\}]^{(1,3)}) &= \overline{\mathsf{S}} \cup \overline{\mathsf{S}}\overline{\mathsf{S}} \cup \overline{\mathsf{S}}\overline{\mathsf{S}}\overline{\mathsf{S}} = \\ &= \{a, b\} \cup \{aa, ab, ba, bb\} \cup \{aaa, aab, aba, abb, baa, bab, bba, bbb\} = \\ &= \{a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb\}\end{aligned}$$

Note that, if *min* had been 0 instead of 1, the result would have been:

$$\gamma_{\overline{\mathcal{B}}}([\{a, b\}]^{(0,3)}) = \overline{\mathsf{S}}^0 \cup \overline{\mathsf{S}}^1 \cup \overline{\mathsf{S}}^2 \cup \overline{\mathsf{S}}^3 = \epsilon \cup \overline{\mathsf{S}} \cup \overline{\mathsf{S}}\overline{\mathsf{S}} \cup \overline{\mathsf{S}}\overline{\mathsf{S}}\overline{\mathsf{S}}$$

The concretization function for lists of bricks is then the following one:

$$\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{B}}_0\overline{\mathsf{B}}_1 \ldots \overline{\mathsf{B}}_{N-1}) = \{s : s \in \mathsf{K}^* \wedge s = b_0 + b_1 + \cdots + b_{N-1} \wedge \forall i \in [0, N-1] : b_i \in \gamma_{\overline{\mathcal{B}}}(\overline{\mathsf{B}}_i)\}$$

where "+" represents the operator of string concatenation.

Theorem 3.6.15. *Let the abstraction function $\alpha_{\overline{\mathcal{BR}}}$ be defined by $\alpha_{\overline{\mathcal{BR}}} = \lambda\mathsf{Y}.\sqcap_{\overline{\mathcal{BR}}}\{\overline{\mathsf{B}} : \mathsf{Y} \subseteq \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{B}})\}$.*

Then $\langle \wp(\mathsf{S}), \subseteq\rangle \underset{\alpha_{\overline{\mathcal{BR}}}}{\overset{\gamma_{\overline{\mathcal{BR}}}}{\leftrightarrows}} \langle \overline{\mathcal{BR}}, \leq_{\overline{\mathcal{BR}}}\rangle$.

Proof. By Theorem 2.8.3 we only need to prove that $\gamma_{\overline{\mathcal{BR}}}$ is a complete meet morphism. Formally, we have to prove that $\gamma_{\overline{\mathcal{BR}}}(\bigsqcap_{\substack{\mathcal{BR} \\ (\overline{\mathsf{B}})\in\overline{\mathsf{X}}}} \overline{\mathsf{B}}) = \bigcap_{\overline{\mathsf{B}}\in\overline{\mathsf{X}}} \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{B}})$. For the sake of simplicity, we suppose that all list of bricks in $\overline{\mathsf{X}}$ contain n bricks.

$\gamma_{\overline{\mathcal{BR}}}(\bigsqcap_{\substack{\mathcal{BR} \\ (\overline{\mathsf{B}})\in\overline{\mathsf{X}}}} \overline{\mathsf{B}}) =$

By definition of $\sqcap_{\overline{\mathcal{BR}}}$

$= \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{B}}') : \forall i \in [0..n-1] : \overline{\mathsf{B}}'[i] = [\bigcap_{\overline{\mathsf{B}}\in\overline{\mathsf{X}}, \overline{\mathsf{B}}[i]=[\overline{\mathsf{S}}]^{(m,M)}} \overline{\mathsf{S}}]^{(\max_{\overline{\mathsf{B}}\in\overline{\mathsf{X}}, \overline{\mathsf{B}}[i]=[\overline{\mathsf{S}}]^{(m,M)}} m, \min_{\overline{\mathsf{B}}\in\overline{\mathsf{X}}, \overline{\mathsf{B}}[i]=[\overline{\mathsf{S}}]^{(m,M)}} M)}$

By definition of $\gamma_{\overline{\mathcal{BR}}}$

$= \{b_0 + .. + b_{n-1} : \forall i \in [0..n-1] : i_1 = \max_{\overline{\mathsf{B}}\in\overline{\mathsf{X}}, \overline{\mathsf{B}}[i]=[\overline{\mathsf{S}}]^{(m,M)}} m, i_2 = \min_{\overline{\mathsf{B}}\in\overline{\mathsf{X}}, \overline{\mathsf{B}}[i]=[\overline{\mathsf{S}}]^{(m,M)}} M,$
$b_i \in \bigcup_{j=i_1}^{i_2} (\bigcap_{\overline{\mathsf{B}}\in\overline{\mathsf{X}}, \overline{\mathsf{B}}[i]=[\overline{\mathsf{S}}]^{(m,M)}} \overline{\mathsf{S}})^j\}$

By definition of $\cap, \min, \max$

$= \{b_0 + .. + b_{n-1} : \forall i \in [0..n-1] : \forall \overline{\mathsf{B}} \in \overline{\mathsf{X}} : \overline{\mathsf{B}}[i] = [\overline{\mathsf{S}}]^{(m,M)}, b_i \in \bigcup_{j=m}^{M} \overline{\mathsf{S}}^j\}$

By definition of $\cap$

$= \bigcap_{\overline{\mathsf{B}}\in\overline{\mathsf{X}}} \{b_0 + .. + b_{n-1} : \forall i \in [0..n-1] : \overline{\mathsf{B}}[i] = [\overline{\mathsf{S}}]^{(m,M)}, b_i \in \bigcup_{j=m}^{M} \overline{\mathsf{S}}^j\}$

By definition of $\gamma_{\overline{\mathcal{BR}}}$

$= \bigcap_{\overline{\mathsf{B}}\in\overline{\mathsf{X}}} \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{B}})$

□

Now that we presented the concretization function, we can prove that the normalization of a list of bricks does not change its concretization, i.e. the set of strings it represents.

Lemma 3.6.16 (Soundness of the normalization rules)**.** *Given a normalization rule r_i ($i \in [1,5]$) and a list of bricks $\overline{\mathsf{L}}$, suppose that $\overline{\mathsf{L}}'$ is the list of bricks resulting from the application of r_i to $\overline{\mathsf{L}}$. Then, $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}) = \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}')$.*

Proof. We will prove the theorem for one rule at a time.

- r_1: trivial, since it just removes empty bricks which represent the empty string, i.e., the neutral element of concatenation.

- r_2: let $\overline{\mathsf{B_1}} = [S_1]^{(1,1)}$ and $\overline{\mathsf{B_2}} = [S_2]^{(1,1)}$ be the two bricks which Rule 2 merges. The first brick represents the strings in S_1 (since its indices are both 1), while the second represents, for the same reasons, the strings in S_2. The concatenation of these two bricks, then, represents the strings set $S_1\,S_2$ (remember that $S\,T$ represents the concatenation between the two strings sets S and T, i.e. the set containing all strings which can be obtained by concatenating a string from S and a string from T, in this order). Rule 2 transforms these two bricks in $\overline{\mathsf{B}}' = [S_1\,S_2]^{(1,1)}$, which represents exactly the same set of strings as the two original bricks, i.e. $S_1\,S_2$.

- r_3: let $\overline{\mathsf{B}} = [\mathsf{S}]^{(m,m)}$ be the brick which Rule 3 modifies. Its concretization is $\bigcup_{j=m}^{m}(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}}) = (\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{m times}}) = \mathsf{S}^m$. Rule 3 transforms such brick in $\overline{\mathsf{B}}' = [\mathsf{S}^m]^{(1,1)}$, which concretization is S^m, exactly the same as the original one.

- r_4: let $\overline{\mathsf{B_1}} = [\mathsf{S}]^{(m_1,M_1)}$ and $\overline{\mathsf{B_2}} = [\mathsf{S}]^{(m_2,M_2)}$ be the two bricks which Rule 4 merges. Their concretization is, respectively, $\bigcup_{j=m_1}^{M_1}(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}})$ and $\bigcup_{j=m_2}^{M_2}(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}})$. The concatenation of these two bricks represents the concatenation of their concretizations: $C_1 = \{(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{m_1\text{ times}}), \ldots, (\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{M_1\text{ times}})\}$ concatenated to $C_2 = \{(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{m_2\text{ times}}), \ldots, (\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{M_2\text{ times}})\}$. The result is the set $\{\mathsf{S_1S_2} : \mathsf{S}_1 \in C_1 \wedge \mathsf{S}_2 \in C_2\} = \{\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{j_1\text{ times}}\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{j_2\text{ times}} : j_1 \in [m_1, M_1] \wedge j_2 \in [m_2, M_2]\} = \{\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}} : j \in [m_1+m_2, M_1+M_2]\}$. Rule 4 merges these two bricks into the single brick $\overline{\mathsf{B}} = [\mathsf{S}]^{(m_1+m_2,M_1+M_2)}$, which concretization is $\bigcup_{j=(m_1+m_2)}^{M_1+M_2}(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}})$, exactly the same of the original one.

- r_5: let $\overline{\mathsf{B}} = [\mathsf{S}]^{(m,M)}$ be the brick which is split by Rule 5, where $m \geq 1 \wedge M \neq m$. Its concretization is $\bigcup_{j=m}^{M}(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}})$. Rule 5 transforms such brick in the

concatenation of the two bricks $\overline{\mathsf{B}_1} = [\mathsf{S}^m]^{(1,1)}$ and $\overline{\mathsf{B}_2} = [\mathsf{S}]^{(0,M-m)}$. Their concretizations are, respectively, $C_1 = \mathsf{S}^m$ and $C_2 = \bigcup_{j=0}^{M-m}(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}})$. The concatenation of these two bricks produces the set of strings $\{\mathsf{S}_1\mathsf{S}_2 : \mathsf{S}_1 = \mathsf{S}^m \wedge \mathsf{S}_2 \in C_2\} = \{\mathsf{S}^m\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}} : j \in [0, M-m]\} = \{\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}} : j \in [0+m, M-m+m]\} = \bigcup_{j=m}^{M}(\underbrace{\mathsf{SS}\ldots\mathsf{S}}_{\text{j times}})$.

□

Widening Operator

Let k_L, k_I and k_S be three constant integer values which will bound, respectively, the length of a bricks list, the indices range of a brick and the number of strings in the set of a brick. The widening operator is defined as follows:

$$\nabla_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2) = \begin{cases} \top_{\overline{\mathcal{BR}}} & \text{if } (\overline{\mathsf{L}}_1 \not\leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2 \wedge \overline{\mathsf{L}}_2 \not\leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_1) \vee \\ & \quad (\exists \mathsf{i} \in [1,2] : len(\overline{\mathsf{L}}_i) > \mathsf{k}_L) \\ w(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2) & \text{otherwise} \end{cases}$$

We return the $\top_{\overline{\mathcal{BR}}}$ element of our domain in two cases: (i) if the two abstract values are not comparable with respect to our order ($\overline{\mathsf{L}}_1 \not\leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2 \wedge \overline{\mathsf{L}}_2 \not\leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_1$), or (ii) if the length of one of the two lists is greater than the constant k_L ($\exists \mathsf{i} \in [1,2] : len(\overline{\mathsf{L}}_i) > \mathsf{k}_L$). Otherwise, we return $w(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2)$. Now we have to define what the function w does. Let us assume that $\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2$ and that $len(\overline{\mathsf{L}}_1) = len(\overline{\mathsf{L}}_2) = \mathsf{n}$. If the two lists were not of the same length, we could always add a proper number of empty bricks inside the shorter list using Algorithm 1. The definition of w is thus the following one:

$$w(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2) = [\overline{\mathcal{B}}_0^{\mathsf{new}}(\overline{\mathsf{L}}_1[0], \overline{\mathsf{L}}_2[0]); \overline{\mathcal{B}}_1^{\mathsf{new}}(\overline{\mathsf{L}}_1[1], \overline{\mathsf{L}}_2[1]); \ldots; \overline{\mathcal{B}}_{\mathsf{n}-1}^{\mathsf{new}}(\overline{\mathsf{L}}_1[\mathsf{n}-1], \overline{\mathsf{L}}_2[\mathsf{n}-1])]$$

where $\overline{\mathcal{B}}_i^{\mathsf{new}}(\overline{\mathsf{L}}_1[\mathsf{i}], \overline{\mathsf{L}}_2[\mathsf{i}])$ is defined by:

$$\overline{\mathcal{B}}_i^{\mathsf{new}}([\overline{\mathsf{S}}_{1i}]^{\mathsf{m}_{1i},\mathsf{M}_{1i}}, [\overline{\mathsf{S}}_{2i}]^{\mathsf{m}_{2i},\mathsf{M}_{2i}}) = \begin{cases} \top_{\overline{\mathcal{B}}} & \text{if } |\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}| > \mathsf{k}_S \\ & \quad \vee\ \overline{\mathsf{L}}_1[\mathsf{i}] = \top_{\overline{\mathcal{B}}} \vee \overline{\mathsf{L}}_2[\mathsf{i}] = \top_{\overline{\mathcal{B}}} \\ [\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}]^{(0,\infty)} & \text{if } (\mathsf{M}-\mathsf{m}) > \mathsf{k}_I \\ [\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}]^{(\mathsf{m},\mathsf{M})} & \text{otherwise} \end{cases}$$

where $\mathsf{m} = \min(\mathsf{m}_{1i}, \mathsf{m}_{2i})$ and $\mathsf{M} = \max(\mathsf{M}_{1i}, \mathsf{M}_{2i})$.

Let us briefly explain why this widening operator is correct. First of all, the result of a widening between two values must be greater or equal than both values. In our domain, the result of the widening between $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$ can be $\top_{\overline{\mathcal{BR}}}$ or $w(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2)$. If it is $\top_{\overline{\mathcal{BR}}}$, $\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \top_{\overline{\mathcal{BR}}}$ and $\overline{\mathsf{L}}_2 \leq_{\overline{\mathcal{BR}}} \top_{\overline{\mathcal{BR}}}$ follows from the fact that $\top_{\overline{\mathcal{BR}}}$ is the top element

of $\overline{\mathcal{BR}}$. In the other case, we know that $\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \overline{\mathsf{L}}_2$ or viceversa (for argument's sake, we assume that $\overline{\mathsf{L}}_1$ is the smaller value). Thus, the result of the widening is a new list in which each element $\overline{\mathcal{B}}_i^{\mathsf{new}}$ is the combination of $\overline{\mathsf{L}}_1[i]$ and $\overline{\mathsf{L}}_2[i]$. By definition of $\leq_{\overline{\mathcal{BR}}}$, to prove that $\overline{\mathsf{L}}_1 \leq_{\overline{\mathcal{BR}}} \nabla_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2)$ and $\overline{\mathsf{L}}_2 \leq_{\overline{\mathcal{BR}}} \nabla_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2)$ we just need to prove that $\overline{\mathsf{L}}_1[i] \leq_{\overline{\mathcal{B}}} \overline{\mathcal{B}}_i^{\mathsf{new}}$ and $\overline{\mathsf{L}}_2[i] \leq_{\overline{\mathcal{B}}} \overline{\mathcal{B}}_i^{\mathsf{new}} \forall i \in [0, \mathsf{n}-1]$, that is, that each brick of the result is greater or equal to the two corresponding bricks in $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$. By definition of $\overline{\mathcal{B}}_i^{\mathsf{new}}$ we have only three cases: (i) $\overline{\mathcal{B}}_i^{\mathsf{new}} = \top_{\overline{\mathcal{B}}}$, and so we have that $\overline{\mathsf{L}}_1[i] \leq_{\overline{\mathcal{B}}} \top_{\overline{\mathcal{B}}}$ and $\overline{\mathsf{L}}_2[i] \leq_{\overline{\mathcal{B}}} \top_{\overline{\mathcal{B}}}$ by definition of $\top_{\overline{\mathcal{B}}}$; (ii) $\overline{\mathcal{B}}_i^{\mathsf{new}} = [\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}]^{(0,\infty)}$; in this case $\overline{\mathsf{L}}_1[i] = [\overline{\mathsf{S}}_{1i}]^{\mathsf{m}_{1i},\mathsf{M}_{1i}} \leq_{\overline{\mathcal{B}}} [\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}]^{(0,\infty)}$ since $\overline{\mathsf{S}}_{1i} \subseteq (\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}) \wedge 0 \leq \mathsf{m}_{1i} \wedge \mathsf{M}_{1i} \leq +\infty$. The same happens for $\overline{\mathsf{L}}_2[i]$; (iii) $\overline{\mathcal{B}}_i^{\mathsf{new}} = [\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}]^{(\mathsf{m},\mathsf{M})}$ where $\mathsf{m} = \min(\mathsf{m}_{1i}, \mathsf{m}_{2i})$ and $\mathsf{M} = \max(\mathsf{M}_{1i}, \mathsf{M}_{2i})$. In this case we have $\overline{\mathsf{L}}_1[i] = [\overline{\mathsf{S}}_{1i}]^{\mathsf{m}_{1i},\mathsf{M}_{1i}} \leq_{\overline{\mathcal{B}}} [\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}]^{\mathsf{m},\mathsf{M}}$ because $\overline{\mathsf{S}}_{1i} \subseteq (\overline{\mathsf{S}}_{1i} \cup \overline{\mathsf{S}}_{2i}) \wedge \mathsf{m} = \min(\mathsf{m}_{1i}, \mathsf{m}_{2i}) \leq \mathsf{m}_{1i} \wedge \mathsf{M}_{1i} \leq \mathsf{M} = \max(\mathsf{M}_{1i}, \mathsf{M}_{2i})$. The same happens for $\overline{\mathsf{L}}_2[i]$.

Then, we need the widening operator to be convergent. In other words, given an ascending chain $\overline{\mathsf{s}_\mathsf{n}}$, the sequence $(\overline{\mathsf{t}_{\mathsf{n}+1}} = \nabla_{\overline{\mathcal{BR}}}(\overline{\mathsf{t}_\mathsf{n}}, \overline{\mathsf{s}_\mathsf{n}}))$ has to be ultimately stationary. In our case, a value of an ascending chain can increase along three axes: (i) the length of the brick list, (ii) the indices range of a certain brick, and (iii) the strings contained in a certain brick. The growth of an abstract value is bounded along each axis with the help of the three constants k_L,k_S, and k_I. After the list has reached k_L elements, the entire abstract value is approximated to $\top_{\overline{\mathcal{BR}}}$, stopping its possible growth altogether. If the range of a certain brick becomes larger than k_I, the range is approximated to $(0, +\infty)$, stopping the indices possible growth. Finally, if the strings set of a certain brick reaches k_S elements, the brick is approximated to $\top_{\overline{\mathcal{B}}}$, stopping its possible growth altogether.

Gaining More Precision

Normalized values are important in definition of operators like least upper bound and widening. However, normalizing values after each operation is costly and, worse than that, it could entail a big loss of precision (which we documented while analyzing our case studies). For example, the result of the $\overline{\mathcal{BR}}$ domain on the second case study (`prog2`), when normalizing values after *each* operation, is $\top_{\overline{\mathcal{BR}}}$, that is, we are not able to track any kind of information on the program. For these two reasons (performance and, most importantly, precision), we choose to normalize abstract values only *after* executing the least upper bound operator or the widening operator. Any other operation (regarding both the abstract semantics and the lattice) will not be followed by a normalization step.

Semantics

Table 3.7 defines the abstract semantics of string operators in $\overline{\mathcal{BR}}$.

Let us explain in detail the semantics of each operator:

Table 3.7: The abstract semantics of $\overline{\mathcal{BR}}$

$$\overline{\mathbb{S}_{\mathcal{BR}}}[\![\texttt{new String(str)}]\!]() = [\{\texttt{str}\}]^{1,1}$$
$$\overline{\mathbb{S}_{\mathcal{BR}}}[\![\texttt{concat}]\!](\overline{\mathsf{b}}_1, \overline{\mathsf{b}}_2) = \overline{concatList}(\overline{\mathsf{b}}_1, \overline{\mathsf{b}}_2)$$
$$\overline{\mathbb{S}_{\mathcal{BR}}}[\![\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}]\!](\overline{\mathsf{b}}) = \begin{cases} [\overline{\mathsf{T}}']^{1,1} & \text{if } \overline{\mathsf{b}}'[0] = [\overline{\mathsf{T}}]^{1,1} \wedge \forall \bar{\mathsf{t}} \in \overline{\mathsf{T}} : len(\bar{\mathsf{t}}) \geq \mathsf{e} \\ \top_{\overline{\mathcal{BR}}} & \text{otherwise} \end{cases}$$
where $\overline{\mathsf{T}}' = \{\bar{\mathsf{t}}.\texttt{substring}(\mathsf{b}, \mathsf{e}) \; \forall \bar{\mathsf{t}} \in \overline{\mathsf{T}}\} \wedge \overline{\mathsf{b}}' = \overline{normBricks}(\overline{\mathsf{b}})$
$$\overline{\mathbb{B}_{\mathcal{BR}}}[\![\texttt{contains}_{\mathsf{c}}]\!](\overline{\mathsf{b}}) =$$
$$\begin{cases} \mathsf{true} & \text{if } \exists \overline{\mathsf{B}} \in \overline{\mathsf{b}} : \overline{\mathsf{B}} = [\overline{\mathsf{T}}]^{m,M} \wedge 1 \leq m \leq M \wedge (\forall \bar{\mathsf{t}} \in \overline{\mathsf{T}} : \mathsf{c} \in char(\bar{\mathsf{t}})) \\ \mathsf{false} & \text{if } \forall [\overline{\mathsf{T}}]^{\mathsf{m},\mathsf{M}} \in \overline{\mathsf{b}}, \forall \bar{\mathsf{t}} \in \overline{\mathsf{T}} : \mathsf{c} \notin char(\bar{\mathsf{t}}) \\ \top_{\mathsf{B}} & \text{otherwise} \end{cases}$$

- When a constant string value is evaluated (`new String(str)`), the semantics returns a single brick containing exactly that string with $[1, 1]$ as indices.
- For the concatenation of two strings, we rely on the $\overline{concatList}$ function that concatenates two lists of bricks.
- To define the semantics of $\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}$, we first normalize the abstract value in input (we can do that since we know, by Lemma 3.6.16, that the normalization does not change the set of represented strings). Remember that, in a normalized list of bricks, each brick is in the form $[\overline{\mathsf{T}}]^{(0,max>0)}$ or $[\overline{\mathsf{T}}]^{(1,1)}$. If the first brick of the normalized abstract value $\overline{\mathsf{b}}'$ has the form $[\overline{\mathsf{T}}]^{(0,max>0)}$, then we have too much uncertainty on how the string begins: we cannot compute a substring based on start and end indices. Instead, if the first brick has the form $[\overline{\mathsf{T}}]^{(1,1)}$ then we are sure that the string will begin with any of the strings in $\overline{\mathsf{T}}$. If all the strings in $\overline{\mathsf{T}}$ are long enough ($len(\bar{\mathsf{t}}) \geq \mathsf{e} \; \forall \bar{\mathsf{t}} \in \overline{\mathsf{T}}$) we can pack all the possible substrings in a new abstract value, which we will return.
- The semantics of $\texttt{contains}_{\mathsf{c}}$ returns true iff the character c appears in all the strings of a certain brick with minimal index $min \geq 1$. It returns false iff we are sure that c does not appear in any string of any brick of the abstract value. Otherwise, we have to return $\top_{\mathsf{B}}$.

We now prove the soundness of the abstract operations defined above.

Theorem 3.6.17 (Soundness of the abstract semantics)**.** $\overline{\mathbb{S}_{\overline{\mathcal{BR}}}}$ *and* $\overline{\mathbb{B}_{\overline{\mathcal{BR}}}}$ *are a sound overapproximation of* $\mathbb{S}$ *and* $\mathbb{B}$*, respectively. Formally,* $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathbb{S}_{\mathcal{BR}}}[\![\mathsf{s}]\!](\overline{\mathsf{L}})) \supseteq \{\mathbb{S}[\![\mathsf{s}]\!](\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}})\}$ *and* $\gamma_{\overline{\mathcal{PR}}}(\overline{\mathbb{B}_{\mathcal{BR}}}[\![\mathsf{s}]\!](\overline{\mathsf{L}})) \geq_{\mathsf{B}} \{\mathbb{B}[\![\mathsf{s}]\!](\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}})\}$.

Proof. We prove the soundness separately for each operator.

- $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathbb{S}_{\mathcal{BR}}}[\![\texttt{new String(str)}]\!]()) \supseteq \{\mathbb{S}[\![\texttt{new String(str)}]\!]()\}$ follows immediately from the definition of $\overline{\mathbb{S}_{\mathcal{BR}}}[\![\texttt{new String(str)}]\!]()$ and of $\gamma_{\overline{\mathcal{BR}}}$.

- Consider the binary operator `concat`. Let $\overline{\mathsf{L}}_1$ and $\overline{\mathsf{L}}_2$ be two lists of bricks. We have to prove that $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathbb{S}_{\mathcal{BR}}}[\![\texttt{concat}]\!](\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2)) \supseteq \{\mathbb{S}[\![\texttt{concat}]\!](\mathsf{c}_1, \mathsf{c}_2) : \mathsf{c}_1 \in \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1) \wedge \mathsf{c}_2 \in \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_2)\}$. Let s be an element in $\{\mathbb{S}[\![\texttt{concat}]\!](\mathsf{c}_1, \mathsf{c}_2) : \mathsf{c}_1 \in \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1) \wedge \mathsf{c}_2 \in \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_2)\}$. By definition of $\mathbb{S}$, this means that there exist two strings $\mathsf{c}_1, \mathsf{c}_2$ such that $\mathsf{s} = \mathsf{c}_1 + \mathsf{c}_2$ and that $\mathsf{c}_1 \in \gamma(\overline{\mathsf{L}}_1) \wedge \mathsf{c}_2 \in \gamma(\overline{\mathsf{L}}_2)$. On the other hand, $\overline{\mathbb{S}_{\mathcal{BR}}}[\![\texttt{concat}]\!](\overline{\mathsf{L}}_1, \overline{\mathsf{L}}_2)$ produces a new list of bricks $\overline{\mathsf{L}}$ which concatenates the two lists in input by definition of $\overline{\mathbb{S}_{\mathcal{BR}}}$ By the definition of $\gamma_{\overline{\mathcal{BR}}}$ and the associative property of the concatenation between strings, we can say that the strings belonging $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}})$ are all the strings obtained through the concatenation of one string belonging to $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_1)$ and another belonging to $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}}_2)$. Then, surely s belongs to $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}})$.

- Consider the unary operator $\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}$ and let $\overline{\mathsf{L}}$ be a (normalized) list of bricks. Consider the following cases:

  - if $\overline{\mathsf{L}}[0] = [\overline{\mathsf{T}}]^{1,1} \wedge \forall \overline{\mathsf{t}} \in \overline{\mathsf{T}} : \mathit{len}(\overline{\mathsf{t}}) \geq \mathsf{e}$, then we have that $\gamma_{\overline{\mathcal{B}}}(\overline{\mathsf{L}}[0]) = \overline{\mathsf{T}}$. Thus, all the strings in $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}})$ have as prefix one of the strings of $\overline{\mathsf{T}}$, by definition of $\gamma_{\overline{\mathcal{BR}}}$. Moreover, by hypothesis all strings of $\overline{\mathsf{T}}$ are longer than e characters. Then, a string belonging to $\{\mathbb{S}[\![\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}]\!](\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}})\}$ is certainly a substring of one string of $\overline{\mathsf{T}}$, from the b-th character to the e-th character, by definition of $\mathbb{S}$. This corresponds exactly to $\gamma_{\mathcal{BR}}(\overline{\mathbb{S}_{\mathcal{BR}}}[\![\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}]\!](\overline{\mathsf{L}}))$, since the abstract semantics applied to $\overline{\mathsf{L}}$ produces a single brick containing the substrings of all strings in $\overline{\mathsf{T}}$, from the b-th character to the e-th character.
  - otherwise, the abstract semantics returns $\top_{\overline{\mathcal{BR}}}$, that soundly approximates any possible result of the concrete semantics.

- Consider the unary operator $\texttt{contains}_{\mathsf{c}}$ and let $\overline{\mathsf{L}}$ be a list of bricks. Regarding the character c, we have three possible cases:

  - if $\exists \overline{\mathsf{b}} \in \overline{\mathsf{L}} : \overline{\mathsf{b}} = [\overline{\mathsf{T}}]^{m,M} \wedge 1 \leq m \leq M \wedge (\forall \overline{\mathsf{t}} \in \overline{\mathsf{T}} : \mathsf{c} \in \mathit{char}(\overline{\mathsf{t}}))$, this means that there exists at least one brick whose strings all contain the character c. Let $\overline{\mathsf{b}}$ be this brick. Then, all the strings belonging to $\gamma_{\overline{\mathcal{B}}}(\overline{\mathsf{b}})$ contain the character c, since its minimum index is ≥ 1. $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}})$ concatenates all the concretizations of its bricks, so each string belonging to this concretization surely contains the character c. The result of the concrete semantics is, then, always `true`. Since $\overline{\mathbb{B}_{\mathcal{BR}}}[\![\texttt{contains}_{\mathsf{c}}]\!](\overline{\mathsf{L}}) = \mathsf{true}$, the abstract semantics is a sound approximation of the concrete semantics.
  - if $\forall [\overline{\mathsf{T}}]^{\mathsf{m},\mathsf{M}} \in \overline{\mathsf{L}}, \forall \overline{\mathsf{t}} \in \overline{\mathsf{T}} : \mathsf{c} \notin \mathit{char}(\overline{\mathsf{t}})$, this means that no brick in the list $\overline{\mathsf{L}}$ has a string containing the character c. Then, no concrete string in $\gamma_{\overline{\mathcal{BR}}}(\overline{\mathsf{L}})$ contains such character, and for this reason the concrete semantics

#I	Var	$\overline{\mathcal{BR}}$
1	query	$[\{\mathtt{s}_1\}]^{1,1}$
3	l	$\top_{\overline{\mathcal{B}}}$
4	query	$[\{\mathtt{s}_1\}]^{1,1}[\{\mathtt{s}_2\}]^{1,1}\top_{\overline{\mathcal{B}}}[\{\mathtt{s}_3\}]^{1,1}$
5	query	$[\{\mathtt{s}_1\}]^{1,1}[\{\mathtt{s}_2\}]^{0,1}\top_{\overline{\mathcal{B}}}[\{\mathtt{s}_3\}]^{0,1}$
6	per	$[\{\mathtt{s}_4\}]^{1,1}$
8	query	$[\{\mathtt{s}_1\}]^{1,1}[\{\mathtt{s}_2\}]^{0,1}\top_{\overline{\mathcal{B}}}[\{\mathtt{s}_3\}]^{0,1}$ $[\{\mathtt{s}_5\}]^{1,1}[\{\mathtt{s}_4\}]^{1,1}[\{\mathtt{s}_6\}]^{1,1}$

(a) Analysis of `prog1`

#I	Var	$\overline{\mathcal{BR}}$
1	x	$[\{\text{“}a\text{”}\}]^{1,1}$
3	x	$[\{\text{“0”}\}]^{0,1}[\{\text{“}a\text{”}\}]^{1,1}[\{\text{“1”}\}]^{0,1}$
4	x	$[\{\text{“0”}\}]^{0,+\infty}[\{\text{“}a\text{”}\}]^{1,1}[\{\text{“1”}\}]^{0,+\infty}$

(b) Analysis of `prog2`

Figure 3.6: The results of $\overline{\mathcal{BR}}$

always returns `false`. Since $\overline{\mathbb{B}_{\mathcal{BR}}}[\![\mathtt{contains_c}]\!](\overline{\mathsf{L}}) = \mathsf{false}$, this precisely approximates the results of the concrete semantics.

– otherwise, the abstract semantics on $\overline{\mathsf{L}}$ returns $\top_{\mathsf{B}}$, and the property is immediately proven, since $\top_{\mathsf{B}}$ is a superset of any set of boolean values.

□

Case studies

The results of the analysis of the two case studies using $\overline{\mathcal{BR}}$ are depicted in Figures 3.6(a) and 3.6(b).

For `prog1`, at line 1 we represent `query` with a single brick with a singleton set (containing $\mathtt{s}_1$, the string associated to `query`) and indices min = max = 1. The variable `l` has an unknown value, so it is associated to $\top_{\overline{\mathcal{BR}}}$. At line 4 we concatenate the value of `query` to $\mathtt{s}_2$, `l` and $\mathtt{s}_3$ and we obtain a list of four bricks: the first two are made up by a singleton set (containing, respectively, $\mathtt{s}_1$ and $\mathtt{s}_2$) and indices min = max = 1, the third one is $\top_{\overline{\mathcal{BR}}}$ (because of `l`), and the fourth one is made up by a singleton set (containing $\mathtt{s}_3$) and indices min = max = 1. This means that we know that, just after line 4, the string associated to `query` starts with $\mathtt{s}_1 + \mathtt{s}_2$, then it has an unknown part, and then it ends with $\mathtt{s}_3$. Then, we have to compute the lub between the values of `query` after lines 1 and 4. To do this, firstly we use Algorithm 1 to make the two lists have the same size: the algorithm adds three empty bricks at the end of the bricks list of the abstract value at line 1, thus maintaining the correspondence between $[\{\mathtt{s}_1\}]^{1,1}$ in the two lists. The result

of the lub is, again, a list of four bricks: the first one is made up by a singleton set (containing $\mathtt{s}_1$) and indices min = max = 1, the second one is made up by another singleton set (containing $\mathtt{s}_2$) and indices min = 0, max = 1, the third one is $\top_{\overline{\mathcal{BR}}}$ (because of `l`) and the last one is made up by a singleton set (containing $\mathtt{s}_3$) and indices min = 0, max = 1. This means that we know that, just after line 5, the string associated to `query` surely starts with $\mathtt{s}_1$, then it *could* continue with $\mathtt{s}_2$, then it has an unknown part and then it *could* end with $\mathtt{s}_3$. At line 7, the abstract value of the variable `per` is composed by a single brick with a singleton set (containing $\mathtt{s}_4$, the string associated to `per`) and indices min = max = 1. Finally, at line 8 there is another concatenation. The bricks of the abstract value associated to `query` after line 8 are seven: (i) the first brick represents the string $\mathtt{s}_1$, (ii) the second brick could be the empty string ϵ or $\mathtt{s}_2$, (iii) the third brick corresponds to the (unknown) input `l`, (iv) the fourth brick could be the empty string ϵ or $\mathtt{s}_3$, and (v) the last three bricks represent the concatenation of $\mathtt{s}_5$, $\mathtt{s}_4$, and $\mathtt{s}_6$. We can see that the precision is higher than in the previous domains, but still not the best we aim to get: amongst the concrete results we have, for example, $\mathtt{s}_1 + \mathtt{s}_3 + \mathtt{s}_5 + \mathtt{s}_4 + \mathtt{s}_6$, which cannot be computed in any execution of the analyzed code.

For `prog2`, after line 1 the abstract value associated to `x` is a single brick with a singleton set (containing "a") and indices min = max = 1. After the first iteration of the loop, the result of the concatenation is made up by three bricks, all of them with a singleton set (containing, respectively, "0", "a" and "1") and indices min = max = 1. To compute the least upper bound between this value and the value of `x` *before* the loop ($[\{\text{“}a\text{”}\}]^{1,1}$) we first execute Algorithm 1, obtaining the new list $E[\{\text{“}a\text{”}\}]^{1,1}E$ instead of just $[\{\text{“}a\text{”}\}]^{1,1}$. The result of the lub is then the abstract value $[\{\text{“0”}\}]^{0,1}[\{\text{“}a\text{”}\}]^{1,1}[\{\text{“1”}\}]^{0,1}$. The normalization step does not change this abstract value. Starting from this value, we execute the second iteration, and we obtain $[\{\text{“0”}\}]^{1,1}[\{\text{“0”}\}]^{0,1}[\{\text{“}a\text{”}\}]^{1,1}[\{\text{“1”}\}]^{0,1}[\{\text{“1”}\}]^{1,1}$. To compute the least upper bound between the values after the first and second iterations (we do not know how many iterations the loop will do), we apply Algorithm 1 on the shorter list, obtaining the new list $E[\{\text{“0”}\}]^{0,1}[\{\text{“}a\text{”}\}]^{1,1}[\{\text{“1”}\}]^{0,1}E$. The result of the lub is then the abstract value $[\{\text{“0”}\}]^{0,1}[\{\text{“0”}\}]^{0,1}[\{\text{“}a\text{”}\}]^{1,1}[\{\text{“1”}\}]^{0,1}[\{\text{“1”}\}]^{0,1}$, which, after the normalization step, becomes $[\{\text{“0”}\}]^{0,2}[\{\text{“}a\text{”}\}]^{1,1}[\{\text{“1”}\}]^{0,2}$. Following the same reasoning, after the third iteration we obtain $[\{\text{“0”}\}]^{1,1}[\{\text{“0”}\}]^{0,2}[\{\text{“}a\text{”}\}]^{1,1}[\{\text{“1”}\}]^{0,2}[\{\text{“1”}\}]^{1,1}$ which becomes $[\{\text{“0”}\}]^{0,3}[\{\text{“}a\text{”}\}]^{1,1}[\{\text{“1”}\}]^{0,3}$ after the lub with the value of the previous iteration and after the normalization step. We can see that, after each iteration, we obtain an abstract value which first and last bricks have an augmented range with respect to the value in the previous iteration: min is always zero, but max increases by one at each iteration. The convergence of the analysis is obtained through to the use of the widening operator, which, when a brick's indices range reaches the threshold k_I, forces the range of the brick to min = 0, max = $+\infty$. Since k_I is a constant value, we will certainly reach it after a finite number of iterations. Therefore, after the loop, we associate `x` to $[\{\text{“0”}\}]^{0,+\infty}[\{\text{“}a\text{”}\}]^{1,1}[\{\text{“1”}\}]^{0,+\infty}$. The result is almost optimal: the imprecision is due to the fact the number of occurrences of

0s and 1s are not restricted to be the same. For example, $0a11$ is a concrete value represented by our resulting abstraction, but we know that this string can never be produced by the program `prog2`.

3.6.4 String Graphs

In the first domain we presented ($\overline{\mathcal{CI}}$) the only focus of the approximation was character inclusion. In the next two domains ($\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$) we also considered order, but limited at the beginning (prefix) or at the end (suffix) of the string. In the $\overline{\mathcal{BR}}$ domain we considered (like in $\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$) both inclusion and order among characters, but this time it was not limited to the beginning or the end of the string. $\overline{\mathcal{BR}}$ approximates a string with a list of bricks, where each brick represents a set of strings. The precision of this domain is definitely better than that of the previous ones, as it was made clear by the analysis of `prog1` and `prog2`. We obtained very good results analyzing such programs, even though there is still room for improvements. The new abstract domain we are going to present in this section tracks a kind of information similar to the one tracked by $\overline{\mathcal{BR}}$ (inclusion and order), but equipped with more precise lattice and semantics operators. This domain exploits type graphs [104], a data structure which represents tree automata, and adapts them to represent set of strings. Type graphs were introduced in 1992 by Janssens & Bruynooghe, when they developed a method for obtaining descriptions of possible values of program variables (extended modes or a kind of type information). Their method was based upon a framework for Abstract Interpretation. Many of the concepts we are going to present about string graphs come from the original definition of type graphs, and we refer the interested reader to [104] for more details about them.

Domain Definition

A string graph $\overline{\mathsf{T}}$ is a triple $(\overline{\mathsf{N}}, \overline{\mathsf{A}}_F, \overline{\mathsf{A}}_B)$ where $\overline{\mathsf{T}}_r = (\overline{\mathsf{N}}, \overline{\mathsf{A}}_F)$ is a rooted tree whose arcs in $\overline{\mathsf{A}}_F$ are called forward arcs, and $\overline{\mathsf{A}}_B$ is a restricted class of arcs, backward arcs, superimposed on $\overline{\mathsf{T}}_r$. Ancestors and descendants are defined in the usual way. The backward arcs $(\overline{\mathsf{n}}, \overline{\mathsf{m}})$ in $\overline{\mathsf{A}}_B$, have the property that $\overline{\mathsf{m}}$ belongs to the ancestors of $\overline{\mathsf{n}}$. A forward path is a path composed of forward arcs. The depth of a node $\overline{\mathsf{n}}$, denoted by $\overline{depth}(\overline{\mathsf{n}})$, is the length of the shortest path from the root of the type graph to $\overline{\mathsf{n}}$. We use the convention that $\overline{\mathsf{n}}/\mathsf{i}$ denotes the i-th son of node $\overline{\mathsf{n}}$, and the set of sons of a node $\overline{\mathsf{n}}$ is then denoted as $\{\overline{\mathsf{n}}/1, \ldots, \overline{\mathsf{n}}/k\}$ with $\overline{\mathsf{k}} = \overline{outdegree}(\overline{\mathsf{n}})$ where $\overline{outdegree}$ is a function that, given a node, returns the number of its sons. We also define the $\overline{indegree}$ function, which, given a node, returns the number of its predecessors. The root of the tree (i.e., the only node with no incoming forward arcs) is called $\overline{\mathsf{n}}_0$.

Each node $\overline{\mathsf{n}} \in \overline{\mathsf{N}}$ of a string graph has a label, denoted by $\overline{lb}(\overline{\mathsf{n}})$, indicating the kind of term it describes. The nodes are divided into three classes:

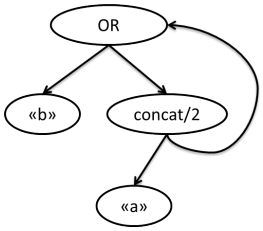


Figure 3.7: An example of string graph

- *Simple nodes* have a label from the set $\{max, \bot_{\overline{\mathcal{SG}}}, \epsilon\} \cup \mathsf{K}$. This means that the leaves of string graphs trees can represent (i) all possible strings, K^* (if the node has label max), (ii) no strings, $\emptyset$ (if the node has label $\bot_{\overline{\mathcal{SG}}}$), (iii) the empty string (if the node has label ϵ), and (iv) a string made by a single character taken from the alphabet K, respectively.

- *Concat nodes* are labelled with the functor $concat/k$ (with the obvious meaning of string concatenation) and have outdegree k with $k > 0$;

- *OR nodes* have the label OR and an outdegree k.

The graphical representation of string graphs is straightforward. The nodes of a string graph are represented by their labels and every node is encircled. The direction of the arc is indicated by its arrow: forward arcs are drawn downwards, backward arcs upwards. The root of the string graph is the topmost node. An example is depicted in Figure 3.7. The root of the string graph is an OR-node with two sons: (i) a simple node (b) , and (ii) a *concat*-node with two sons of its own (a simple node (a), and the root (with the use of a backward arc)). This string graph represents an infinite set of strings, that is the set of strings which start with an indefinite number of a (even zero) and surely end with a b, that is, $\{b, ab, aab, aaab, \dots\} = a^*b$.

The structure of the string graph together with the labels of its nodes determines the set of represented strings. The set of finite strings represented by a node $\overline{\mathsf{n}}$ in the string graph $\overline{\mathsf{T}}$ is said to be the *denotation* of the node $\overline{\mathsf{n}}$, $\mathbb{D}(\overline{\mathsf{n}})$.

Definition 3.6.1. *The denotation $\mathbb{D}(\overline{\mathsf{n}})$ of a node $\overline{\mathsf{n}}$ in a string graph is defined as follows:*

function $\mathbb{D}(\overline{\mathsf{n}})$
  if $lb(\overline{\mathsf{n}}) = max$ ***then***
    return K^*
  else if $lb(\overline{\mathsf{n}}) = \bot$ ***then***
    return $\emptyset$
  else if $lb(\overline{\mathsf{n}}) \in \mathsf{K} \vee lb(\overline{\mathsf{n}}) = \epsilon$ ***then***
    return $\{lb(\overline{\mathsf{n}})\}$
  else if $lb(\overline{\mathsf{n}}) = concat/k$ *and* $\overline{\mathsf{n}}/1, \dots, \overline{\mathsf{n}}/k$ *are its sons* ***then***

return $\{concat(t_1, \ldots, t_k) : t_i$ *is finite* $\wedge\, t_i \in \mathbb{D}(\overline{\mathsf{n}}/i)\ \forall i \in [1,k]\}$
else
return $\bigcup_{i=1}^{k} \mathbb{D}(\overline{\mathsf{n}}/i)$, *as* $lb(\overline{\mathsf{n}}) = OR$ *and* $\overline{\mathsf{n}}/1, \ldots, \overline{\mathsf{n}}/k$ *are its sons*
end if
end function

The order of the sons of a concat node is important because string concatenation is not commutative, whereas the order of the sons of an OR-node is irrelevant. $\mathbb{D}(\overline{\mathsf{n}})$ can be $\emptyset$ or a (finite or infinite) set of finite strings. With $\overline{\mathsf{n}}_0$ the root of string graph $\overline{\mathsf{T}}$, we use $\mathbb{D}(\overline{\mathsf{T}})$ as a synonym for $\mathbb{D}(\overline{\mathsf{n}}_0)$.

Note that several distinct string graphs can have the same denotation. The existence of superfluous nodes and arcs makes some operators, such as $\leq$, quite complex and inefficient. To reduce this variety of string graphs, we impose some additional restrictions, which correspond to the definition of *compact type graphs* in [104] (where you can also find a compaction algorithm). For example, one of these restrictions is that an OR-node must have strictly more than one son and each son must not be a *max*-node. The denotation of the string graph is preserved when carrying out a compaction, i.e. the set of represented strings does not change.

Notice also that compact string graphs are not the most economical representation. Nodes in different branches can have the same denotation. In particular, different sons of an OR node may have overlapping, even identical denotations. This makes testing whether a particular string is in the denotation of a compact string graph and the comparison of the denotations of two string graphs inefficient, so we impose a further restriction which will result in the definition of normal string graphs. Such restriction limits the expressive power of the string graphs but is necessary to achieve efficient operations. First, we introduce two functions, *prnd* and *prlb*. The function $prnd(\overline{\mathsf{n}})$ denotes the set of *principal nodes* of a node $\overline{\mathsf{n}}$, and $prlb(\overline{\mathsf{n}})$ its set of *principal labels*.

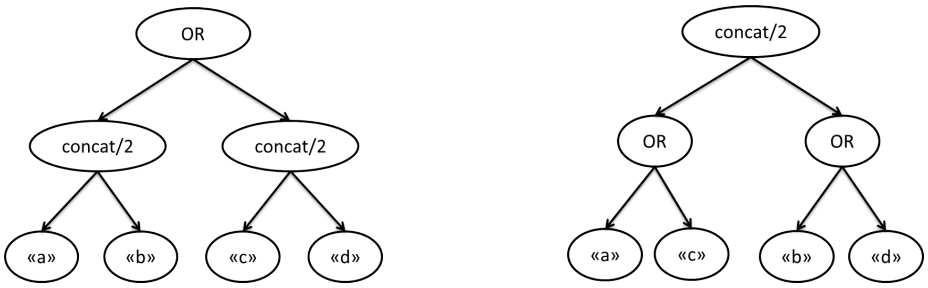
$$prnd(\overline{\mathsf{n}}) = \begin{cases} \bigcup_{i=1}^{k} prnd(\overline{\mathsf{n}}/i) & \text{if } lb(\overline{\mathsf{n}}) = OR \wedge k = \overline{outdegree}(\overline{\mathsf{n}}) \\ \overline{\mathsf{n}} & \text{else} \end{cases}$$

$$prlb(\overline{\mathsf{n}}) = \{lb(\overline{\mathsf{n}}_j) : \overline{\mathsf{n}}_j \in prnd(\overline{\mathsf{n}})\}$$

Two sets of principal labels are *overlapping* if their intersection is not empty.

Definition 3.6.2 (Principal label restriction)**.** *The principal label restriction states that each pair of sons of an OR-node must have non-overlapping sets of principal labels.*

Normal string graphs are compact string graphs satisfying the principal label restriction. In [103] you can find the definition of a normalization algorithm, $normalize(\overline{\mathsf{T}})$, which takes in input a *compact* type graph and returns in output the corresponding *normal* type graph. Adapting it to string graphs is straightforward. The principal label restriction limits the expressiveness of string graphs:



(a) Compact string graph before normalization (b) Normal string graph

Figure 3.8: An example of string graphs normalization

string graphs violating this restriction sometimes have to be replaced by a string graph denoting a larger set of strings. An example of compact string graph before and after normalization is depicted in Figure 3.8. The string graph in Fig. 3.8(a) does not satisfy the principal label restriction, since the two sons of the root node have the same label $concat/2$; its denotation is $\{ab, cd\}$. The string graph in Fig. 3.8(b) is normal; its denotation is $\{ab, ad, cb, cd\}$, a larger set than $\{ab, cd\}$. In fact, the normalization process makes us lose the information that, when the first character of the string is a, then the second is always b (and the same for c and d).

Normal string graphs must also satisfy, besides the principal label restriction, other four restrictions (not present in the original definition of normal *type* graphs), which we are now going to introduce.

Rule 1 Given a node $\overline{\mathsf{n}}$ with label $concat/1$ and $\overline{\mathsf{n}}/1$ as successor, replace $\overline{\mathsf{n}}$ with $\overline{\mathsf{n}'} = \overline{\mathsf{n}}/1$. Any backward arc $(\overline{\mathsf{m}}, \overline{\mathsf{n}})$ should be replaced with the arc $(\overline{\mathsf{m}}, \overline{\mathsf{n}'})$. This rule simplifies some naïve occurrences of the functor $concat/k$. In fact, when *concat* has only one son ($k = 1$), the result of its application is the argument itself. We thus discard every $concat/1$ node, replacing it with its argument.

Rule 2 Given a node $\overline{\mathsf{n}}$ with label $concat/k$ such that $\overline{\mathsf{n}}/i = max\ \forall i \in [1, k]$, replace $\overline{\mathsf{n}}$ with $\overline{\mathsf{n}}' = max$. This rule simplifies a node with label $concat/k$ and which successors $\overline{\mathsf{n}}/i$ all have the label max. In fact, the concatenation of all possible strings with all possible strings gives us all possible strings, again.

Rule 3 Given a node $\overline{\mathsf{n}}$ with label $concat/k$ such that $\exists i : \overline{\mathsf{n}}/i = concat/k_1 \wedge \overline{\mathsf{n}}/(i+1) = concat/k_2$, $indegree(\overline{\mathsf{n}}/i) = 1$ and $indegree(\overline{\mathsf{n}}/(i+1)) = 1$, replace $\overline{\mathsf{n}}/i$ and $\overline{\mathsf{n}}/(i+1)$ with a single new node $\overline{\mathsf{n}'} = concat/(k_1 + k_2)$ whose sons are

$$\overline{\mathsf{n}}'/j = \begin{cases} (\overline{\mathsf{n}}/i)/j & \text{if } j \leq k_1 \\ (\overline{\mathsf{n}}/(i+1))/(j-k_1) & \text{otherwise} \end{cases}$$

where $j \in [1, k_1 + k_2]$. This rule merges two successive sons of a *concat*-node, which labels are both *concat*. In fact, if we concat some characters obtaining

the string s_1, then we concat some other characters obtaining the string s_2, and finally we concat s_1 and s_2, we obtain the same result as concatenating *all* the characters in the first place.

Rule 4 Given a node $\overline{\mathsf{n}}$ with label $concat/k$ such that $\exists i : \overline{\mathsf{n}}/i = concat/k_1 \wedge \overline{indegree}(\overline{\mathsf{n}}/i) = 1$, replace $\overline{\mathsf{n}}/i$ with k_1 nodes such that $\overline{\mathsf{n}}/(i+j-1) = (\overline{\mathsf{n}}/i)/j \ \forall j \in [1, k_1]$. All the sons of $\overline{\mathsf{n}}$ with index $> i$ change index, which gets augmented of $k_1 - 1$ (i.e., the generic index k becomes $k + k_1 - 1$). This rule imposes that the sons of a *concat*-node must be simple nodes (leaves), OR-nodes or *concat*-nodes with in-degree > 1. In fact, if a *concat*-node (T_1) has a *concat* son (T_2) with indegree $= 1$, we replace T_2 with all its sons, thus increasing the arity of T_1.

We can prove that such normalization rules do not affect the expressiveness of the string graphs. In fact, the denotation of a string graph does not change after the application of one of the four normalization rules.

Lemma 3.6.18 (Soundness of the normalization rules)**.** *Given a normalization rule r_i ($i \in [1, 4]$) and a string graph $\overline{\mathsf{T}}$, suppose that $\overline{\mathsf{T}}'$ is the string graph resulting from the application of r_i to $\overline{\mathsf{T}}$. Then, $\mathbb{D}(\overline{\mathsf{T}}') = \mathbb{D}(\overline{\mathsf{T}})$.*

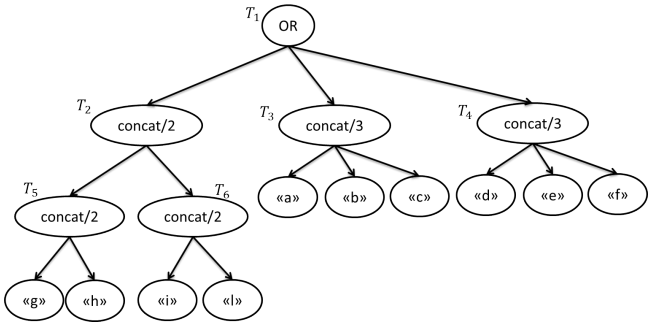
Proof. We refer to [44] for the complete proof of this theorem. □

In Figure 3.9 we can see an example of the normalization process. First of all we apply rule r_3 to T_2 and its sons T_5 and T_6: since T_5 and T_6 are two consecutive sons of a *concat*-node and they are both *concat*-node themselves, we merge them in a single *concat*-node with, as sons, all the sons of T_5 followed by all the sons of T_6. Now we can apply rule r_1 to T_2: since it is a *concat*-node with only one son (T_7), we replace it with such son. Finally, we must apply the principal label restriction because two sons (T_3 and T_4) of the root OR-node have the same label ($concat/3$). We merge such sons in only one, moving the choice (represented by the OR) "downward" the tree (i.e., instead of choosing between the two *concat*-nodes, we choose at the level of their sons, one by one). The string graph in Figure 3.9(d) is the fixpoint of the application of the normalization rules; in fact we cannot apply any more rules to it. Note that the denotation has increased, being $\{ghil, abc, abf, aec, aef, dbc, dbf, dec, def\}$ instead of the original $\{ghil, abc, def\}$; this is caused by the application of the principal label restriction.

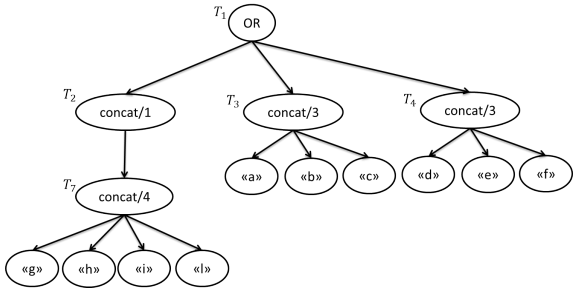
The abstract domain $\overline{\mathcal{SG}}$ is then defined as

$$\overline{\mathcal{SG}} = \overline{\mathsf{NSG}}$$

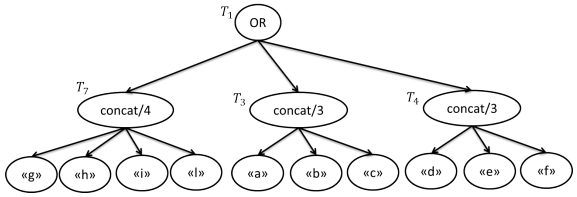
where $\overline{\mathsf{NSG}}$ is the set of all normal string graphs, i.e., compact string graphs which satisfy the principal label restriction and the additional rules 1-4 stated above.



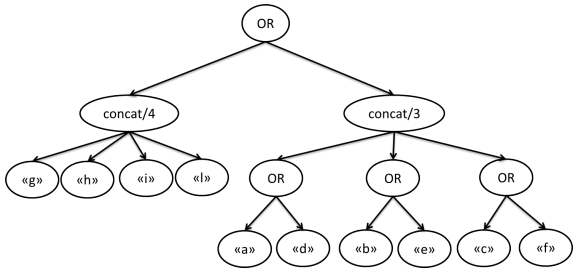
(a) String graph before the application of normalization rules



(b) Application of rule r_3 to T_2 and its sons, T_5 and T_6



(c) Application of rule r_1 to T_2 and its only son T_7



(d) Application of principal label restriction to T_3 and T_4

Figure 3.9: A complete example of string graphs normalization

Partial Order

To define the partial order of the domain we can exploit the algorithm defined in [104] for computing $\leq (\overline{\mathsf{n}}, \overline{\mathsf{m}}, \emptyset)$. The algorithm compares $\mathbb{D}(\overline{\mathsf{n}})$ with $\mathbb{D}(\overline{\mathsf{m}})$ and returns true if $\mathbb{D}(\overline{\mathsf{n}}) \subseteq \mathbb{D}(\overline{\mathsf{m}})$, which is exactly what we need. In particular, the algorithm compares the two nodes in input $(\overline{\mathsf{n}}, \overline{\mathsf{m}})$. In some cases the procedure is recursively called, for example if $\overline{\mathsf{n}}$ and $\overline{\mathsf{m}}$ are both *concat* or OR nodes. Note that the recursive call adds a new edge ($\{\overline{\mathsf{n}}, \overline{\mathsf{m}}\}$) to the third input parameter (a set of edges). If, at the next execution of the procedure ($\leq (\overline{\mathsf{n}'}, \overline{\mathsf{m}'}, \overline{\mathsf{E}})$), the edge $\{\overline{\mathsf{n}'}, \overline{\mathsf{m}'}\}$ is contained in $\overline{\mathsf{E}}$, then the procedure immediately returns true.

The formal definition of the algorithm is the following:

Algorithm 2 Algorithm for node comparison

function $\leq(\overline{\mathsf{n}}, \overline{\mathsf{m}}, \mathsf{S}^C)$
  if $(\overline{\mathsf{n}}, \overline{\mathsf{m}}) \in \mathsf{S}^C$ **then**
    return *true*
  else if $lb(\overline{\mathsf{m}}) = max$ **then**
    return *true*
  else if $lb(\overline{\mathsf{n}}) = lb(\overline{\mathsf{m}}) = concat/k \wedge k > 0$ **then**
    return $\forall i \in [1, k] : \leq (\overline{\mathsf{n}}/i, \overline{\mathsf{m}}/i, \mathsf{S}^C \cup \{(\overline{\mathsf{n}}, \overline{\mathsf{m}})\})$
  else if $lb(\overline{\mathsf{n}}) = lb(\overline{\mathsf{m}}) = OR$ where $k = outdegree(\overline{\mathsf{n}})$ **then**
    return $\forall i \in [1, k] : \leq (\overline{\mathsf{n}}/i, \overline{\mathsf{m}}, \mathsf{S}^C \cup \{(\overline{\mathsf{n}}, \overline{\mathsf{m}})\})$
  else if $lb(\overline{\mathsf{m}}) = OR \wedge \exists m_d \in prnd(\overline{\mathsf{m}}) : lb(\overline{\mathsf{m}_\mathsf{d}}) = lb(\overline{\mathsf{n}})$ **then**
    return $\leq (\overline{\mathsf{n}}, \overline{\mathsf{m}_\mathsf{d}}, \mathsf{S}^C \cup \{(\overline{\mathsf{n}}, \overline{\mathsf{m}})\})$
  else
    return $lb(\overline{\mathsf{n}}) = lb(\overline{\mathsf{m}})$
  end if
end function

Given two string graphs $\overline{\mathsf{T}}_1$ and $\overline{\mathsf{T}}_2$, to check if $\overline{\mathsf{T}}_1 \leq_{\overline{\mathcal{SG}}} \overline{\mathsf{T}}_2$ we will compute $\leq (\overline{\mathsf{n}}_0, \overline{\mathsf{m}}_0, \emptyset)$ where $\overline{\mathsf{n}}_0$ is the root of $\overline{\mathsf{T}}_1$ and $\overline{\mathsf{m}}_0$ is the root of $\overline{\mathsf{T}}_2$. The order is then:

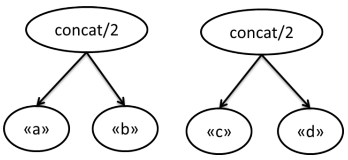
$$\overline{\mathsf{T}}_1 \leq_{\overline{\mathcal{SG}}} \overline{\mathsf{T}}_2 \Leftrightarrow \overline{\mathsf{T}}_1 = \bot_{\overline{\mathcal{SG}}} \vee (\leq (\overline{\mathsf{n}}_0, \overline{\mathsf{m}}_0, \emptyset) : \overline{\mathsf{n}}_0 = \overline{root}(\overline{\mathsf{T}}_1) \wedge \overline{\mathsf{m}}_0 = \overline{root}(\overline{\mathsf{T}}_2))$$

where $\overline{root}(\overline{\mathsf{T}})$ is the root element of the tree defined in $\overline{\mathsf{T}}$.

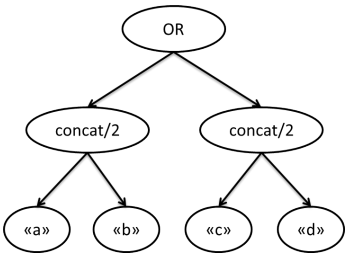
The bottom element $\bot_{\overline{\mathcal{SG}}}$ is a string graph made by one node, a $\bot$-node that represents $\emptyset$. The top element $\top_{\overline{\mathcal{SG}}}$ is a string graph made by only one node, a max-node that represents K^*.

Least Upper Bound and Greatest Lower Bound

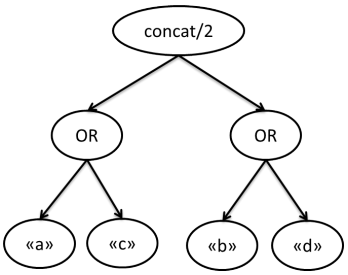
The least upper bound between two string graphs $\overline{\mathsf{T}}_1$ and $\overline{\mathsf{T}}_2$ can be computed by creating a new string graph $\overline{\mathsf{T}}$ whose root is an OR-node whose sons are $\overline{\mathsf{T}}_1$ and $\overline{\mathsf{T}}_2$.



(a) The two string graphs $\overline{\mathsf{T}}_1$ and $\overline{\mathsf{T}}_2$



(b) $\mathsf{OR}(\overline{\mathsf{T}}_1, \overline{\mathsf{T}}_2)$



(c) Result of the least upper bound: $\overline{normStringGraph}(\mathsf{OR}(\overline{\mathsf{T}}_1, \overline{\mathsf{T}}_2))$

Figure 3.10: Computation of the lub

Then we apply the compaction plus normalization algorithm that will transform $\overline{\mathsf{T}}$ in a normal string graph:

$$\bigsqcup_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}}_1, \overline{\mathsf{T}}_2) = \overline{normStringGraph}(\mathsf{OR}(\overline{\mathsf{T}}_1, \overline{\mathsf{T}}_2))$$

An example is depicted in Figure 3.10.

The greatest lower bound operator behaves like the glb between type graphs, which is described in the appendix of [104]. The authors present an algorithm, $intersection(\overline{\mathsf{n}}_1, \overline{\mathsf{n}}_2)$, which computes the type graph $\overline{\mathsf{T}}'$, whose denotation is the intersection of the denotations of the type graphs with roots $\overline{\mathsf{n}}_1$ and $\overline{\mathsf{n}}_2$. Their strategy to deal with this kind of problem is to leave the old type graphs unchanged and to construct the new type graph step by step. The initialization creates the root $\bar{\mathsf{l}}_0$ of $\overline{\mathsf{T}}'$ whose required denotation is defined in terms of the nodes $\overline{\mathsf{n}}_1$ and $\overline{\mathsf{n}}_2$. At this point the root $\bar{\mathsf{l}}_0$ is called an *unexpanded leaf*. They define the function *is* which associates at every step in the construction of $\overline{\mathsf{T}}'$ with each node in $\overline{\mathsf{T}}'$ a set of nodes from the given type graphs such that the second function on the nodes of $\overline{\mathsf{T}}'$, $\mathbb{D}$-is, specifies for each node $\bar{\mathsf{l}}$ of $\overline{\mathsf{T}}'$ its intended denotation.

$$\mathbb{D} - is(\bar{\mathsf{l}}) = \bigcap_{\overline{\mathsf{n}} \in is(\bar{\mathsf{l}})} \mathbb{D}(\overline{\mathsf{n}})$$

Each step extends $\overline{\mathsf{T}}'$ without decreasing the denotation of its nodes. This is done by transforming one of the unexpanded leaves $\bar{\mathsf{l}}$ of $\overline{\mathsf{T}}'$ into a usual node (after the

transformation, $\bar{\mathsf{I}}$ is called a *safe node*), and new unexpanded leaves may be added as sons of $\bar{\mathsf{I}}$. The nodes of $\overline{\mathsf{T}}'$, in each step of its construction, belong either to S^{ul}, the set of unexpanded leaves, or to S^{sn}, the set of safe nodes.

Lemma 3.6.19. *The abstract domain $\overline{\mathcal{SG}}$ is a lattice, $\sqcup_{\overline{\mathcal{SG}}}$ is the least upper bound operator and $\sqcap_{\overline{\mathcal{SG}}}$ is the greatest lower bound operator.*

Proof. Since string graphs are just a particular case of type graphs, we refer to [103, 104] for the complete proof of this lemma. □

Note that the domain is not a complete lattice because the domain is infinite and does not satisfy the ascending chain condition. In fact, it is not even a cpo (directed-complete partial order). To overcome this difficulty, [103] use a finite subdomain by restricting the number of occurrences of a functional symbol on the paths of the graphs. We decided to follow a different approach, based on the widening operator proposed in [147], which we are going to explain after the next paragraph.

Abstraction and Concretization Functions

The concretization function is simply defined by:

$$\gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}}) = \mathbb{D}(\overline{\mathsf{T}})$$

Let the abstraction function $\alpha_{\overline{\mathcal{SG}}}$ be defined by:

$$\alpha_{\overline{\mathcal{SG}}} = \lambda \mathsf{Y}.\, \sqcap_{\overline{\mathcal{SG}}} \{\overline{\mathsf{T}} : \mathsf{Y} \subseteq \gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}})\}$$

Lemma 3.6.20 (Galois connection)**.** *The two functions $\alpha_{\overline{\mathcal{SG}}}$ and $\gamma_{\overline{\mathcal{SG}}}$ form a Galois connection, i.e. $\langle \wp(\mathsf{S}), \subseteq \rangle \underset{\alpha_{\overline{\mathcal{SG}}}}{\overset{\gamma_{\overline{\mathcal{SG}}}}{\leftrightarrows}} \langle \overline{\mathcal{SG}}, \leq_{\overline{\mathcal{SG}}} \rangle$.*

Proof. See [103, 104] for the proof of this assertion.

□

Widening Operator

For the widening operator, we can exploit the one defined in [147]. The widening operator is always applied to an old graph g_{old} and a new graph g_{new} to produce a new graph g_{res}. The main idea behind the widening operator of [147] for type graphs is to consider two graphs:

$$g_0 = g_{old} \text{ and } g_n = (g_{old} \sqcup g_{new})$$

and exploit the topology of the graphs to guess where g_n is growing compared to g_0. The key notion is the concept of topological clash which occurs in situations where: (i) an OR-node v_0 in g_0 corresponds to an OR-node v_n in g_n where $prlb(v_0) \neq$

Table 3.8: The abstract semantics of $\overline{\mathcal{SG}}$

$$\overline{\mathbb{S}_{\mathcal{SG}}}[\![\texttt{new String(str)}]\!]() = \mathsf{concat/k}\{\texttt{str}[\mathsf{i}] : \mathsf{i} \in [0, \mathsf{k}-1]\}$$
$$\overline{\mathbb{S}_{\mathcal{SG}}}[\![\texttt{concat}]\!](\bar{\mathsf{t}}_1, \bar{\mathsf{t}}_2) = \overline{normStringGraph}(\mathsf{concat/2}\{\bar{\mathsf{t}}_1, \bar{\mathsf{t}}_2\})$$
$$\overline{\mathbb{S}_{\mathcal{SG}}}[\![\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}]\!](\bar{\mathsf{t}}) = \begin{cases} \overline{\mathsf{res}} & \text{if } \overline{root}(\bar{\mathsf{t}}) = \mathsf{concat/k} \wedge \forall \mathsf{i} \in [0, \mathsf{e}-1] : \overline{lb}(\overline{root}(\bar{\mathsf{t}})/\mathsf{i}) \in \mathsf{K} \\ \top_{\overline{\mathcal{SG}}} & \text{otherwise} \end{cases}$$
where $\overline{\mathsf{res}} = \mathsf{concat/(e-b)}\{(\overline{root}(\bar{\mathsf{t}})/\mathsf{i}) : \mathsf{i} \in [\mathsf{b}, \mathsf{e}-1]\}$
$$\overline{\mathbb{B}_{\mathcal{SG}}}[\![\texttt{contains}_{\mathsf{c}}]\!](\bar{\mathsf{t}}) = \begin{cases} \mathsf{true} & \text{if } \exists \overline{\mathsf{m}} \in \bar{\mathsf{t}} : \overline{\mathsf{m}} = \mathsf{concat/k} \wedge OR \notin \overline{path}(\overline{\mathsf{root}}, \overline{\mathsf{m}}) \wedge \\ & \qquad \exists \mathsf{i} \in [0, \mathsf{k}-1] : \overline{lb}(\overline{\mathsf{m}}/\mathsf{i}) = \mathsf{c} \\ \mathsf{false} & \text{if } \nexists \overline{\mathsf{n}} \in \bar{\mathsf{t}} : \overline{lb}(\overline{\mathsf{n}}) = \mathsf{max} \vee \overline{lb}(\overline{\mathsf{n}}) = \mathsf{c} \\ \top_{\mathsf{B}} & \text{otherwise} \end{cases}$$

$prlb(v_n)$, or (ii) an OR-node v_0 in g_0 corresponds to an OR-node v_n in g_n where $depth(v_0) < depth(v_n)$. In these cases the widening operator tries to prevent the graph from growing by introducing a cycle in g_n. Given a clash (v_0, v_n), the widening searches for an ancestor v_a to v_n such that $prlb(v_n) \subseteq prlb(v_a)$. If such an ancestor is found and if $v_a \geq v_n$, a cycle can be introduced.

When no ancestor with a suitable *prlb*-set can be found, the widening operator simply allows the graph to grow. Termination will be guaranteed because this growth necessarily adds along the branch of a *prlb*-set which is not a subset of any existing *prlb*-set in the branch. This case happens frequently in early iterations of the fixpoint. Letting the graph grow in this case is of great importance to recover the structure of the type in its entirety.

The last case to consider appears when there is an ancestor v_a with a suitable *prlb*-set, but $v_a \geq v_n$ is false. In this case, introducing a cycle would produce a graph g_{res} whose denotation may not include the denotation of g_n, and hence the widening cannot perform cycle introduction. Instead, the operation replaces v_a by a new OR-node which is an upper bound to v_a and v_n but decreases the overall size of the graph. The widening is then applied again on the resulting graph.

In conclusion, such widening operator can be viewed as a sequence of transformations on g_n which are of two types: cycle introduction and node replacement, until no more topological clashes can be resolved.

Semantics

Table 3.8 defines the abstract semantics on $\overline{\mathcal{SG}}$.

Let us discuss in detail the semantics of each operator:

- The evaluation of a string (made by k characters) returns a *concat*-node with all the characters that compose the string as sons.

- When we concatenate two strings, we create a new string graph, whose root is a *concat*-node with two sons. The two sons are the roots of the two input abstract values. Then we need to normalize the result, to be sure that it is a normal string graph.

- The semantics of $\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}$ returns a precise value only if the root is a *concat*-node whose first e sons are characters. In fact, if the root of the string graph is a *concat*-node and its first `endIndex` sons are simple nodes (leaves), then we can return the exact substring. Otherwise, we return $\top_{\overline{\mathcal{SG}}}$.

- The semantics of $\texttt{contains}_{\mathsf{c}}$ returns false iff we are sure that the character c does not appear in the string, that is, there is no simple node labelled with such character and there is no *max*-node. We can return true iff we find in the string graph a *concat*-node $\overline{\mathsf{m}}$ containing a son with label c, and the path from the root to $\overline{\mathsf{m}}$ does not contain any *OR*-node. Otherwise, we will have to return $\top_{\overline{\mathcal{SG}}}$.

We now prove the soundness of the abstract operations defined above.

Theorem 3.6.21 (Soundness of the abstract semantics)**.** $\overline{\mathbb{S}_{\overline{\mathcal{SG}}}}$ *and* $\overline{\mathbb{B}_{\overline{\mathcal{SG}}}}$ *are a sound overapproximation of* $\mathbb{S}$ *and* $\mathbb{B}$*, respectively. Formally,* $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathbb{S}_{\mathcal{SG}}}[\![\mathsf{s}]\!](\overline{\mathsf{T}})) \supseteq \{\mathbb{S}[\![\mathsf{s}]\!](\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}})\}$ *and* $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathbb{B}_{\mathcal{SG}}}[\![\mathsf{s}]\!](\overline{\mathsf{T}})) \geq_{\mathsf{B}} \{\mathbb{B}[\![\mathsf{s}]\!](\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}})\}$.

Proof. We prove the soundness separately for each operator.

- $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathbb{S}_{\mathcal{SG}}}[\![\texttt{new String(str)}]\!]()) \supseteq \{\mathbb{S}[\![\texttt{new String(str)}]\!]()\}$ follows immediately from the definition of $\overline{\mathbb{S}_{\mathcal{SG}}}[\![\texttt{new String(str)}]\!]()$ and of $\gamma_{\overline{\mathcal{SG}}}$.

- Consider the binary operator `concat`. Let $\overline{\mathsf{T}}_1$ and $\overline{\mathsf{T}}_2$ be two string graphs. $\{\mathbb{S}[\![\texttt{concat}]\!](\mathsf{c}_1, \mathsf{c}_2) : \mathsf{c}_1 \in \gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}}_1) \wedge \mathsf{c}_2 \in \gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}}_2)\}$ contains strings which are the concatenation of one string from $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}}_1)$ and one from $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}}_2)$ by definition of $\mathbb{S}$. Let s be one of these strings. s belongs to $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathbb{S}_{\mathcal{SG}}}[\![\texttt{concat}]\!](\overline{\mathsf{T}}_1, \overline{\mathsf{T}}_2))$, since $\overline{\mathbb{S}_{\mathcal{SG}}}[\![\texttt{concat}]\!](\overline{\mathsf{T}}_1, \overline{\mathsf{T}}_2)$ produces a new string graph which has a concat-node as root and the two original string graphs as sons [5], and the concretization of such string graph is $\{concat(t_1, t_2) : t_i \text{ is finite } \wedge\, t_i \in \mathbb{D}(\overline{\mathsf{root}}/i)\ \forall i \in [1, 2]\}$ by definition of $\gamma_{\mathcal{SG}}$.

- Consider the unary operator $\texttt{substring}^{\mathsf{e}}_{\mathsf{b}}$ and let $\overline{\mathsf{T}}$ be a string graph. Consider the two following cases:

  - if $\overline{root}(\overline{\mathsf{T}}) = \mathsf{concat}/\mathsf{k} \wedge \forall \mathsf{i} \in [0, \mathsf{e} - 1] : \overline{lb}(\overline{root}(\overline{\mathsf{T}})/\mathsf{i}) \in \mathsf{K}$, then the root of $\overline{\mathsf{T}}$ is a concat-node and its first e sons are all simple characters. In this case, all strings belonging to $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}})$ will start with the

[5] The string graph is also normalized, but the normalization can only increase the concretization of an abstract state, thus we can ignore it: if a string belongs to the concretization of a not-normal string graph, it will surely belong also to its normalized version.

concatenation of these characters, by definition of $\mathbb{D}(\overline{\mathsf{T}})$. This prefix is also certainly longer than e characters. Then, a string belonging to $\{\mathbb{S}[\![\mathtt{substring}_{\mathsf{b}}^{\mathsf{e}}]\!](\mathsf{c}) : \mathsf{c} \in \gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}})\}$ is composed by the concatenation of all the characters of the nodes from $\overline{root}(\overline{\mathsf{T}})/\mathsf{b}$ to $\overline{root}(\overline{\mathsf{T}})/(\mathsf{e}-1)$ by definition of $\mathbb{S}$. This corresponds exactly to $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathbb{S}_{\mathcal{SG}}}[\![\mathtt{substring}_{\mathsf{b}}^{\mathsf{e}}]\!](\overline{\mathsf{T}}))$, since the abstract semantics applied to $\overline{\mathsf{T}}$ produces a string graph whose root is a concat-node and which sons are the nodes from $\overline{root}(\overline{\mathsf{T}})/\mathsf{b}$ to $\overline{root}(\overline{\mathsf{T}})/(\mathsf{e}-1)$.

- otherwise, the abstract semantics returns $\top_{\overline{\mathcal{SG}}}$, that approximates any possible value of the concrete semantics.

- Consider the unary operator $\mathtt{contains}_{\mathsf{c}}$ and let $\overline{\mathsf{T}}$ be a string graph. Regarding the character c, we have three cases:
  - if $\exists\overline{\mathsf{m}} \in \overline{\mathsf{T}} : \overline{\mathsf{m}} = \mathsf{concat}/\mathsf{k} \wedge OR \notin \overline{path}(\overline{\mathsf{root}}, \overline{\mathsf{m}}) \wedge \exists \mathsf{i} : \overline{lb}(\overline{\mathsf{m}}/\mathsf{i}) = \mathsf{c}$, this means that there exists a concat-node in $\overline{\mathsf{T}}$ that (i) has a son with the character c as label, and (ii) the path from the root to such node does not contain OR nodes. Then, by definition of $\mathbb{D}(\overline{\mathsf{T}})$, the character c belongs to all strings in $\gamma_{\overline{\mathcal{SG}}}(\overline{\mathsf{T}})$, and then the result of the concrete semantics is always $\mathtt{true}$. Since $\overline{\mathbb{B}_{\mathcal{SG}}}[\![\mathtt{contains}_{\mathsf{c}}]\!](\overline{\mathsf{T}}) = \mathtt{true}$ by the definition of the abstract semantics, the abstract semantics soundly approximates the concrete semantics.
  - if $\nexists\overline{\mathsf{n}} \in \overline{\mathsf{T}} : \overline{lb}(\overline{\mathsf{n}}) = \mathsf{max} \vee \overline{lb}(\overline{\mathsf{n}}) = \mathsf{c}$, this means that no node of the string graph has label c or max. Then, the character c cannot be contained in any of the concrete strings corresponding to the abstract state $\overline{\mathsf{T}}$ and for this reason the concrete semantics always returns $\mathtt{false}$. Since $\overline{\mathbb{B}_{\mathcal{SG}}}[\![\mathtt{contains}_{\mathsf{c}}]\!](\overline{\mathsf{T}}) = \mathtt{false}$ by the definition of the abstract semantics, the abstract semantics soundly approximates the concrete semantics.
  - otherwise, the abstract semantics on $\overline{\mathsf{T}}$ returns $\top_{\mathsf{B}}$, and this soundly approximates any possible result of the concrete semantics.

□

Case studies

The results of the analysis of the two case studies through string graphs are depicted in Figures 3.11(a) and 3.11(b). For sake of simplicity, we adopt the notation $\mathsf{concat}[\mathsf{s}]$ to indicate a string graph with a *concat* node whose sons are all the characters of the string s. The symbol + represents, as usual, string concatenation, while ; is used to separate different sons of a node.

For `prog1`, at line 1 we represent `query` with a string graph made by a *concat*-node with all the characters of s_1 as sons. The `l` variable (line 3) corresponds simply to a *max*-node, since we do not know its value. At line 4 we concatenate the current

#I	Var	$\overline{\mathcal{SG}}$
1	query	concat[s_1]
3	l	max
4	query	concat[$s_1 + s_2$; max; s_3]
5	query	$\overline{SG}_1$ = OR[concat[s_1]; concat[$s_1 + s_2$; max; s_3]]
6	per	concat[s_4]
8	query	concat[$\overline{SG}_1$; concat[$s_5 + s_4 + s_6$]]

(a) Analysis of `prog1`

#I	Var	$\overline{\mathcal{SG}}$
1	x	concat[“a”]
3	x	OR[“a”; concat[“0”; “a”; “1”]]
4	x	OR_1[“a”; concat[“0”; OR_1; “1”]]

(b) Analysis of `prog2`

Figure 3.11: The results of $\overline{\mathcal{SG}}$

value of `query` with s_2, `l` and s_3: the abstract value of `query` then is a *concat* node with, as sons, all the characters of s_1, followed by all the characters of s_2, followed by a *max*-node, followed by all the characters of s_3. Since the value of `l` is unknown, we must compute the least upper bound between the values of `query` after line 1 and 4. We obtain a string graph made by an *OR*-node with the two input string graphs as sons. Then, at line 6 we associate the variable `per` to the abstraction of s_4. Finally, at line 8 we concatenate `query` to s_5, `per` and s_6: we obtain a string graph which is made by a *concat* node as root, and, as sons, the string graph associated to `query` at line 5 and then all the characters of s_5, s_4 and s_6, one after the other. The resulting string graph for `query` represents exactly the two possible outcomes of the procedure.

For `prog2`, after line 1 we represent `x` with a *concat* node with just one son, containing an `a` character. After the first iteration of the loop, line 3, the abstract value associated to `x` is a *concat* node with three sons, `0`, `a` and `1`. The least upper bound between the two abstract values (before entering the loop and after the first iteration) is an *OR*-node with two sons: one is an `a` character, the other is the value of `x` after the first iteration. Since we have not reached convergence, we must compute the value of after the second iteration also. In this domain, though, computing the least upper bound of the values of the first n iterations is not sufficient to reach convergence, since we always add some new branch to the string graph. We need to use the widening operator and the result (after reaching convergence) is as follows: OR_1[“a”; concat[“0”; OR_1; “1”]]. The string graph root is an *OR*-node with two sons: an `a` character and a *concat* node with three sons. The first and last sons

Table 3.9: Comparison of the abstract domains results

Abstract domain	prog1	prog2
$\overline{\mathcal{CI}}$	$(\pi_1(\alpha_{\mathcal{CI}}(\mathsf{s}_1)) \cup \pi_1(\alpha_{\mathcal{CI}}(\mathsf{s}_4)) \cup$ $\pi_1(\alpha_{\mathcal{CI}}(\mathsf{s}_5)) \cup \pi_1(\alpha_{\mathcal{CI}}(\mathsf{s}_6)), \mathsf{K})$	$(\{a\}, \{0, a, 1\})$
$\overline{\mathcal{PR}}$	$\overline{\mathsf{s}_1}$	$\top$
$\overline{\mathcal{SU}}$	$\overline{\mathsf{s}_6}$	$\top$
$\overline{\mathcal{BR}}$	$[\{\mathsf{s}_1\}]^{1,1}[\{\mathsf{s}_2\}]^{0,1}\top_{\overline{\mathcal{B}}}[\{\mathsf{s}_3\}]^{0,1}[\{\mathsf{s}_5\}]^{1,1}[\{\mathsf{s}_4\}]^{1,1}[\{\mathsf{s}_6\}]^{1,1}$	$[\{\text{“0”}\}]^{0,+\infty}[\{\text{“}a\text{”}\}]^{1,1}[\{\text{“1”}\}]^{0,+\infty}$
$\overline{\mathcal{SG}}$	$\mathsf{concat}[\overline{\mathsf{SG}_1}; \mathsf{concat}[\mathsf{s}_5 + \mathsf{s}_4 + \mathsf{s}_6]]$ where $\overline{\mathsf{SG}_1} = \mathsf{OR}[\mathsf{concat}[\mathsf{s}_1]; \mathsf{concat}[\mathsf{s}_1 + \mathsf{s}_2; \mathsf{max}; \mathsf{s}_3]]$	$\mathsf{OR}_1[\text{“}a\text{”}; \mathsf{concat}[\text{“0”}; \mathsf{OR}_1; \text{“1”}]]$

are, respectively, a `0` character and a `1` character. The second son is, instead, the root *OR*-node, thanks to the use of a backward arc. The resulting string graph for `x` represents *exactly* all the concrete possible values of `x`. Note that the resulting string graph contains a backward arc to allow the repetition of the pattern $0^n \ldots 1^n$.

This abstract domain is the most precise domain for the analysis of both case studies: it tracks information similarly to $\overline{\mathcal{BR}}$ domain, but its lub and widening operators are slightly more accurate.

3.6.5 Discussion: Relations Between the Five Domains

The abstract domains we introduced track different types of information. In Table 3.9 we recap the result obtained by each domain when applied to the two case studies of Section 3.2. It is immediate to see that $\overline{\mathcal{CI}}$, $\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$ are less precise than $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$, even though they are still able to track a limited amount of information.

We are now going to elaborate in more detail about the relations between the five domains. At the beginning of Section 3.6, we pointed out that a string is essentially characterized by two pieces of information: the *characters* contained in the string, and their *position* inside the string. $\overline{\mathcal{CI}}$ domain considers only character inclusion and completely disregards the order. $\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$ domains consider also the order, but limiting themselves to the initial/final segment of the string, and in the same way they collect only partial information about character inclusion. $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$, instead, track both inclusion and order along the string.

In [44] we studied these relationships in details: to understand the degree of precision of each representation and how we can go from one representation to another, we defined pairs of functions (abstraction and concretization) from domain to domain, and showed that $\overline{\mathcal{CI}}$, $\overline{\mathcal{PR}}$ and $\overline{\mathcal{SU}}$ are more abstract (i.e., less precise) than both $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$. We report here the definition of such functions for each pair of domains (excluding $\overline{\mathcal{SU}}$, for which the definitions are symmetrical to those

of $\overline{\mathcal{PR}}$); for further explanations or examples, see [44].

$\overline{\mathcal{CI}}$ and $\overline{\mathcal{SG}}$

We argue that string graphs are more precise than the pair of sets of certainly contained and maybe contained characters, that is, $\overline{\mathcal{CI}}$ domain is more abstract than $\overline{\mathcal{SG}}$ domain: $\overline{\mathcal{SG}} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \overline{\mathcal{CI}}$. The input of the abstraction function is a string graph (i.e., the abstraction of a set of strings) and its output is a rougher abstraction of such set of strings, that is, an element of $\overline{\mathcal{CI}}$. The input of the concretization function is an element of $\overline{\mathcal{CI}}$ and its output is a set of string graphs.

Let $\overline{\mathsf{T}}$ be a string graph, and let $L(\overline{\mathsf{T}})$ be the set of nodes (leaves) of $\overline{\mathsf{T}}$ which label is a character: $L(\overline{\mathsf{T}}) = \{l : l \in \overline{\mathsf{T}} \wedge \overline{lb}(l) \in \mathsf{K}\}$. Let $L_c(\overline{\mathsf{T}})$ be the set of characters associated to the nodes of $L(\overline{\mathsf{T}})$: $L_c(\overline{\mathsf{T}}) = \{c : c \in \mathsf{K} \wedge \exists l \in L(\overline{\mathsf{T}}) : c = \overline{lb}(l)\}$. Finally, let $L_{sure}(\overline{\mathsf{T}})$ be the set of characters associated to the leaves which path from the root to them does not contain any OR-node: $L_{sure}(\overline{\mathsf{T}}) = \{c : c \in \mathsf{K} \wedge \exists l \in L(\overline{\mathsf{T}}) : (c = \overline{lb}(l) \wedge OR \notin \overline{path}(root(\overline{\mathsf{T}}), l))\}$. Note that $L_{sure}(\overline{\mathsf{T}}) \subseteq L_c(\overline{\mathsf{T}})$.

Then, the abstraction function is defined as follows:

$$\alpha(\overline{\mathsf{T}}) = \begin{cases} (L_{sure}(\overline{\mathsf{T}}), \mathsf{K}) & \text{if } \exists n \in \overline{\mathsf{T}} : \overline{lb}(n) = max \\ (L_{sure}(\overline{\mathsf{T}}), L_c(\overline{\mathsf{T}})) & \text{otherwise} \end{cases}$$

while the concretization function is defined as follows:

$$\gamma((\overline{\mathsf{CC}}, \overline{\mathsf{MC}})) = \begin{cases} \{\overline{\mathsf{T}} : \overline{\mathsf{CC}} \subseteq L_{sure}(\overline{\mathsf{T}}) \wedge L_c(\overline{\mathsf{T}}) \subseteq \overline{\mathsf{MC}} \wedge max \notin \overline{\mathsf{T}}\} & \text{if } \overline{\mathsf{MC}} \neq \mathsf{K} \\ \{\overline{\mathsf{T}} : \overline{\mathsf{CC}} \subseteq L_{sure}(\overline{\mathsf{T}})\} & \text{otherwise} \end{cases}$$

$\overline{\mathcal{CI}}$ and $\overline{\mathcal{BR}}$

We argue that bricks are more precise than the pair of sets of certainly contained and maybe contained characters, that is, $\overline{\mathcal{CI}}$ domain is more abstract than $\overline{\mathcal{BR}}$ domain: $\overline{\mathcal{BR}} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \overline{\mathcal{CI}}$. The input of the abstraction function is a list of bricks and its output is an element of $\overline{\mathcal{CI}}$. The input of the concretization function is a pair of sets of characters and its output is a set of lists of bricks.

Let $\overline{\mathsf{L}}$ be a list of bricks. We define $CS(\overline{\mathsf{L}}) = \{s : \exists \overline{\mathsf{b}} \in \overline{\mathsf{L}} : (\overline{\mathsf{b}} = [\{s\}]^{1,1})\}$ as the set of the strings contained in bricks with indices $min = max = 1$ and with only one string in the brick set: these strings will certainly be contained in each of the concrete strings represented by $\overline{\mathsf{L}}$. We also define $US(\overline{\mathsf{L}}) = \{s : \exists \overline{\mathsf{b}} \in \overline{\mathsf{L}} : (\overline{\mathsf{b}} = [S]^{m,M} \wedge s \in S)\}$ as the set of all the strings contained in bricks of $\overline{\mathsf{L}}$: all these strings *could* be contained in each of the concrete strings represented by $\overline{\mathsf{L}}$. Finally, we define their corresponding sets of characters: $CC(\overline{\mathsf{L}}) = \{c : \exists s \in CS(\overline{\mathsf{L}}) : s.contains(c)\}$ and $UC(\overline{\mathsf{L}}) = \{c : \exists s \in US(\overline{\mathsf{L}}) : s.contains(c)\}$. These two sets contain, respectively, all the characters which appear in strings of $CS(\overline{\mathsf{L}})$ and all the characters which appear in strings of $US(\overline{\mathsf{L}})$.

The abstraction function is then defined as follows:

$$\alpha(\overline{\mathsf{L}}) = \begin{cases} (CC(\overline{\mathsf{L}}), \mathsf{K}) & \text{if } \top_{\overline{\mathcal{B}}} \in \overline{\mathsf{L}} \\ (CC(\overline{\mathsf{L}}), UC(\overline{\mathsf{L}})) & \text{otherwise} \end{cases}$$

The concretization function is defined as follows:

$$\gamma((\overline{\mathsf{CC}}, \overline{\mathsf{MC}})) = \begin{cases} \{\overline{\mathsf{L}} : \overline{\mathsf{CC}} \subseteq CC(\overline{\mathsf{L}}) \wedge UC(\overline{\mathsf{L}}) \subseteq \overline{\mathsf{MC}} \wedge \top_{\overline{\mathcal{BR}}} \notin \overline{\mathsf{L}}\} & \text{if } \overline{\mathsf{MC}} \neq \mathsf{K} \\ \{\overline{\mathsf{L}} : \overline{\mathsf{CC}} \subseteq CC(\overline{\mathsf{L}})\} & \text{otherwise} \end{cases}$$

$\overline{\mathcal{PR}}$ and $\overline{\mathcal{SG}}$

We argue that string graphs are more precise than the prefix domain, that is, $\overline{\mathcal{PR}}$ domain is more abstract than $\overline{\mathcal{SG}}$ domain: $\overline{\mathcal{SG}} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \overline{\mathcal{PR}}$. The input of the abstraction function is a string graph and its output is an element of $\overline{\mathcal{PR}}$. The input of the concretization function is a prefix and its output is a set of string graphs.

Let $\overline{\mathsf{T}}$ be a string graph, and $r = root(\overline{\mathsf{T}})$ be its root. The abstraction function is defined as follows:

$$\alpha(\overline{\mathsf{T}}) = \begin{cases} \overline{\mathsf{p}} & \text{if } \overline{lb}(r) = concat/k \wedge \exists k_1 > 0 : (\overline{lb}(r/i) \in \mathsf{K}\ \forall i \in [1, k_1] \wedge \overline{lb}(r/(k_1+1)) \notin \mathsf{K}) \\ * & \text{otherwise} \end{cases}$$

where $\overline{\mathsf{p}}$ is a sequence of k_1 characters, such that $\forall i \in [0, k_1 - 1] : \overline{\mathsf{p}}[i] = \overline{lb}(r/(i+1))$.

The concretization function is defined as follows (supposing again that $r = root(\overline{\mathsf{T}})$):

$$\gamma(\overline{\mathsf{p}}) = \{\overline{\mathsf{T}} : (\overline{lb}(r) = concat/k \wedge (k \geq k_p) \wedge (\forall i \in [1, k_p] : \overline{lb}(r/i) = \overline{\mathsf{p}}[i-1]))\}$$

where k_p is the length of the abstract prefix $\overline{\mathsf{p}}$.

$\overline{\mathcal{PR}}$ and $\overline{\mathcal{BR}}$

We argue that bricks are more precise than the prefix domain, that is, $\overline{\mathcal{PR}}$ domain is more abstract than $\overline{\mathcal{BR}}$ domain: $\overline{\mathcal{BR}} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \overline{\mathcal{PR}}$. The input of the abstraction function is a list of bricks and its output is an element of $\overline{\mathcal{PR}}$. The input of the concretization function is a prefix and its output is a set of lists of bricks.

The abstraction function is defined as follows:

$$\alpha(\overline{\mathsf{L}}) = \begin{cases} \overline{\mathsf{p}} & \text{if } \overline{\mathsf{L}}[0] = [S]^{(1,1)} \\ * & \text{otherwise} \end{cases}$$

where $\overline{\mathsf{p}}$ is the longest common prefix between all the strings contained in S.

The concretization function is defined as follows:

$$\gamma(\overline{\mathsf{p}}) = \{\overline{\mathsf{L}} : len(\overline{\mathsf{L}}) \geq 1 \wedge \overline{\mathsf{L}}[0] = [S]^{(1,1)} \wedge \forall s \in S : isPrefix(\overline{\mathsf{p}}, s)\}$$

where $isPrefix$ is a helper function which, given two strings in input, returns true if the first string is a prefix of the second one.

$\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$

In the case of $\overline{\mathcal{BR}}$ versus $\overline{\mathcal{SG}}$, the comparison is more complex, since they exploit very different data structures. For example, $\overline{\mathcal{SG}}$ has OR-nodes, while $\overline{\mathcal{BR}}$ can only trace alternatives inside bricks but not outside (like: "these three bricks *or* these other two"). From this perspective, $\overline{\mathcal{SG}}$ is more precise than $\overline{\mathcal{BR}}$. Another important difference is that $\overline{\mathcal{SG}}$ has backward arcs which allow repetitions of patterns, but they can be traversed how many times we want. With $\overline{\mathcal{BR}}$, instead, we can indicate exactly how many times a certain pattern should be repeated (through the range of bricks). This makes $\overline{\mathcal{BR}}$ more expressive than $\overline{\mathcal{SG}}$ in that respect. So, these domains are not directly comparable.

Domains hierarchy

Another way to evaluate the different precision of our domains is to consider the definition of precision gave in Chapter 2, which compares the results of abstracting a concrete element and then concretizing the abstract result. Concrete elements are made by sets of strings: however, in order to show more immediate and intuitive results, we will consider here only a single string:

- domain $\overline{\mathcal{CI}}$: the abstraction of a string s returns the pair of sets $(char(\mathsf{s}), char(\mathsf{s}))$. The concretization of such abstract element, $\gamma_{\overline{\mathcal{CI}}}(\alpha_{\overline{\mathcal{CI}}}(\mathsf{s}))$, is the set of all strings composed by characters in $char(\mathsf{s})$ (and with at least one occurrence of each different character).

- domain $\overline{\mathcal{PR}}$: the abstraction of a string s returns the same string as prefix (s). The concretization of such abstract element, $\gamma_{\overline{\mathcal{PR}}}(\alpha_{\overline{\mathcal{PR}}}(\mathsf{s}))$, is the set of all strings starting with s.

- domain $\overline{\mathcal{SU}}$: the abstraction of a string s returns the same string as suffix (s). The concretization of such abstract element, $\gamma_{\overline{\mathcal{SU}}}(\alpha_{\overline{\mathcal{SU}}}(\mathsf{s}))$, is the set of all strings ending with s.

- domain $\overline{\mathcal{BR}}$: the abstraction of a string s returns a list made by only one brick, $[\{\mathsf{s}\}]^{1,1}$. The concretization of such abstract element, $\gamma_{\overline{\mathcal{BR}}}(\alpha_{\overline{\mathcal{BR}}}(\mathsf{s}))$, is the singleton $\{\mathsf{s}\}$.

- domain $\overline{\mathcal{SG}}$: the abstraction of a string s (of length k) returns the string graph defined as $\mathsf{concat/k}\{\mathsf{s}[\mathsf{i}] : \mathsf{i} \in [0, \mathsf{k}-1]\}$. The concretization of such abstract element, $\gamma_{\overline{\mathcal{SG}}}(\alpha_{\overline{\mathcal{SG}}}(\mathsf{s}))$, is the singleton $\{\mathsf{s}\}$.

It is immediate to see that:

- both $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$ are more precise than $\overline{\mathcal{CI}}$: $\gamma_{\overline{\mathcal{BR}}}(\alpha_{\overline{\mathcal{BR}}}(\mathsf{s})) \subseteq \gamma_{\overline{\mathcal{CI}}}(\alpha_{\overline{\mathcal{CI}}}(\mathsf{s}))$ and $\gamma_{\overline{\mathcal{SG}}}(\alpha_{\overline{\mathcal{SG}}}(\mathsf{s})) \subseteq \gamma_{\overline{\mathcal{CI}}}(\alpha_{\overline{\mathcal{CI}}}(\mathsf{s}))$

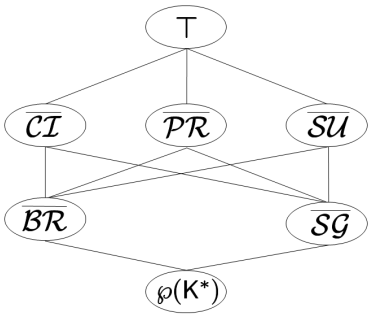


Figure 3.12: The hierarchy of abstract domains

- both $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$ are more precise than $\overline{\mathcal{PR}}$: $\gamma_{\overline{\mathcal{BR}}}(\alpha_{\overline{\mathcal{BR}}}(\mathsf{s})) \subseteq \gamma_{\overline{\mathcal{PR}}}(\alpha_{\overline{\mathcal{PR}}}(\mathsf{s}))$ and $\gamma_{\overline{\mathcal{SG}}}(\alpha_{\overline{\mathcal{SG}}}(\mathsf{s})) \subseteq \gamma_{\overline{\mathcal{PR}}}(\alpha_{\overline{\mathcal{PR}}}(\mathsf{s}))$
- both $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$ are more precise than $\overline{\mathcal{SU}}$: $\gamma_{\overline{\mathcal{BR}}}(\alpha_{\overline{\mathcal{BR}}}(\mathsf{s})) \subseteq \gamma_{\overline{\mathcal{SU}}}(\alpha_{\overline{\mathcal{SU}}}(\mathsf{s}))$ and $\gamma_{\overline{\mathcal{SG}}}(\alpha_{\overline{\mathcal{SG}}}(\mathsf{s})) \subseteq \gamma_{\overline{\mathcal{SU}}}(\alpha_{\overline{\mathcal{SU}}}(\mathsf{s}))$

Combining these informal results, we obtain the lattice depicted in Figure 3.12, where the upper domains are more approximated. We denote by $\top$ the abstract domain that does not track any information about string values, and by $\wp(\mathsf{K}^*)$ the (naïve and uncomputable) domain that tracks all the possible values of strings we can have.

In conclusion, the first three domains ($\overline{\mathcal{CI}}$, $\overline{\mathcal{PR}}$, $\overline{\mathcal{SU}}$) are not so precise but the complexity is kept linear, whereas the other domains ($\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$) are more demanding (though in the practice complexity is still kept polynomial) but also more precise.

3.7 Experimental Results

We developed a preliminary implementation of all the abstract domains formalized in Section 3.6 in Sample. Remember from Chapter 1 that Sample is a generic analyzer of object-oriented programs which is parametric on a value (e.g., numerical) domain, a heap abstraction, and on the property of interest or an engine to infer annotation (e.g., pre- and post- conditions). The string analyses are plugged as value analyses. Notice that the results we have reported on the two case studies introduced in Section 3.2 (`prog1`, `prog2`) are obtained through this implementation.

We discuss now the application of our analysis to two other case studies, `prog3` and `prog4`. Figure 3.13 reports their code. In particular, `prog3` is an interesting example cited in [35] as motivation for their work. The code creates a SQL query by first assigning a constant value ("SELECT * FROM address") to the string `q` and then concatenating it with another constant string ("WHERE studentId="), but

```
1 var q : String = "SELECT * FROM address";
2 if (i != 0)
3         q = q + "WHERE studentId="
```

(a) The first additional case study, `prog3`

```
1  var sql1 : String = "";
2  var sql2 : String = "";
3  sql1 = "SELECT";
4  sql1 = sql1 + " " + l;
5  sql1 = sql1 + " " + "FROM";
6  sql1 = sql1 + " " + l;
7  sql2 = "UPDATE";
8  sql2 = sql2 + " " + l;
9  sql2 = sql2 + " " + "SET";
10 sql2 = sql2 + " " + l + " = " + l;
```

(b) The second additional case study, `prog4`

Figure 3.13: Two additional case studies

only if some condition (unknown at compile time) holds. In Table 3.10 we report the results of the analysis of this case study with all our five domains:

- with $\overline{\mathcal{CI}}$ we get that: (i) all the characters of the string "SELECT * FROM address" will *certainly* be contained in `q` at the end of the program, and (ii) all the characters of the string "WHERE studentId=" (in addition to those of "SELECT * FROM address") *could* be contained in `q` at the end of the program;
- $\overline{\mathcal{PR}}$ tells us that `q` will certainly start with the string "SELECT * FROM address";
- $\overline{\mathcal{SU}}$ is not able to give us any information, since its result is $\top_{\overline{\mathcal{SU}}}$;
- $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$, instead, infer the same information (even though encoded in different ways), which also corresponds *exactly* to the outcome of the program: `q` could have value "SELECT * FROM addressWHERE studentId=" or "SELECT * FROM address".

From the result given by $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$ we can discover the bug hidden in the program: there is a space missing between "address" and "WHERE", which will make the SQL query to sometimes fail at runtime (when `i` $\neq$ `0`). Note that the resulting bricks list is made by just two bricks, while the resulting string graph is composed by 61 nodes

Table 3.10: Results of `prog3`

Abstract domain	**Value of `q`**
$\overline{\mathcal{CI}}$	$(\{E, e, s, *, T, F, a, M, , L, C, r, R, O, S, d\}, \{E, e, s, *, n, T, =$ $, t, u, F, a, M, I, , L, C, H, W, r, R, O, S, d\})$
$\overline{\mathcal{PR}}$	"SELECT * FROM ADDRESS"
$\overline{\mathcal{SU}}$	$\top_{\overline{\mathcal{SU}}}$
$\overline{\mathcal{BR}}$	[{"SELECT * FROM address"}]$^{(1,1)}$[{"WHERE studentId="}]$^{(0,1)}$
$\overline{\mathcal{SG}}$	OR [concat["SELECT * FROM addressWHERE studentId="] , concat["SELECT * FROM address"]]

(one OR, two concat, 37+21 simple nodes). The bricks list is, in this case, definitely more compact than the string graph.

`prog4` is inspired from an example in [33]. This program creates two strings `sql1` and `sql2` (which will be executed as SQL queries) by successive concatenations: each statement concatenates the previous value of the string variable with some other string (sometimes coming from another variable). The variable `l` is used in these concatenations, but we do not know the value of such variable at compile time since it is an input. In Tables 3.11 and 3.12 we report the results of the analysis on this case study with all our five domains:

- $\overline{\mathcal{CI}}$ discovers that (i) the string `sql1` surely contains all the characters of "SELECT", " " and "FROM", but it could contain any character of the alphabet K (because of the concatenation with `l`), and (ii) the string `sql2` surely contains all the characters of "UPDATE", " ", "SET" and " = ", but it could contain any character of the alphabet K (because of the concatenation with `l`);
- $\overline{\mathcal{PR}}$ tells us that `sql1` starts with "SELECT" and `sql2` starts with "UPDATE";
- $\overline{\mathcal{SU}}$ is not able to track any information about the resulting values of the two strings;
- as it happened in `prog3`, $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$ infer both the same information, which also corresponds *exactly* to the outcome of the program. The variable `sql1` starts with "SELECT ", then there is an unknown part (due to `l`), then it continues with " FROM " and it finally ends with another unknown part. The other string variable, `sql2`, starts with "UPDATE ", then there is an unknown part, then it continues with " SET " followed by another unknown part, then " = " and finally the last unknown part.

The information inferred by $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$ about the variable `sql1` is encoded, respectively, through 7 bricks (in the resulting bricks list) and 16 nodes (in the resulting

Table 3.11: Results of `prog4` for variable `sql1`

Abstract domain	**Value of `sql1`**
$\overline{\mathcal{CI}}$	$(\{E, T, F, M, L, \text{' '}, C, R, O, S\}, \mathsf{K})$
$\overline{\mathcal{PR}}$	"SELECT"
$\overline{\mathcal{SU}}$	$\top_{\overline{\mathcal{SU}}}$
$\overline{\mathcal{BR}}$	$[\{\text{"SELECT"}\}]^{(1,1)}[\{\text{" "}\}]^{(1,1)}\top_{\overline{\mathcal{B}}}$ $[\{\text{" "}\}]^{(1,1)}[\{\text{"FROM"}\}]^{(1,1)}[\{\text{" "}\}]^{(1,1)}\top_{\overline{\mathcal{B}}}$
$\overline{\mathcal{SG}}$	concat["SELECT " ; max ; " FROM " ; max]

Table 3.12: Results of `prog4` for variable `sql2`

Abstract domain	**Value of `sql2`**
$\overline{\mathcal{CI}}$	$(\{E, T, =, \text{' '}, U, A, P, D, S\}, \mathsf{K})$
$\overline{\mathcal{PR}}$	"UPDATE"
$\overline{\mathcal{SU}}$	$\top_{\overline{\mathcal{SU}}}$
$\overline{\mathcal{BR}}$	$[\{\text{"UPDATE"}\}]^{(1,1)}[\{\text{" "}\}]^{(1,1)}\top_{\overline{\mathcal{B}}}[\{\text{" "}\}]^{(1,1)}[\{\text{"SET"}\}]^{(1,1)}$ $[\{\text{" "}\}]^{(1,1)}\top_{\overline{\mathcal{B}}}[\{\text{" = "}\}]^{(1,1)}\top_{\overline{\mathcal{B}}}$
$\overline{\mathcal{SG}}$	concat["UPDATE " ; max ; " SET " ; max ; " = " ; max]

string graph). Note that, if we normalized the result of $\overline{\mathcal{BR}}$, we would reduce the number of bricks in the list to 4. The resulting information about `sql2` is, instead, encoded through 9 bricks (which could be reduced to 6 with a normalization step) and 19 nodes. Note that, on simple programs, the $\overline{\mathcal{BR}}$ and $\overline{\mathcal{SG}}$ domains tend to produce the same results. The (small) difference between the two domains lies in the widening operator, which is used to deal with loops. In such cases, $\overline{\mathcal{SG}}$ results are *slightly* better than those of $\overline{\mathcal{BR}}$, as it can be seen in the case study `prog2`. However, the bricks lists produced by $\overline{\mathcal{BR}}$ tend to be more compact than the graphs produced by $\overline{\mathcal{SG}}$, since each single character of the string corresponds to a node in the graph, while bricks deal with strings themselves and not single characters.

Regarding the performances of our analysis, the preliminary experimental results point out that $\overline{\mathcal{CI}}$ and $\overline{\mathcal{PR}} \times \overline{\mathcal{SU}}$ are quite efficient, $\overline{\mathcal{BR}}$ is slightly slower but still fast, while $\overline{\mathcal{SG}}$'s is the slowest domain of the framework. In fact, the analyses using $\overline{\mathcal{CI}}$ and $\overline{\mathcal{PR}} \times \overline{\mathcal{SU}}$ lasted just a fraction of second, using $\overline{\mathcal{BR}}$ a little more (always remaining below the second, though), while with $\overline{\mathcal{SG}}$ the analysis lasts some seconds, especially when the code is not trivial (e.g., with string concatenations inside loops).

3.8 Related Work

As already discussed in Section 3.1, strings are nowadays used in many applications: for example, to build SQL queries, to construct semi-structured Web documents, to create XPath and JavaScript expressions, and so on. After being dynamically generated, often in combination with user inputs, strings are sent to their respective processors. However, strings are usually not evaluated for their validity or security despite the importance of these issues. The static determination of approximated values of string expressions has then many practical applications (like checking the validity and security of generated strings, as well as to collect useful string properties). For these reasons, string analysis has been widely studied in the last years: in Section 3.1 we briefly cited some of the main approaches to this verification issue. These approaches can be classified with respect to the technique used for the analysis (i.e., to cite the most common ones, type systems, data-flow, Abstract Interpretation) and to the way the abstraction is built (i.e., through context-free grammars, push-down automata, regular expressions, etc.). For example, [109] uses Abstract Interpretation and automata, [67] employs data-flow and grammars, [142] devises a type system based on regular expressions, and so on. In this section we are now going to explore these approaches one by one in more depth, comparing them to our novel framework for string analysis.

Hosoya and Pierce [99] designed a statically typed processing language (called XDuce and based on the theory of finite tree automata) for building XML documents. Its sound type system ensures that dynamically generated documents conform to "templates" defined by the document types. This work differs a lot from ours, since, first of all, they use type systems instead of Abstract Interpretation. Moreover, they are focused on building XML documents, while our focus is on collecting possible values of generic string variables. Lastly, they require to manually annotate the code through types while our approach is completely automatic.

Tabuchi *et al.* [142] presented a type system based on regular expressions. It is focused on a minimal λ-calculus supporting concatenation and pattern matching. This calculus established a theoretical foundation of using regular expressions as types of strings in text processing languages. Also in this case (as in XDuce), the approach is very different from ours, since it employs type system. The only resemblance regards the use of regular expressions, which we use in the $\overline{\mathcal{BR}}$ domain.

Thiemann [144] introduced another type system for string analysis (based on context-free grammars) and presented a type inference algorithm based on Earley's parsing algorithm. It was not discussed how to deal with string operations other than concatenation (while in our framework we included the semantics of various other string operations). His analysis is more precise than those based on regular expressions, but his type inference algorithm is incomplete (though sound). Also, the analysis is tuned at a fixed level of precision.

Context-free grammars are also the basis of the analysis of Christensen, Møller and Schwartzbach [35]. This analysis (implemented in a tool called JSA, Java String

Analyzer) is tuned at a fixed level of abstraction and it statically determines the values of string expressions in Java programs. This work has considerable similarities with the $\overline{\mathcal{SG}}$ domain because type graphs are closely related to context free grammars. However, they generally obtain a regular grammar which *contains* the reference grammar (i.e., the grammar which encodes the possible outputs of the program), but they are not the same grammar. In the second case study of this chapter, `prog2`, $\overline{\mathcal{SG}}$ domain reaches a better precision than theirs, being able to model precisely the reference grammar ($S \rightarrow$ "a"|"0"S"1") without the need of any kind of approximation. Moreover, they precisely abstract only the concatenation operation, while for other string operators they use less precise automata operations or character set approximations; our work deals precisely also with other operators and can be easily extended to as many as needed. Møller published many other papers concerning abstractions for string analysis, but every one of them is strictly focused on some particular case ([127] on a set of HTML pages, [36, 111, 126] on XML documents, [124] on XSLT, [27] on XHTML, [105, 112, 110, 125] on type checking), without producing a unifying framework, while we aim at a higher generality.

To statically check the properties of Web pages generated dynamically by a server-side program, Minamide [122] developed a static program analysis that approximates the string output of a program with a context-free grammar. His analysis is based on the Java string analyzer (JSA) of [35], but the novelty of his analysis is the application of finite-state-automata transducers to revise the flow equations due to string-update operations embedded in the program, reaching a simpler and more precise analyzer than [35]. This work is specific for HTML pages. Even though the obtained results are similar to ours, his work suffers from some other limitations: after extracting from the program the corresponding grammar with operation productions, such grammar must be transformed into a context-free one. This restricts the string operations supported by the framework (to those which transform a context-free grammar into another context-free grammar) and it imposes that no string operation must occur in a cycle of production. Finally, the validity check between the reference grammar and the context-free grammar is very costly and can be done only when the nesting depth of the elements in the generated document is bounded.

A combination of grammars and Abstract Interpretation was studied by Cousot and Cousot in [59], where they showed that set constraint solving of a particular program `P` could be understood as an Abstract Interpretation over a finite domain of tree grammars, constructed from `P`. However, their work is at a very high level and their concern is not the approximation of string variables, so no string operators are considered in such article.

Abstract Interpretation specifically focused on string analysis can be found in Choi *et al.* [33], where they used standard abstract-interpretation techniques with heuristic widening to devise a string analyzer that handles heap variables and context sensitivity. They selected a restricted subset of regular expressions as abstract domain (which results in limited loss of expressibility). Our $\overline{\mathcal{BR}}$ domain is similar to

this work, and, even though most of the lattice operators are different, we obtain the same result on the second case study (0^*a1^*). $\overline{\mathcal{SG}}$ domain, instead, is more precise than their domain. In fact, on the second case study (`prog2`) the string graphs are able to produce exactly the reference grammar ($0^n a 1^n$), while their result does not constrain the number of 0s and 1s to be the same.

Kim and Choe [109] introduced recently another approach to string analysis based on Abstract Interpretation. They abstract strings with pushdown automata (PDA). The result of their analysis is compared with a grammar to determine if all the strings generated by the PDA belong to the grammar. This approach has a fixed precision, and in the worst case (not often encountered in practice) it has exponential complexity.

Automata were also exploited by Yu *et al.* in [150]. They presented an automata-based approach for the verification of string operations in PHP programs based on symbolic string analysis. They encode the set of string values that string variables can take as deterministic finite automaton (DFA): the language accepted by the DFA corresponds to the values that the corresponding string variable can assume at that program point. Using this technique, it is possible to automatically verify the sanitization of a string, showing that attacks are not possible. The information tracked by this analysis is fixed, and it is specific for PHP programs. However, in 2011 they proposed a unifying framework [151] of their previous works, i.e. an abstraction lattice which can be tuned to provide various trade-offs between precision and performance. The framework is based on the *regular abstraction* [152], a relational analysis in which values of string variables are represented as multi-track DFA (each track corresponds to a specific string variable). As the number of variables increases, such relational analysis becomes intractable, so they add two other string abstraction (*relation abstraction* and *alphabet abstraction*) to improve the scalability of their approach. They also propose a heuristic to choose a particular point in their abstraction lattice, depending on the program and property to be verified. The *alphabet abstraction* can be seen as a more complex version of $\overline{\mathcal{CI}}$, since it also keeps track of the position of characters; such abstraction must be applied to automata, thus obtaining more convoluted operations than in our domain $\overline{\mathcal{CI}}$.

Doh *et al.* [67] reported the "abstract parsing" technique, which statically analyzes string values from programs. They combine LR(k)-parsing technology and data-flow analysis to analyze, in advance of execution, the documents dynamically generated by a program. Based on the document languages context-free reference grammar and the programs control structure, the analysis predicts how the documents will be generated and parses the predicted documents. Their technique is quite precise, but the level of abstraction is fixed.

Given this context, our work is the first one (together with [151], published at the same time as [45]) that (i) is a generic, flexible, and extensible approach to the analysis of string values, and (ii) can be tuned at different levels of precision and efficiency.

3.9 Discussion

In this chapter we approached the issue of string analysis through Abstract Interpretation. In particular, we focused on the construction of a flexible and generic framework to approximate the values assumed by string variables. The results obtained through our approximation can then be used to verify different kind of properties, depending on the specific program to analyze.

To this purpose, we designed five abstract domains ($\overline{\mathcal{CI}}$, $\overline{\mathcal{PR}}$, $\overline{\mathcal{SU}}$, $\overline{\mathcal{BR}}$, $\overline{\mathcal{SG}}$), each of them tracking a different type of information. Since a string can be seen as an ordered sequence of characters, such domains all focus on the two aspects of character inclusion and order between characters. For each domain, we defined its lattice structure and the abstract semantics of a significant subset of string-manipulating operations. All domains are equipped with formal proofs to show their correctness. We applied each domain to some case studies, especially chosen to illustrate the operating principles of our domains. From such applications, we gathered that the first three domains ($\overline{\mathcal{CI}}$, $\overline{\mathcal{PR}}$, $\overline{\mathcal{SU}}$) are quite simple and the information we can trace with them is limited. However, they are not computationally expensive (the prefix and suffix in particular) and they do not need to define a widening operator. For these reasons, they could be useful when not much precision is needed and the performance of the analysis is a critical point. The last two domains ($\overline{\mathcal{BR}}$, $\overline{\mathcal{SG}}$) are certainly more complex, and they let us trace more interesting patterns. The lattices of such domains are infinite and do not satisfy ACC; thus, we had to define a widening operator. Even though these two domains are both quite precise, $\overline{\mathcal{SG}}$ seems to be the *most* precise domain of our framework (and, for the usual trade-off between performance and precision, the most costly).

4

The Trapezoid Step Functions Abstract Domain

In this chapter we focus on the second of the three goals of our thesis, i.e. improving the existing abstraction of continuous inputs of hybrid systems given by [24] (IVSF domain).

Hybrid systems are made up of discrete (that is, the program) and continuous (that is, the physical environment) components. The program receives inputs from the physical environment through sensors that are usually modelled by volatile variables. Hybrid systems often deal with safety critical systems, like flight controllers. The reliability of these systems is crucial: even a single bug can spell disaster, and this is a relevant challenge for formal verification methods. It is thus necessary to precisely approximate both the discrete and the continuous parts of the system. In static analysis, a lot of precise abstractions already exist to approximate the discrete part of the system, while there is still not much work done on the continuous part. In this chapter we focus on this problem and we introduce the Trapezoid Step Functions (TSF) domain. Our main insight is to approximate $\mathcal{C}^2_+$ functions [1] by a finite sequence of trapezoids, one for each slot of time, adopting linear functions (instead of constant intervals) to abstract the upper and the lower bounds of a continuous variable in each slot of time. We formalize the lattice structure of TSF, and show how to build and compute a sound abstraction of a given continuous function. Finally, we apply TSF to some case studies comparing our results with the domain of Interval Valued Step Functions (IVSF), the current state-of-the-art in the context of abstraction of continuous functions in static analysis. The experimental results underline the effectiveness of the approach in terms of both precision and efficiency.

This chapter is structured like described in 1.6, with some small differences (i.e., no specific notation is needed, and the abstraction function of the proposed domain requires an entire section because of its complexity). In particular, Section 4.1 introduces the problem, Section 4.2 presents a case study, Section 4.3 establishes the syntax of the language supported by our analysis, Section 4.4 defines the concrete domain, while Sections 4.5, 4.6 and 4.7 formalize the abstract domain, its abstraction function and the abstract semantics of operations on functions. In Section 4.8 we

[0] This chapter is partially derived from [46].

[1] We define $\mathcal{C}^2_+$ as the set of all continuous functions in $\mathbb{R}^+ \mapsto \mathbb{R}$ that have continuous first two derivatives.

present some experimental results when applying TSF to the abstraction of different functions, and how our results compare with IVSF. Finally, Section 4.9 discusses the related work and Section 4.10 concludes.

Contents

4.1 Introduction

Context

A *hybrid system* is a dynamical system whose evolution depends on a tight coupling between variables that take values in a continuum and variables that take values in a finite or countable set [146]. More generally, hybrid systems are mixtures of real-time (continuous) dynamics and discrete events. These continuous and discrete dynamics not only coexist, but *interact* as well: changes occur both in response to discrete, instantaneous, events and in response to dynamics as described by differential or difference equations in time. Hybridness is characteristic of all *embedded control systems* because it arises from several sources. One of these sources is that every digital hardware/software implementation of a control design is ultimately a discrete approximation that interacts through sensors and actuators with a continuous physical environment. A simple example of hybrid system is a temperature control system consisting of a heater and a thermostat, where the variables of such system include the current temperature (a real value) and the operating mode of the heater (a boolean value indicating on or off). The hybridness comes from the interaction of such variables (i.e., if the thermostat switches off when the temperature reaches a certain threshold).

Hybrid systems are everywhere (cars, airplanes, computers, and so on) and their use is becoming more and more pervasive during the last years. For example, the introduction of advanced automation into manually operated systems has been extremely successful in increasing the performance and flexibility of such systems, as well as significantly reducing the workload of the human operator. However, accompanying this increase in automation is the necessity of ensuring that the automated system always performs as expected. This is especially crucial for safety critical systems: if an error occurs in the automated avionics on board a commercial jet, the results can be disastrous. In the past, the validation of such systems relied on two main factors: (i) operating the system well within its performance limits and (ii) extensive testing. In fact, today most of the cost in control system development is spent on validation techniques that rely almost exclusively on exhaustively testing more or less complete versions of complex non-linear control systems. To address this bottleneck, approaches to dynamic modelling have been developed, which focus mainly on one part of the system (the discrete one or the continuous one) and deal in an ad-hoc manner with the other part. Such approaches, however, are not enough. A well-known and dramatic example is the Ariane 5 launch in 1996 (self-destruction mode 37 seconds after lift-off), already cited in Chapter 1. This failure was caused by a software error. However, the bugged program was the same as the one that had worked perfectly in the launch of Ariane 4. The problem was that the continuous dynamical system around the software had changed: the physical structure of the new launcher had been sized up considerably compared to its predecessor. The catastrophe was then due to the execution of a trusted code within a new physi-

cal environment. For these reasons, more systematic ways of dealing with hybrid systems are necessary: the newest research direction in control focuses on building an analytical foundation based on hybrid systems. A leading objective is to extend standard program analysis techniques to systems which incorporate some kind of continuous dynamics.

The area of hybrid systems is then loosely defined as the study of systems which involve the interaction of discrete event and continuous time dynamics, with the purpose of proving properties such as reachability and stability. In the past, basic formal models (hybrid automata) for hybrid systems have been built and various approaches for hybrid control law design, simulation, and verification have been developed. One class of approaches to modelling and analysis of hybrid systems has been to extend techniques for finite state automata to include systems with simple continuous dynamics. These approaches generally use: *model checking* (which verifies a system specification symbolically on all system trajectories) and *deductive theorem proving* (which proves a specification by induction on all system trajectories). A second class of models and analysis techniques for hybrid systems comes from research in continuous state space and continuous time dynamical systems and control. In this last case, the emphasis has been on extending the standard modelling, reachability and stability analyses, and controller design techniques to capture the interaction between the continuous and discrete dynamic.

Let us now consider a more detailed example of a basic hybrid system, the bouncing ball modelled in Figure 4.1 (the case study in Chapter 5 is a complication of such example). The bouncing ball is cited in almost every text on hybrid systems, for example [115, Chapter 3] and [146, Chapter 2]. We report it here in the form presented in [115]. The bouncing ball is a physical system with impact. The ball is dropped from an initial height and bounces off the ground, dissipating its energy with each bounce. The continuous dynamics of the system consists in the motion of the ball from one bounce to the following one. The discrete dynamics is represented by the impact of the ball into the ground, since its velocity undergoes a discrete change modelled after an inelastic collision. A model for a bouncing ball can be represented as the simple hybrid system in Figure 4.1, with a single discrete state and a continuous state of dimension two $x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, where x_1 denotes the vertical position of the ball and x_2 its vertical velocity. The continuous motion of the ball is governed by Newton's laws of motion. This is indicated by the differential equation that appears in the vertex, where g denotes the gravitational acceleration. This differential equation is only valid when the ball is above the ground (i.e., as long as $x_1 \geq 0$), as we can see from the logical expression $x_1 \geq 0$ that appears in the vertex below the differential equation. The ball bounces when $x_1 = 0$ and $x_2 \leq 0$, and we can see that from the logical expression that appears near the beginning of the edge (arrow). At each bounce, the ball loses a fraction of its energy, as we can see from the equation $x_2 := -cx_2$ (with $c \in [0, 1]$) near the end of the edge. This statement is an assignment, which means that after the bounce the speed of

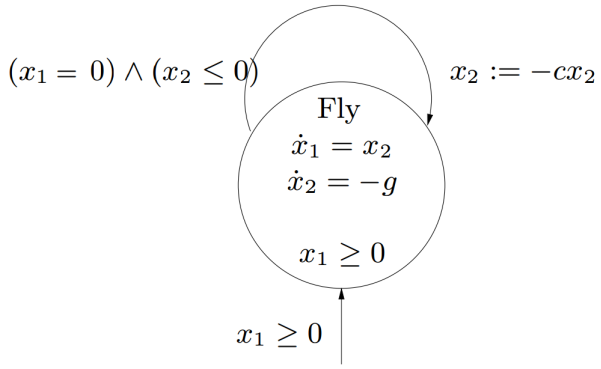


Figure 4.1: The hybrid system modelling a bouncing ball [115]

the ball will be c times the speed of the ball before the bounce, and going to the opposite direction. The bouncing ball is an interesting hybrid system, as it exhibits Zeno behaviour. Zeno behaviour can be informally described as the system making an infinite number of jumps in a finite amount of time [96]. In this example, each time the ball bounces it loses energy, making the subsequent jumps (impacts with the ground) closer and closer together in time.

State of the art

On the one hand, there is a large literature on the static analysis of discrete programs. On the other hand, these approaches do not perform well when they are applied to continuous environments. For instance, in the context of the Abstract Interpretation framework, the Interval domain [53] abstracts continuous systems with the minimal and the maximal values a sensor can return at any time. This approach is definitely too rough for the static analysis of embedded programs. To improve precision, Bouissou and Martel recently proposed the Interval Valued Step Functions (IVSF) domain [24] for approximating the behaviour of a function in a given interval of time (i.e, a step) with the minimal and the maximal values the function could achieve during that period of time.

Contribution

In this chapter, we go one step further by introducing the Trapezoid Step Functions (TSF) domain. TSF abstracts the values of a function in a given slot of time with two linear functions, tracking in this way linear relationships between the time and the output value. Our work is inspired by the evolution of numerical domains. Since the Interval domain is not precise enough in many contexts, various relational domains (e.g., Polyhedra [62] and Octagons [123]) have been introduced. In a similar way, our domain, instead of tracking the minimal and the maximal values of a function in a given slot of time, tracks two linear functions that approximate values of the function

```
1  #define SUP = 4
2  #define INF = -4
3  volatile float x;
4  static float intgrx = 0.0, h = 1.0/8;
5  void main() {
6    while(true) { //assume frequency = 8 KHz
7      float xi = x;
8      intgrx += xi*h;
9      if(intgrx > SUP)
10             intgrx = SUP;
11     if(intgrx < INF)
12             intgrx = INF;
13   }
14 }
```

Figure 4.2: Simple integrator

as a linear multivalued function [15, 28]. The two linear functions, together with the two vertical lines that delimit the time slot, form a trapezoid. We approximate the function with a finite number of trapezoids, one for each step.

The main contributions of this chapter are (i) the formal definition of TSF and its lattice operators, (ii) the introduction of a sound abstraction function that, given a continuous and derivable function, builds up its abstraction in TSF, and (iii) the discussion of some experimental results and the comparison with the ones obtained by IVSF. The experimental results strongly emphasize how relevant is the accuracy improvement when adopting TSF, possibly combined with the IVSF domain, and the effectiveness of the abstract operations on TSF.

4.2 Case Study

In this chapter, we chose to use the case study reported in Figure 4.2: it regards a special case of hybrid system, where we have a discrete system (an embedded program) which takes a continuous environment as input.

This motivating example has been inspired by [83], and it is the same example used in [24] in order to show the main features of IVSF. This program is an integrator with thresholds, a quite common component of embedded software: it integrates a function (whose values are provided through the `volatile` variable `x`) using the rectangle method on a sampling step `h`. The integration is carried out up to some threshold defined by the interval $[INF, SUP]$. We assume that the function we integrate is $\sin(2\pi t)$, and that the input data are given by a sensor (hence the `volatile` variable `x`) at a frequency of 8KHz (it is very frequent to find a comment such as "this loop runs at 8KHz" in the code usually given to static analyzers). This scenario

$$V \in \mathcal{V}, f \in \mathcal{C}^2_+, c \in \mathbb{R}, x \in \mathbb{R}^+$$

$$E := V \mid \texttt{newFun}(f) \mid \texttt{constFun}(c) \mid |E| \mid E\ \texttt{aop}\ E \text{ where } \texttt{aop} \in \{+, -, \times, \div, \circ, \min, \max\}$$

$$R := \texttt{constVal}(c) | \texttt{valueAt}(E, x)$$

$$B := B\ \texttt{and}\ B | \texttt{not}\ B | B\ \texttt{or}\ B | R\ \texttt{bop}\ R \text{ where } \texttt{bop} \in \{\geq, >, \leq, <, \neq\}$$

$$P := V = E | \texttt{if}(B)\ \texttt{then}\ P\ \texttt{else}\ P | \texttt{while}(B)\ P | P; P$$

Figure 4.3: Syntax

is particularly interesting for the analysis of numerical precision, since its behaviour is extremely depending of the input data (i.e. the physical environment), of the frequency of the integration process (i.e. the sampling rate) and of the precision of the sensor. The sensor will produce the sequence of values $[0, \frac{\sqrt{2}}{2}, 1, \frac{\sqrt{2}}{2}, 0, -\frac{\sqrt{2}}{2}, -1, -\frac{\sqrt{2}}{2}]$ on `x`. Therefore, in a perfect arithmetic computation the summation of these values multiplied by `h` will be equal to zero after $8 \times i$ iterations $\forall i \in \mathbb{N}$ and in general will be bounded between $[0, 2]$. Nevertheless, in a real system this summation would produce some approximate values because of floating point approximation.

As explained in the previous section, a naive abstraction of the continuous environment would approximate `x` by the interval $[-1, 1]$ (since it is a value coming from the trigonometric function sin). Then, after unrolling the main loop n times, the interval-based analyzer would bind the variable `intgrx` with the interval $[-n \times h, n \times h]$. After 104 iterations, for example, the value of `intgrx` would be included in $[-4, 4]$, because of the thresholds (otherwise it would be even bigger, $[-13, 13]$). Such result can definitely be improved by using a better abstraction of the continuous environment.

This code is particularly interesting to test the precision of abstract domains since it propagates the approximation error of our abstract domain at each iteration of the `while` loop, and therefore it is a good candidate to test the precision of TSF.

4.3 Language Syntax

Let $\mathcal{V}$ be a finite set of variables, c a real value and $\mathcal{C}^2_+$ the set of all univariate functions from $\mathbb{R}^+$ to $\mathbb{R}$ with two continuous derivatives, respectively. Figure 4.3 defines the language supported by our analysis.

We focus on programs dealing with operations over function-valued variables (numerical variables can be easily added to the syntax and included in the analysis, by working in cooperation with an already existent numerical abstract domain). We consider expressions built through the most common mathematical operators (sum, subtraction, multiplication and so on). A functional expression E can be

a constant function ($\texttt{constFun}(c)$ where $c \in \mathbb{R}$), a generic function ($\texttt{newFun}(f)$ where $f \in \mathcal{C}^2_+$), a variable ($V \in \mathcal{V}$), the modulus of a functional expression ($|E|$) or the pointwise combination of two functional expressions (E `aop` E where `aop` $\in \{+, -, \times, \div, \circ, \min, \max\}$). We also consider boolean conditions built through the comparison of two numerical expressions, where a numerical expression can be a simple numeric constant ($\texttt{constVal}(c)$ where $c \in \mathbb{R}$) or the value of a functional expression in a specific point of its domain ($\texttt{valueAt}(E, x)$ where $x \in \mathbb{R}^+$). Boolean conditions can be combined as usual with logical operators (and, or, not). As for statements, we support the assignment of an expression to a variable, `if` $-$ `then` $-$ `else`, `while` loops, and concatenation.

4.4 Concrete Domain and Semantics

The concrete domain $\mathcal{D}$ is defined as the power-set of continuous functions in $\mathbb{R}^+ \to \mathbb{R}$ [2] which have two continuous derivatives (i.e., the set $\mathcal{C}^2_+$).

As explained in Section 2.4, when the lattice is the power-set of a set, the other operators immediately follow: the partial order is the set inclusion $\subseteq$, the least upper bound corresponds to set union $\cup$, the greatest lower bound corresponds to set intersection $\cap$, the top element $\top$ is the set itself, while the bottom element $\bot$ is $\emptyset$. The complete definition of the lattice $\mathcal{D}$ is then:

$$\mathcal{D} = \langle \wp(\mathcal{C}^2_+), \subseteq, \cup, \cap, \mathcal{C}^2_+, \emptyset \rangle$$

We can now define the concrete semantics of the language introduced in Section 4.3. For the statements operations and the logic combination of boolean conditions we refer to the usual semantics of the classical Abstract Interpretation framework. Then, we will have to specify only the semantics (both concrete and abstract) of operators dealing with functional expressions. We formalize the concrete semantics in Table 4.1.

For the creation of functional expressions, we define the semantics $\mathbb{S}$ that, given the statement and eventually some sets of concrete functional expressions in $\mathcal{C}^2_+$ (containing the values of the arguments of the statement), returns a set of concrete functions resulting from that operation. In particular:

- `newFun(f)` returns a singleton containing `f`;
- `constFun(c)` returns a singleton containing the constant function which always assumes value `c`;
- || returns, for every function of the set in input, another function obtained by computing its modulus in every point of the domain;

[2] The functions' domain is $\mathbb{R}^+$ instead of $\mathbb{R}$ because, in the scenario of embedded programs, the input variable represents the time.

Table 4.1: Concrete semantics

$$
\begin{array}{ll}
\mathbb{S}[\![\texttt{newFun(f)}]\!]() & = \{\lambda x.\texttt{f}(x)\ \forall x \in \mathbb{R}^+\} \\
\mathbb{S}[\![\texttt{constFun(c)}]\!]() & = \{\lambda x.\texttt{c}\ \forall x \in \mathbb{R}^+\} \\
\mathbb{S}[\![|\ |]\!](\mathsf{F}) & = \{\lambda x.|\texttt{f}(x)|\ \forall \texttt{f} \in \mathsf{F}, \forall x \in \mathbb{R}^+\} \\
\mathbb{S}[\![\texttt{aop}]\!](\mathsf{F}_1, \mathsf{F}_2) & = \{\lambda x.\texttt{f}_1(x)\ \texttt{aop}\ \texttt{f}_2(x)\ \forall(\texttt{f}_1, \texttt{f}_2) \in \mathsf{F}_1 \times \mathsf{F}_2, \forall x \in \mathbb{R}^+\} \\
 & \text{where } \texttt{aop} \in \{+, -, \times, \div\} \\
\mathbb{S}[\![\circ]\!](\mathsf{F}_1, \mathsf{F}_2) & = \{\lambda x.\texttt{f}_1(\texttt{f}_2(x))\ \forall(\texttt{f}_1, \texttt{f}_2) \in \mathsf{F}_1 \times \mathsf{F}_2, \forall x \in \mathbb{R}^+\} \\
\mathbb{S}[\![\min]\!](\mathsf{F}_1, \mathsf{F}_2) & = \{\lambda x.\min(\texttt{f}_1(x), \texttt{f}_2(x))\ \forall(\texttt{f}_1, \texttt{f}_2) \in \mathsf{F}_1 \times \mathsf{F}_2, \forall x \in \mathbb{R}^+\} \\
\mathbb{S}[\![\max]\!](\mathsf{F}_1, \mathsf{F}_2) & = \{\lambda x.\max(\texttt{f}_1(x), \texttt{f}_2(x))\ \forall(\texttt{f}_1, \texttt{f}_2) \in \mathsf{F}_1 \times \mathsf{F}_2, \forall x \in \mathbb{R}^+\} \\
\mathbb{V}[\![\texttt{constVal(c)}]\!]() & = \{\texttt{c}\} \\
\mathbb{V}[\![\texttt{valueAt}_\texttt{x}]\!](\mathsf{F}) & = \{\texttt{f}(\texttt{x}) : \texttt{f} \in \mathsf{F}\} \\
\mathbb{B}[\![\texttt{bop}]\!](\mathsf{R}_1, \mathsf{R}_2) & = \begin{cases} \mathsf{true} & \text{if } \forall \texttt{r}_1 \in \mathsf{R}_1, \texttt{r}_2 \in \mathsf{R}_2 : \texttt{r}_1\ \texttt{bop}\ \texttt{r}_2 \\ \mathsf{false} & \text{if } \forall \texttt{r}_1 \in \mathsf{R}_1, \texttt{r}_2 \in \mathsf{R}_2 : \texttt{r}_1\ \texttt{inv(bop)}\ \texttt{r}_2 \\ \top_\mathsf{B} & \text{otherwise} \end{cases}
\end{array}
$$

where $\texttt{bop} \in \{\geq, >, \leq, <, \neq\} \wedge$

bop	inv(bop)
$\geq$	$<$
$>$	$\leq$
$\leq$	$>$
$<$	$\geq$
$\neq$	$=$

- $\texttt{aop} \in \{+, -, \times, \div\}$ returns, for every pair of functions (one from the first set in input, one from the second one), the function resulting from their pointwise combination through aop. For example, when aop is $+$, we return the function obtained by pointwise summing the two original functions.

- $\circ$ returns, for every pair of functions (one from the first set in input, one from the second one), the function resulting from their pointwise composition.

- min and max respectively return, for every pair of functions f_1, f_2 (one from the first set in input, one from the second one), the function which assumes, in each point of the domain, the minimum (resp., maximum) value between the two values assumed by f_1 and f_2.

For the creation of numerical expressions, we define the semantics $\mathbb{V}$ that returns a set of concrete real values resulting from the specific operation. In particular:

- $\texttt{constVal}(\texttt{c})$ returns a singleton containing c;

- $\texttt{valueAt}_{\texttt{x}}$ returns, for every function of the set in input, its value in correspondence of the input x.

For $\texttt{bop} \in \{\geq, >, \leq, <, \neq\}$ we define the semantics $\mathbb{B} : \wp(\mathbb{R}) \times \wp(\mathbb{R}) \to \{\mathsf{true}, \mathsf{false}, \top_{\mathsf{B}}\}$. Given two sets of real values, the semantics of this operator returns:

- true if all the possible pairs of values r_1, r_2 (one from the first set in input, one from the second one) satisfy the boolean comparison operator bop. For example, when bop is $\geq$, we return true if all values in R_1 are greater or equal than those in R_2.

- false if all the possible pairs of values r_1, r_2 (one from the first set in input, one from the second one) satisfy the inverse comparison operator of bop. For example, when bop is $\geq$, its inverse is $<$ (see Table 4.1 for the complete list of inverse operators); thus, we return false if all values in R_1 are strictly smaller than all those in R_2.

- $\top_{\mathsf{B}}$ otherwise. This special boolean value represents a situation in which the boolean condition may be evaluated to true some times, and to false other times, depending on the two specific values in $\mathsf{R}_1, \mathsf{R}_2$ we are considering. The partial order $\geq_{\mathsf{B}}$ over these values was already defined in the previous chapter (Section 3.5).

Note that the concrete semantics of our language is uncomputable because the functions domain is $\mathbb{R}^+$ and we cannot compute their values in each point of the domain in a finite time.

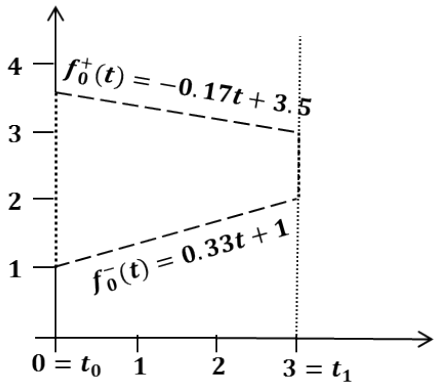


Figure 4.4: Example of a trapezoid defined on $[0, 3]$

4.5 Abstract Domain

In this section we are going to present the abstract domain of Trapezoid Step Functions, TSF. The presentation is roughly structured as explained in Section 1.6 (i.e.: domain elements definition, partial order, top and bottom elements, glb and lub operators, abstraction and concretization functions, widening), with the only difference that in this section we are only going to give a validity condition for the definition of the abstraction function, while the *actual* abstraction function is defined in a separate section (4.6) because of its complexity. The abstract semantics, also, requires a section on its own (4.7). We prove the correctness of all our definitions, by showing the soundness of each operation on abstract elements (partial order, refine, glb, lub, compact, widening), thus demonstrating that TSF is a lattice.

We start the abstract domain definition by introducing the key-idea behind it. Given a function f and a set of ordered indices $\{t_i\}_{0 \leq i \leq N}$, we approximate the values of f in a step $[t_i, t_{i+1}]$ by a trapezoid whose (i) two parallel sides are vertical, in correspondence of t_i and t_{i+1}, and (ii) the other two sides are in the form $f^-(x) = m^- x + q^-$ and $f^+(x) = m^+ x + q^+$ and approximate lower and upper values of f inside $[t_i, t_{i+1}]$. For instance, Figure 4.4 depicts a trapezoid defined on the step $[0, 3]$ with $f^-(t) = 0.33t + 1$ and $f^+(t) = -0.17t + 3.5$ as lower and upper sides, respectively.

Formally, given a step $[t_i, t_{i+1}]$, a single trapezoid is defined by two linear functions, and each of these two functions is defined by two real numbers representing the slope and intercept. Therefore, the pair of sides of each trapezoid are defined by a tuple

$$\mathbf{v} = (m^-, q^-, m^+, q^+)$$

where $m^-, q^-, m^+, q^+ \in \mathbb{R} \cup \{-\infty, +\infty\}$, which represents the two lines $f^-(x) = m^- x + q^-$ and $f^+(x) = m^+ x + q^+$. We denote by f^- and f^+ the lower and the upper side, respectively. TSF can be seen as a generalization of IVSF, whose two horizontal sides are parallel, i.e., with $f^+(x) = q^+$ and $f^-(x) = q^-$.

Note that the two lines of a trapezoid in TSF can also go to ∞: in particular, we define that f^- can have value $-\infty$ and f^+ can have value $+\infty$. In such cases, the trapezoid does not have a lower or upper bound. In our domain, this is represented by $m^- = q^- = -\infty$ or $m^+ = q^+ = +\infty$. [3] We do not admit the case where the slope and intercept of a line are both ∞ but with different sign (i.e., $f(x) = -\infty \times x + \infty$ is not allowed).

Following the standard notation on step functions [24], consider a generic set V of values. These values could be simple numbers (integers, reals, ...) but also more complicated structures like intervals or tuples. In our domain, a value will be composed by two lines, the upper (f^+) and the lower (f^-) side of trapezoids. Then, we represent a step function from time to V as a conjunction of constraints of the form "$t_i : \mathbf{v_i}$" such that $t_i \in \mathbb{R}^+ \wedge \mathbf{v_i} \in V$. This means that the step function switches to $\mathbf{v_i}$ at time t_i. For $t \in [t_i, t_{i+1}]$ the function respects the constraints represented by value $\mathbf{v_i}$. When $t = t_{i+1}$, the abstract value of the function switches "abruptly" to $\mathbf{v_{i+1}}$. We consider only finite conjunctions of constraints, otherwise the abstract operations of our domain would not be computable. A finite sequence of constraints $f = t_0 : \mathbf{v_0} \wedge t_1 : \mathbf{v_1} \wedge \cdots \wedge t_N : \mathbf{v_N}$ represents the step function f such that $\forall t \in \mathbb{R}^+ : f(t) = \mathbf{v_i}$ with $i = \max(\{j \in [0, N] : t_j \leq t\})$. We use the compact notation $f = \bigwedge_{0 \leq i \leq N} t_i : \mathbf{v_i}$, with $N \in \mathbb{N} \wedge \forall i \in [0, N] : (t_i \in \mathbb{R}^+ \wedge \mathbf{v_i} \in V)$. V is the set of tuples $\{(m^-, q^-, m^+, q^+) : m^-, q^-, m^+, q^+ \in \mathbb{R} \cup \{-\infty, +\infty\}\}$. To lighten up the notation, a quadruple $\mathbf{v_i} = (m_i^-, q_i^-, m_i^+, q_i^+)$ denotes $f_i^-(t) = m_i^- t + q_i^-$ and $f_i^+(t) = m_i^+ t + q_i^+$. We will alternatively denote a step value as $\mathbf{v_i} = (m_i^-, q_i^-, m_i^+, q_i^+)$ or $\mathbf{v_i} = (f_i^-, f_i^+)$.

For example, the step function h with two steps $t_0 = 0, t_1 = 3$ with values, respectively, $v_0 = (1, -1, 2, 1), v_1 = (0, 3.5, \frac{1}{3}, 4)$ will be written as $h = (0 : [1, -1, 2, 1]) \wedge (3 : [0, 3.5, \frac{1}{3}, 4])$. The graphical representation of h (for the restricted domain $[0, 10]$) is depicted in Figure 4.5, where we can see the two trapezoids composing the step function. On the upper and lower sides of the trapezoids we reported the equations of the corresponding lines.

4.5.1 Normal Form and Equivalence Relation

However, different abstract values may represent exactly the same step function. For example, the conjunctions $(0 : [0, 0, 1, 1]) \wedge (4 : [0, 0, 1, 1])$ and $(0 : [0, 0, 1, 1]) \wedge (7 : [0, 0, 1, 1])$ define the same step function which, for every input $t \in [0, +\infty)$, returns as output value the interval $[0, t + 1]$. To avoid that, we use the same notion of normal form defined in [24]:

1. the switching times t_i of a conjunction are sorted and different (if $f = \bigwedge_{0 \leq i \leq N} \{t_i : \mathbf{v_i}\}$ then $0 = t_0 < t_1 < \cdots < t_n < \cdots < t_N$);

[3] The numerical order relationship $\leq$ is extended to consider $-\infty$ and $+\infty$.

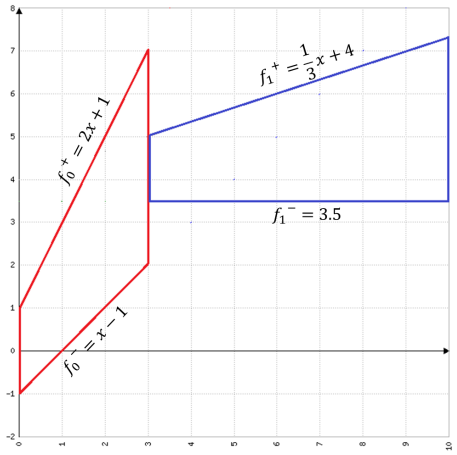


Figure 4.5: A TSF abstract element on the domain $[0, 10]$

2. two consecutive constraints cannot have equal values (each $\mathbf{v_i}$ must be different from $\mathbf{v_{i+1}}$, that is $\forall i \in [0, N-1], \mathbf{v_i} \neq \mathbf{v_{i+1}}$)

With these conditions, the representation is unique. We will denote by *Norm* the normalization procedure. The algorithm to compute the normalized form $Norm(f)$ of a given conjunction of constraints f is defined as follows. First we sort the constraints by ascending switching times, with the convention that if two constraints have the same time, then we only keep the one with the highest index. In this way, we fulfil the first normalization condition. Then we remove any constraint $t_i : \mathbf{v_i}$ such that $\mathbf{v_{i-1}} = \mathbf{v_i}$. In this way, we fulfil the second normalization condition. Note that the normalization process does not change the meaning of the representation: for a conjunction f, it holds that $\forall t \in \mathbb{R}^+ f(t) = Norm(f)(t)$. In our previous example, we would obtain a representation with a single constraint $0 : [0, 0, 1, 1]$.

Given two normalized conjunctions, we define the same equality test as in [24]:

$$\bigwedge_{0 \leq i \leq N} \{t_i : \mathbf{v_i}\} = \bigwedge_{0 \leq j \leq M} \{u_i : \mathbf{w_i}\} \Leftrightarrow N = M \wedge \forall i \in [0, N], t_i = u_i \wedge \mathbf{v_i} = \mathbf{w_i}$$

The normalization process induces an equivalence relation $\equiv$ defined by $f \equiv g \Leftrightarrow Norm(f) = Norm(g)$.

4.5.2 Validity Constraints

We impose two constraints on abstract elements in order to be valid:

$$\forall i \in [0, N], f_i^-(t_i) \leq f_i^+(t_i) \wedge f_i^-(t_{i+1}) \leq f_i^+(t_{i+1}) \tag{4.1}$$
$$\forall i \in [0, N-1], [f_i^-(t_{i+1}), f_i^+(t_{i+1})] \cap [f_{i+1}^-(t_{i+1}), f_{i+1}^+(t_{i+1})] \neq \emptyset \tag{4.2}$$

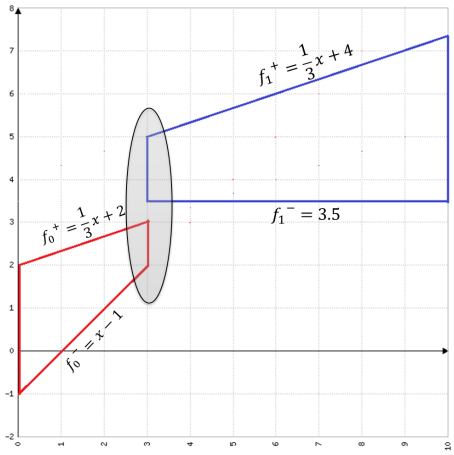


Figure 4.6: A TSF abstract element on the domain $[0, 10]$ which violates the second validity constraint at $t = 3$

(1) states that, at each step $[t_i, t_{i+1}]$ of the function, the two lines f_i^+ and f_i^- do not intersect. This assumption is not restrictive, since we can always split a step with intersecting sides into two smaller steps with not intersecting sides through the *refine* operator defined in Section 4.5.5.

(2) states that two consecutive steps $[t_i, t_{i+1}], [t_{i+1}, t_{i+2}]$ of a step function must have at least one point in common at t_{i+1}. This means that the interval of values at $t = t_{i+1}$ individuated by $\mathbf{v_i}$ and the one individuated by $\mathbf{v_{i+1}}$ at the same input t must have a non-empty intersection. This constraint is needed because otherwise the concrete functions represented by the abstract element would not be continuous, and it would be abstracted by a bottom value in our domain since we abstract functions in $\mathcal{C}^2_+$. In Figure 4.6 we can see an example of an abstract state which violates this constraint. In particular, at $t = 3$ we can see that the upper side of the left trapezoid passes for $y = 3$, while the lower side passes for $y = 2$, so the interval of values individuated by the left trapezoid is $[2, 3]$. The interval individuated by the right trapezoid, on the other hand, is $[3.5, 5]$. Since $[2, 3] \cap [3.5, 5] = \emptyset$, this abstract state is not valid and it collapses to bottom. This is a sound approximation since there is no continuous function that may satisfy this constraint.

4.5.3 Abstract Elements

The elements (that is, states) of our abstract domain, denoted by $D^\sharp$, are normalized finite conjunctions of constraints

$$f = \bigwedge_{0 \leq i \leq N} t_i : \mathbf{v_i}$$

with

$$N \in \mathbb{N} \wedge \forall i \in [0, N] : (t_i \in \mathbb{R}^+ \wedge \mathbf{v_i} \in V)$$

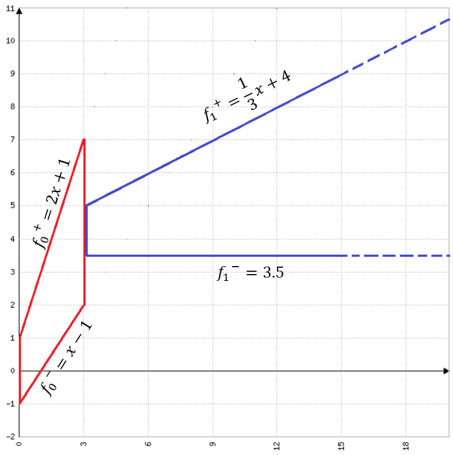


Figure 4.7: A TSF abstract element on the entire domain $\mathbb{R}^+$

which satisfy the equations 4.1 and 4.2.

Note that, if it is not specified otherwise, the abstract states of TSF refer to the entire domain $\mathbb{R}^+$. Since the conjunctions we consider are finite, this means that the last trapezoid lacks its right side (it is not closed): the upper and lower sides extend (on the right) to infinite. We can see an example in Figure 4.7, where we depict the same abstract state of Figure 4.5 but with an unrestricted domain. In Figure 4.5 instead, the domain was restricted to $[0, 10]$ and so the trapezoid on the right was closed.

4.5.4 Partial Order

The partial order $\subseteq^\sharp$ on two functions f, g in $D^\sharp$ is defined pointwisely, that is, for all possible inputs t we check that the set of values assumed by f in that point is a subset of the set of values assumed by g at the same point. Formally:

$$f \subseteq^\sharp g \Leftrightarrow \forall t \in \mathbb{R}_+ : f(t) \subseteq g(t)$$

where $f(t) = \{v : f_j^-(t) \leq v \leq f_j^+(t) \wedge t \in [t_j, t_{j+1}]\}$ and the same holds for $g(t)$.

To define the partial order on step functions, we first define a partial order on single steps. Let $\mathbf{v_i} = (f_i^-, f_i^+)$ and $\mathbf{w_j} = (g_j^-, g_j^+)$ be the values of two steps on the same domain $[a, b]$. Then:

$$\begin{aligned}
\mathbf{v_i} \sqsubseteq_{[a,b]} \mathbf{w_j} &\Leftrightarrow \forall t \in [a,b] : f_i^-(t) \geq g_j^-(t) \wedge f_i^+(t) \leq g_j^+(t) \\
&\Leftrightarrow \forall t \in [a,b] : [f_i^-(t), f_i^+(t)] \subseteq [g_j^-(t), g_j^+(t)] \\
&\Leftrightarrow [f_i^-(a), f_i^+(a)] \subseteq [g_j^-(a), g_j^+(a)] \wedge [f_i^-(b), f_i^+(b)] \subseteq [g_j^-(b), g_j^+(b)]
\end{aligned} \tag{4.3}$$

In other words, $\mathbf{v_i}$ is smaller than $\mathbf{w_j}$ if the area of the trapezoid identified by $\mathbf{v_i}$ (in the domain $[a, b]$) is contained in the area of the trapezoid identified by $\mathbf{w_j}$ (in

$[a, b]$ as well). To do this, we have to compare only the upper and lower sides of the trapezoids. Formally, we should check that $\forall t \in [a, b] : (f_i^- \geq g_j^- \wedge f_i^+ \leq g_j^+)$. Since the sides are defined by straight lines and they do not intersect, this results into checking only the values of such lines at the left and right extremes of the trapezoid.

Now we can give an effective condition for testing whether $f \subseteq^\sharp g$. Let $f = \bigwedge_{0 \leq i \leq N} \{t_i : \mathbf{v_i}\}$ and $g = \bigwedge_{0 \leq j \leq M} \{u_j : \mathbf{w_j}\}$ be two abstract states, then:

$$\begin{array}{c} f \subseteq^\sharp g \\ \Updownarrow \\ \forall (i, j) \in [0, N] \times [0, M] : [a, b] = [t_i, t_{i+1}] \cap [u_j, u_{j+1}] \neq \emptyset \Rightarrow \mathbf{v_i} \sqsubseteq_{[a,b]} \mathbf{w_j} \end{array} \tag{4.4}$$

The condition 4.4 considers all intersections between steps from the two abstract states, that is, all pairs of steps (t_i, u_j) from f and g that have a non-empty intersection ($[t_i, t_{i+1}] \cap [u_j, u_{j+1}] \neq \emptyset$). To check if two steps have an intersection ($[t_i, t_{i+1}] \cap [u_j, u_{j+1}] \neq \emptyset$), we can use the condition $u_j \leq t_{i+1} \wedge u_{j+1} \geq t_i$. Moreover, if $u_j \leq t_i$ we have that $[a, b] = [t_i, u_{j+1}]$. Otherwise, we have that $[a, b] = [u_j, t_{i+1}]$. For each intersection, we then compare the two values $(\mathbf{v_i}, \mathbf{w_j})$ in $[a, b]$.

If each step value of f is smaller than the value of every intersected step of g (with respect to their intersection on the domain), then $f \subseteq^\sharp g$.

Lemma 4.5.1 (Soundness of the partial order)**.** *If $f, g \in D^\sharp$ are normalized, then*

$$f \subseteq^\sharp g \Leftrightarrow \forall t \in \mathbb{R}_+, f(t) \subseteq g(t)$$

Proof. We distinguish the two directions:

- ($\Rightarrow$) Let $f = \bigwedge_{0 \leq i \leq N} \{t_i : \mathbf{v_i}\}$ and $g = \bigwedge_{0 \leq j \leq M} \{u_j : \mathbf{w_j}\}$, be such that $f \subseteq^\sharp g$, and let t be $\in \mathbb{R}_+$. Then there exists $i \in [0, N]$ and $j \in [0, M]$ such that $t \in [t_i, t_{i+1}] \wedge t \in [u_j, u_{j+1}]$. Thus, $[t_i, t_{i+1}] \cap [u_j, u_{j+1}] \neq \emptyset$, so $f(t) = [f_i^-(t), f_i^+(t)] \subseteq [g_j^-(t), g_j^+(t)] = g(t)$ by definition of $\sqsubseteq_{[a,b]}$.

- ($\Leftarrow$) Let $f, g \in D^\sharp$ be such that $\forall t \in \mathbb{R}_+, f(t) \subseteq g(t)$. Let $i, j \in [0, N] \times [0, M]$ be such that $[a, b] = [t_i, t_{i+1}] \cap [u_j, u_{j+1}] \neq \emptyset$, and let x be $\in [a, b]$. Since $\forall t \in \mathbb{R}_+, f(t) \subseteq g(t)$, then we also have $\forall x \in [a, b] : f(x) \subseteq g(x)$, that is $\forall x \in [a, b] : ([f_i^-(x), f_i^+(x)] \subseteq [g_j^-(x), g_j^+(x)])$. Then 4.3 we that that $\mathbf{v_i} \sqsubseteq_{[a,b]} \mathbf{w_j}$ for each possible pair (i, j) by the second line of Equation, and $f \subseteq^\sharp g$ by Equation 4.4.

□

We can see an example of the partial order relationship between two abstract states f and g on the domain $[0, 15]$ in Figure 4.8. The abstract state f is represented with straight lines, the abstract state g with dashed lines. f is defined on the set of steps $\{0, 5, 10\}$ while g is defined on the the set of steps $\{0, 4, 7, 11\}$. Then the comparison is made on the following intervals: $[0, 4], [4, 5], [5, 7], [7, 10], [10, 11], [11, 15]$. Since in each of these intervals we have that the straight lines lie within the area defined by the dashed lines, we obtain that $f \leq g$.

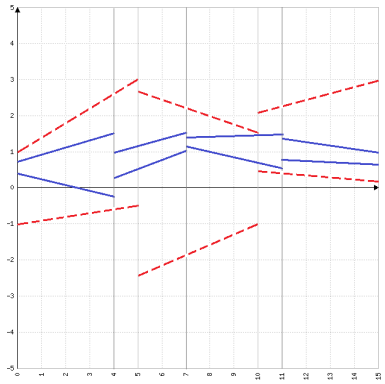


Figure 4.8: Partial order

Top and bottom

The top element of the domain is defined by:

$$\top^\sharp = 0 : [0, -\infty, 0, \infty]$$

that is, the step function with only one step with value $\mathbb{R}$, while $\bot^\sharp$ is a special element such that:

$$\gamma(\bot^\sharp) = \emptyset \wedge \forall f \in D^\sharp, \bot^\sharp \subseteq^\sharp f$$

4.5.5 *Refine* Operator

We define a *refine* operator, which, given an abstract state of TSF and a set of indices, adds these indices to the step list of the state. The concretization of the abstract state remains the same after the application of a *refine* operator, since the values $\mathbf{v_i}$ are not modified. This operator will be useful to make two abstract states directly comparable, by making them defined on the same set of steps.

Consider an abstract state $f = \bigwedge_{0 \leq i \leq N} \{t_i : \mathbf{v_i}\}$ where $T = \{t_i : 0 \leq i \leq N\}$ and a set of indices $U = \{u_j : 0 \leq j \leq M\}$. Let $S = \{s_k : \forall k \in [0, P], (s_k \in (T \cup U) \wedge s_k < s_{k+1})\}$ be the set of all the indices contained in T and U, ordered and without repetitions (therefore $P = N + M - |T \cap U|$). The *refine* operator on this state is defined by:

$$Refine(f, U) = \bigwedge_{0 \leq k \leq P} \{s_k : \widehat{\mathbf{v_k}}\}$$

where $\widehat{\mathbf{v_k}} = \mathbf{v_{max\{i : t_i \leq s_k\}}}$. Intuitively, you can see that this operator does not change the abstract information of the state. In fact we can enunciate the following lemma:

Lemma 4.5.2 (Refinement). *Given a normalized abstract state f, for any set of indices S, it holds that $f \equiv Refine(f, S)$.*

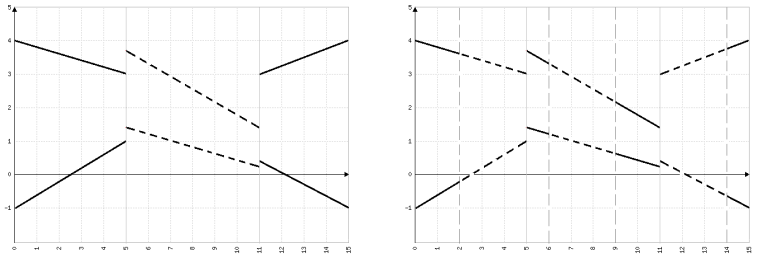


Figure 4.9: Refine operator

Proof. The equivalence relation $\equiv$ considers the normalized version of the two elements being compared. We know by hypothesis that f is already in a normalized form, so we have to compare f with $Norm(Refine(f, S))$.

Suppose that f is defined on the indices set T, that is, $\forall t_i \in T$, the step function f has a step at t_i. To underline this relationship, we write f_T instead of f. Let $Refine(f_T, S) = f'_R$ be the refined version of f_T, where $R = T \cup S$. This means that the result of the *refine* operator is a step function f' defined on the indices set $R = T \cup S$. So we can write $f_T = \bigwedge_{i \in [0,N-1]} \{t_i : \mathbf{v_i}\}$ (supposing $|T| = N$) and $f'_R = \bigwedge_{j \in [0,M-1]} \{r_j : \widehat{\mathbf{v_j}}\}$ (supposing $|R| = M$).

By definition of *refine*, we know that $\forall r_j \in (S \setminus T) \subseteq R$ the value of the step r_j in f'_R is $\widehat{\mathbf{v_j}} = \mathbf{v_{max\{i : t_i \leq r_j\}}}$. So, the value of the step r_j is the same as the one of a consecutive step already present in f_T. The normalization process removes any step which has the same value of the previous one. Then each step $r_j \in (S \setminus T)$ is removed and the result of the normalization of f'_R is a step function f''_T with step indices only from the set T (since $(T \cup S) \setminus (S \setminus T) = T$). Moreover, the value associated to each step t_i of f''_T is $\mathbf{v_i}$, the same as in f_T, since neither the *refine* operator nor the normalization process change any step value. Then f_T and f''_T are defined on the same indices set, and they have the same step values. We can conclude that $f_T = f''_T = Norm(f'_R) = Norm(Refine(f_T, S))$, thus $f = Norm(Refine(f, S))$ and finally, since $f = Norm(f)$, we have $f \equiv Refine(f, S)$ by definition of $\equiv$.

□

We can see an example of the *refine* operator in Figure 4.9. The set of steps of the original abstract state is $[0, 5, 11]$, while the set of indices I is $[2, 6, 9, 14]$. On the left we depict the original abstract state, on the right its refined version with respect to I (the vertical lines are dashed in correspondence with the indices of I).

The *refine* operator will be used when dealing with two abstract states together (for example, in the least upper bound operator) to refine both states through the set of indices of the other state. For example, if f and g are the two abstract states being considered, the preliminary step is to refine f with the steps of g, and to refine g with the steps of f. In this way, we obtain two abstract states defined on the same set of indices (that is, the union of the two original indices sets).

Note that the abstract state resulting from a refinement operation is, in general, not normalized, because it violates the second condition (i.e., two consecutive constraints cannot have equal values). This is not a problem, since the refinement is used as a *preliminary* step of various operations (glb, lub, widening, ...) but each of these operations normalizes its final result, so we are sure that we will never produce (at the end of a computation) an abstract state which is not normalized, even when we used the refinement throughout the computation.

4.5.6 Greatest Lower Bound

Given two elements x and y of the abstract domain, the greatest lower bound operator defines the greatest element $z = x \cap^\sharp y$ that under-approximates both x and y. In TSF, this means that we have to create a sequence of trapezoids that are (i) as vast as possible and (ii) contained in both the given sequences of trapezoids.

Let $f' = \bigwedge_{0 \leq k \leq N} \{x_k : \mathbf{v_k}\}$ and $g' = \bigwedge_{0 \leq j \leq M} \{u_j : \mathbf{w_j}\}$ be two abstract states defined on the step sets $X = \{x_k : k \in [0, N]\}$ and $U = \{u_j : j \in [0, M]\}$, respectively (note that $N+1 = |X|$ and $M+1 = |U|$, that is, the number of steps of an abstract state corresponds to the cardinality of its step set). Let $min(l_1(t), l_2(t), [a, b])$ be an operator which compares the lines $l_1(t)$ and $l_2(t)$ in the domain range $[a, b]$ and returns the one which is always below the other one, in the assumption that the two lines do not intersect each other in such range. Since $l_1(t)$ and $l_2(t)$ are straight lines, it suffices to compare their values in $t = a$ and $t = b$. Let $max(l_1(t), l_2(t), [a, b])$ be the opposite operator, which returns the line always above. Let $StepIndexes(f)$ be a function which, given an abstract state f, extracts its step set. Then, Algorithm 3 represents the formal definition of the greatest lower bound of f' and g' (i.e., $f' \cap^\sharp g'$).

An informal description of the algorithm is as follows:

Step 1 (lines 1-2 of the algorithm) In order to make the two abstract states f', g' directly comparable, we refine them on the same set of steps, creating two new abstract states f, g. Let $f = Refine(f', U)$ and $g = Refine(g', X)$ be these two new states. By Lemma 4.5.2, these refined states are equivalent to f' and g', so the greatest lower bound of f, g is also the greatest lower bound of f', g'.

Step 2 Let $T = \{t_i : i \in [0, P-1]\}$ (where $P = |X \cup U| = |T|$) be the set of steps indices of f and g (line 3). Let $(f_i^-, f_i^+), (g_i^-, g_i^+)$ be the values (i.e., lower and upper sides) of f, g in the generic step $[t_i, t_{i+1}]$. We will split each step $[t_i, t_{i+1}]$ into sub-steps, with respect to intersections of the sides of the two trapezoids (one from f, one from g, lines 10-26). If there are no intersections, we leave the step unchanged (line 25). Each step $[t_i, t_{i+1}]$ will then generate one (or more) steps in $f \cap^\sharp g$, depending on the intersections of the sides in such step. The goal is to obtain sub-steps in which the sides of the two trapezoids do not intersect each other in any point *inside* the sub-step range.

Algorithm 3 Pseudocode algorithm for glb computation

1: $f \leftarrow Refine(f', U)$
2: $g \leftarrow Refine(g', X)$
3: $T \leftarrow StepIndexes(f) = \{t_i\}$ //note that $StepIndexes(f) = StepIndexes(g)$
4: $P \leftarrow |T|$
5: $result \leftarrow \emptyset$
6: **for** i from 0 to $P-1$ **do**
7: **if** f_i^- intersects $g_i^+ \vee f_i^+$ intersects g_i^- in $]t_i..t_{i+1}[$ **then**
8: **return** $\bot^\sharp$
9: **end if**
10: **if** f_i^- intersects g_i^- in $t_A \in]t_i..t_{i+1}[$ **then**
11: **if** f_i^+ intersects g_i^+ in $t_B \in]t_i..t_{i+1}[$ **then**
12: **if** $t_A < t_B$ **then**
13: $resSteps \leftarrow t_i : (\max(f_i^-, g_i^-, [t_i..t_A]), \min(f_i^+, g_i^+, [t_i..t_A])) \wedge \wedge t_A : (\max(f_i^-, g_i^-, [t_A..t_B]), \min(f_i^+, g_i^+, [t_A..t_B])) \wedge \wedge t_B : (\max(f_i^-, g_i^-, [t_B..t_{i+1}]), \min(f_i^+, g_i^+, [t_B..t_{i+1}]))$
14: **else**
15: //the same but reversing t_A and t_B
16: **end if**
17: **else**
18: $resSteps \leftarrow t_i : (\max(f_i^-, g_i^-, [t_i..t_A]), \min(f_i^+, g_i^+, [t_i..t_A])) \wedge \wedge t_A : (\max(f_i^-, g_i^-, [t_A..t_{i+1}]), \min(f_i^+, g_i^+, [t_A..t_{i+1}]))$
19: **end if**
20: **else if** f_i^+ intersects g_i^+ in $t_B \in]t_i..t_{i+1}[$ **then**
21: $resSteps \leftarrow t_i : (\max(f_i^-, g_i^-, [t_i..t_B]), \min(f_i^+, g_i^+, [t_i..t_B])) \wedge \wedge t_B : (\max(f_i^-, g_i^-, [t_B..t_{i+1}]), \min(f_i^+, g_i^+, [t_B..t_{i+1}]))$
22: **else if** $[f_i^-(t), f_i^+(t)] \cap [g_i^-(t), g_i^+(t)] = \emptyset$ in $t = t_i \vee t = t_{i+1}$ **then**
23: **return** $\bot^\sharp$
24: **else**
25: $resSteps \leftarrow t_i : (\max(f_i^-, g_i^-, [t_i..t_{i+1}]), \min(f_i^+, g_i^+, [t_i..t_{i+1}]))$
26: **end if**
27: $result \leftarrow result \wedge resSteps$
28: **end for**
29: **if** $\exists s_i \in StepIndexes(result) : [result_i^-(s_{i+1}), result_i^+(s_{i+1})] \cap [result_{i+1}^-(s_{i+1}), result_{i+1}^+(s_{i+1})] = \emptyset$ //i.e., result does not satisfy the second validity condition **then**
30: **return** $\bot^\sharp$
31: **else**
32: **return** $Norm(result)$
33: **end if**

By Equation 4.1 (first validity condition), we know that, *inside* a step, the upper and lower side of a trapezoid do not intersect. In fact, the two sides could have an extreme in common (the value at t_i or at t_{i+1}) or they could be the same line, but they surely do not have an intersection point inside $[t_i, t_{i+1}]$, otherwise they would violate the first constraint (Equation 4.1). For this reason, we know for sure that f_i^-, f_i^+ do not intersect each other, and the same holds for g_i^-, g_i^+. The possible intersections inside (t_i, t_{i+1}) (extremes excluded) are then: (1) between f_i^- and g_i^-, (2) between f_i^+ and g_i^+, (3) between f_i^- and g_i^+, and (4) between f_i^+ and g_i^-.

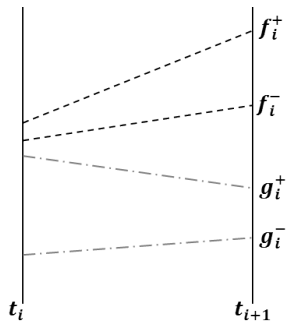
In case 1 and 2 (intersection between f_i^-, g_i^- or between f_i^+, g_i^+, lines 9-21), we split the step in sub-steps with respect to the intersection point. Note that we could obtain two or three sub-steps: if only f_i^-, g_i^- intersect each other (lines 17-18) then we have two sub-steps (and the same happens if the only intersection regards f_i^+, g_i^+, lines 20-21 - see Figure 4.10(d)) but if there are two intersections (one between f_i^-, g_i^- and the other between f_i^+, g_i^+, lines 12-16 - see Figure 4.10(e)) then we obtain three sub-steps.

In case 3 and 4 (intersection between f_i^-, g_i^+ - Figure 4.10(b) - or between f_i^+, g_i^- - Figure 4.10(c), lines 7-8), instead of splitting the step with respect to the intersection point, we immediately return $\bot^\sharp$ as result of the $\cap^\sharp$ operation. In fact, assume that f_i^-, g_i^+ have an intersection (the same reasoning holds if the intersection is between f_i^+, g_i^-). Then, at one extreme of the step (i.e., at t_i or t_{i+1}), the lower side of the trapezoid of f (f_i^-) is greater than the upper side of the trapezoid of g (g_i^+). This means that, at such point, the two states have no values in common since the areas of the trapezoids do not intersect: the lowest value assumed by the continuous functions abstracted by f is greater than the greatest value assumed by the continuous functions abstracted by g. The result of the greatest lower bound must therefore be $\bot^\sharp$.

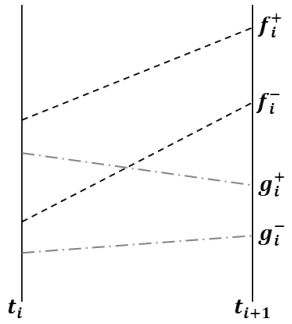
If the step $[t_i, t_{i+1}]$ does not contain any intersection, two cases apply:

- if $[f_i^-(t_i), f_i^+(t_i)] \cap [g_i^-(t_i), g_i^+(t_i)] = \emptyset \vee [f_i^-(t_{i+1}), f_i^+(t_{i+1})] \cap [g_i^-(t_{i+1}), g_i^+(t_{i+1})] = \emptyset$ (Figure 4.10(a), lines 22-23) then we return $\bot^\sharp$, because it means that at one extreme of the step (i.e., at t_i or t_{i+1}) the lowest value of one state is greater than the greatest value of the other one.
- otherwise, we do not need to split the step (line 25), since we are in the case sketched by Figure 4.10(f).

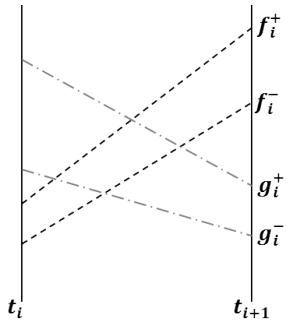
Step 3 We know for sure that, in each of the sub-steps generated by the algorithm: (i) the two states have some values in common (i.e., the areas of the trapezoids have a non-empty intersection) at each point of the sub-step; (ii) inside the sub-step the four sides of the two trapezoids derived from f and g do not have any intersection. Then, for each sub-step it is easy to compute the value of the corresponding trapezoid of the result ($f \cap^\sharp g$): the lower side will correspond



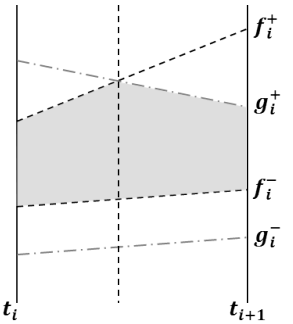
(a) No intersections, since f_i^- is above g_i^+, so the result of the glb operation is $\bot^\sharp$



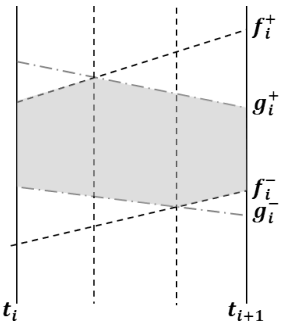
(b) One intersection between f_i^- and g_i^+, so the result of the glb operation is $\bot^\sharp$



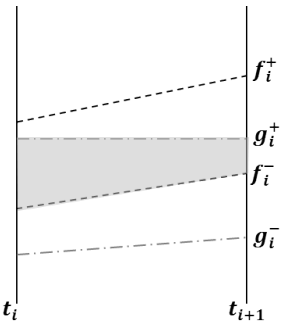
(c) Four intersections; the result of the glb operation is $\bot^\sharp$ because f_i^+ intersects g_i^-



(d) One intersection between f_i^+ and g_i^+, the step is split into two sub-steps. The gray area represents the two resulting trapezoids.



(e) Two intersections, one between f_i^-, g_i^- and one between f_i^+, g_i^+. The step is split into three sub-steps. The gray area represents the three resulting trapezoids.



(f) No intersections, the step is not split into sub-steps. The gray area represents the resulting trapezoid.

Figure 4.10: Examples of the glb sub-step splitting

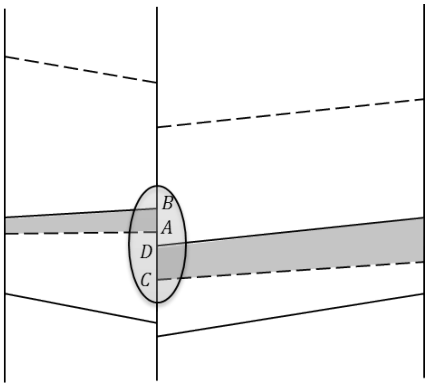


Figure 4.11: The glb does not always preserve the second validity condition

to the greatest of the two lower sides of f, g in the sub-step, while the upper side will correspond to the lowest of the two upper sides of f, g in the sub-step. In Figures 4.10(d), 4.10(e) and 4.10(f) we can see the result of this procedure in three different cases (step split into two sub-steps, step split into three sub-steps, step not split, respectively). The gray area represents the resulting trapezoids of $f \cap^\sharp g$. We add the newly generated sub-steps to the result at line 27 (at the end of each iteration of the loop).

Step 4 At the end of the computation (lines 29-33) we normalize the resulting abstract state using *Norm* (line 32). In the abstract state $f \cap^\sharp g$, it could happen that the intervals individuated by the two sides of two consecutive steps do not intersect at the border between the two steps (for an example, see Figure 4.11, where the straight lines represent the trapezoids of f, the dashed lines represent the trapezoids of g and the darkened areas represent the trapezoids of $f \cap^\sharp g$). For this reason, after computing the greatest lower bound $h = f \cap^\sharp g$, we perform a check that h respects the second validity condition (that is, the situation of Figure 4.11 never happens). If the check fails, we return $\bot^\sharp$ as result of the glb operation (lines 29-30). This happens because, if one of such border intersections is empty, then the two functions f and g do not have any possible value in common at that border and thus their lower bound in the TSF domain does not exist.

In conclusion, the intersection $f \cap^\sharp g$ creates a new step function whose value is at every time t the intersection $f(t) \cap g(t)$. If this intersection is empty in at least one point, we cannot return a continuous abstract function and then we define $f \cap^\sharp g$ as $\bot^\sharp$, the bottom element of $D^\sharp$.

Lemma 4.5.3 ($\cap^\sharp$ maintains the validity conditions)**.** *Let f and g be two abstract states which both respect the validity conditions enunciated in Section 4.5 and which are defined on the same set of indices $T = \{t_i : i \in [0, N]\}$ (if that is not the case,*

we can use the refine operator). Let $f \cap^\sharp g = h$ be their greatest lower bound. Then, h respects the two validity conditions.

Proof. If $h = \bot^\sharp$, then it automatically respects the validity conditions ($\bot^\sharp$ is a valid element of the domain). Now assume $h \neq \bot^\sharp$ and $h = \bigwedge_{j\in[0,M]}\{x_j : \mathbf{v_j} = (h_j^-, h_j^+)\}$. By construction of h, we know that $T \subseteq X = \{x_j : j \in [0, M]\}$, that is, the steps indices of f and g are a subset of those of h. In fact, h has a number of steps which is greater or equal to the one of f and g, since each step of f and g generates from one to three steps in h, depending on the sides intersections (lines 13, 15, 18, 21 and 25).

The validity conditions applied to h are the following ones:

1. $\forall j \in [0, M], h_j^-(t_j) \leq h_j^+(t_j) \wedge h_j^-(t_{j+1}) \leq h_j^+(t_{j+1})$
2. $\forall j \in [0, M-1], [h_j^-(t_{j+1}), h_j^+(t_{j+1})] \cap [h_{j+1}^-(t_{j+1}), h_{j+1}^+(t_{j+1})] \neq \emptyset$

1. Consider the generic step $[x_j, x_{j+1}]$, where $j \in [0, M]$ and (h_j^-, h_j^+) is the value of h in such step. By construction of h, we know that each step range $[x_j, x_{j+1}]$ of h is a subset of a step range from f, g, that is, $\exists i : [x_j, x_{j+1}] \subseteq [t_i, t_{i+1}]$ (lines 13, 15, 18, 21 and 25). Let $(f_i^-, f_i^+), (g_i^-, g_i^+)$ be the values of f and g in such step, respectively. By construction, we also know that the lines $f_i^-, f_i^+, g_i^-, g_i^+$ do not intersect each other inside $[x_j, x_{j+1}]$ (lines 7,10,11,20). Regarding (h_j^-, h_j^+), we know that h_j^- corresponds to f_i^- or g_i^-, and that h_j^+ corresponds to f_i^+ or g_i^+ (since in the algorithm we always use the functions $max(\cdot,\cdot,\cdot)$ and $min(\cdot,\cdot,\cdot)$ which return one of the first two inputs - lines 13, 15, 18, 21 and 25). If both sides h_j^-, h_j^+ correspond to sides from the same state (i.e., $h_j^- = f_i^- \wedge h_j^+ = f_i^+$ or $h_j^- = g_i^- \wedge h_j^+ = g_i^+$), then the first validity condition is satisfied, because both f and g satisfy it. Otherwise, either $h_j^- = f_i^- \wedge h_j^+ = g_i^+$ or $h_j^- = g_i^- \wedge h_j^+ = f_i^+$. Suppose that $h_j^- = g_i^- \wedge h_j^+ = f_i^+$ (the other case is symmetric). Then, to prove that h satisfies the first validity condition, we must prove that $f_i^+(x_j) \geq g_i^-(x_j) \wedge f_i^+(x_{j+1}) \geq g_i^-(x_{j+1})$:

   - To prove $f_i^+(x_j) \geq g_i^-(x_j)$, for the sake of a contradiction, suppose that $f_i^+(x_j) < g_i^-(x_j)$. Then, the result of the glb operation would have been $\bot^\sharp$ (because $[f_i^-(x_j), f_i^+(x_j)] \cap [g_i^-(x_j), g_i^+(x_j)] = \emptyset$), which is in contradiction with the hypothesis that $h \neq \bot^\sharp$.
   - To prove $f_i^+(x_{j+1}) \geq g_i^-(x_{j+1})$, the reasoning is symmetrical.

2. The second validity condition is automatically respected by each output of the glb computation, since we return $\bot^\sharp$ if the property is not satisfied (lines 29-30).

□

Lemma 4.5.4 ($\cap^\sharp$ is the greatest lower bound operator)**.** *Let f, g be two abstract states and let $h = f \cap^\sharp g$ be their greatest lower bound. Then:*

1. $h \subseteq^\sharp f \wedge h \subseteq^\sharp g$
2. $k \subseteq^\sharp h \; \forall k$ *lower bound of* f *and* g

Proof. Let us first assume that $h \neq \bot^\sharp$ (afterwards we will consider also the case where $h = \bot^\sharp$). Also, let f, g, h be defined on the same set of steps $T = \{t_i : i \in [0, N]\}$. We can obtain this situation by refining the three abstract states on the union of their step sets through *refine*. By Lemma 4.5.2 we know that an abstract state and its refined version are equivalent, so the refinement does not change the order relationship between states. For this reason, in all our proofs we can use indifferently either the original abstract state or its refined version.

1. Since f, g, h are defined on the same set of indices T, we can write $f = \bigwedge_{i\in[0,N]}\{t_i : \mathbf{f_i}\}$, $g = \bigwedge_{i\in[0,N]}\{t_i : \mathbf{g_i}\}$ and $h = \bigwedge_{i\in[0,N]}\{t_i : \mathbf{h_i}\}$. By definition of $\subseteq^\sharp$, we have that $h \subseteq^\sharp f \wedge h \subseteq^\sharp g \Leftrightarrow \forall i \in [0, N] : \mathbf{h_i} \sqsubseteq_{[t_i,t_{i+1}]} \mathbf{f_i} \wedge \mathbf{h_i} \sqsubseteq_{[t_i,t_{i+1}]} \mathbf{g_i}$. Let $[t_i, t_{i+1}]$ be a generic step, where $i \in [0, N]$. By construction of h, we know that f_i^+ and g_i^+ do not intersect each other in $[t_i, t_{i+1}]$, as well as f_i^- and g_i^- do not intersect (lines 10,11,20). Then:
   - For the upper line, we know, by construction, that h_i^+ is the lowest between the two non-intersecting sides f_i^+ and g_i^+ (lines 13, 15, 18, 21 and 25, where we use $min(f_i^+, g_i^+, \cdot)$), so $\forall t \in [t_i, t_{i+1}], h_i^+(t) \leq f_i^+(t) \wedge h_i^+(t) \leq g_i^+(t)$.
   - For the lower line, we know, by construction, that h_i^- is the greatest between the two not-intersecting sides f_i^- and g_i^- (lines 13, 15, 18, 21 and 25, where we use $max(f_i^-, g_i^-, \cdot)$), so $\forall t \in [t_i, t_{i+1}], f_i^-(t) \leq h_i^-(t) \wedge g_i^-(t) \leq h_i^-(t)$.
   - From $\forall t \in [t_i, t_{i+1}], h_i^+(t) \leq f_i^+(t)$ and from $\forall t \in [t_i, t_{i+1}], f_i^-(t) \leq h_i^-(t)$, by definition of $\mathbf{u} \sqsubseteq_{[a,b]} \mathbf{v}$ (Equation 4.3), it follows that $\mathbf{h_i} \sqsubseteq_{[t_i,t_{i+1}]} \mathbf{f_i}$. From $\forall t \in [t_i, t_{i+1}], h_i^+(t) \leq g_i^+(t) \wedge g_i^-(t) \leq h_i^-(t)$, it follows that $\mathbf{h_i} \sqsubseteq_{[t_i,t_{i+1}]} \mathbf{g_i}$.
   - Then, $\forall i \in I, \mathbf{h_i} \sqsubseteq_{[t_i,t_{i+1}]} \mathbf{f_i} \wedge \mathbf{h_i} \sqsubseteq_{[t_i,t_{i+1}]} \mathbf{g_i}$ and so we conclude that $h \subseteq^\sharp f \wedge h \subseteq^\sharp g$ by definition of $\subseteq^\sharp$.
2. Let f, g, h be defined on the same set of indices T. By contradiction, assume that $\exists k : k$ is a lower bound of f and g, and $k \not\subseteq^\sharp h$. Let us suppose that k is defined on T (we can always obtain it by applying *refine*). This means that $\exists i \in I : \mathbf{k_i} \not\sqsubseteq_{[t_i,t_{i+1}]} \mathbf{h_i}$. Since $h_i^-, h_i^+, k_i^-, k_i^+$ are straight lines, and by the definition of $\sqsubseteq_{[a,b]}$ (Equation 4.3), at least one of the following equations must be satisfied (otherwise it would hold $\mathbf{k_i} \sqsubseteq_{[t_i,t_{i+1}]} \mathbf{h_i}$):
   - $k_i^+(t_i) > h_i^+(t_i)$
   - $k_i^+(t_{i+1}) > h_i^+(t_{i+1})$

- $k_i^-(t_i) < h_i^-(t_i)$
- $k_i^-(t_{i+1}) < h_i^-(t_{i+1})$

Suppose that $k_i^+(t_i) > h_i^+(t_i)$ is true (the reasoning is symmetrical for the other three equations). By construction of h, we know that $h_i^+ = f_i^+ \vee h_i^+ = g_i^+$ (lines 13, 15, 18, 21 and 25, where $h_i^+ = min(f_i^+, g_i^+, \cdot)$). So we can rewrite $k_i^+(t_i) > h_i^+(t_i)$ as $k_i^+(t_i) > f_i^+(t_i) \vee k_i^+(t_i) > g_i^+(t_i)$. If $k_i^+(t_i) > f_i^+(t_i)$ holds, then $\mathbf{k_i} \not\sqsubseteq_{[t_i,t_{i+1}]} \mathbf{f_i}$, and this implies that $k \not\subseteq^\sharp f$. We have reached a contradiction, because k is a lower bound of f. On the other hand, if $k_i^+(t_i) > g_i^+(t_i)$ holds, then $\mathbf{k_i} \not\sqsubseteq_{[t_i,t_{i+1}]} \mathbf{g_i}$, and this implies that $k \not\subseteq^\sharp g$. We have reached a contradiction in this case, too, because k is also a lower bound of g.

Assume now that $h = \bot^\sharp$.

1. This part of the proof is trivial, since $\bot^\sharp \subseteq^\sharp f \wedge \bot^\sharp \subseteq^\sharp g$ by definition of $\bot^\sharp$.
2. If $h = \bot^\sharp$, then one between the following two facts must be true:
   (a) in some point t of the domain, the two trapezoids of f, g did not have any point in common: $[f_i^-(t), f_i^+(t)] \cap [g_i^-(t), g_i^+(t)] = \emptyset$ (supposing that $t \in [t_i, t_{i+1}]$); this means that $f_i^-(t) > g_i^+(t) \vee g_i^-(t) > f_i^+(t)$ (lines 8 and 23)
   (b) the result of the glb computation did not respect the second validity condition, because at the border between two steps $[t_i, t_{i+1}]$ and $[t_{i+1}, t_{i+2}]$ the two trapezoids of h did not have any point in common: $[h_i^-(t_{i+1}), h_i^+(t_{i+1})] \cap [h_{i+1}^-(t_{i+1}), h_{i+1}^+(t_{i+1})] = \emptyset$ (lines 29-30)

   Consider case (a), and suppose $k \neq \bot^\sharp$. Since k is a lower bound of f, g, it must respect the following condition: $[k_i^-(t), k_i^+(t)] \subseteq [f_i^-(t), f_i^+(t)] \wedge [k_i^-(t), k_i^+(t)] \subseteq [g_i^-(t), g_i^+(t)]$ throughout all the domain and in particular in the point t where, by hypothesis, $f_i^-(t) > g_i^+(t) \vee g_i^-(t) > f_i^+(t)$. Such condition implies that: $k_i^-(t) \geq f_i^-(t) \wedge k_i^+(t) \leq f_i^+(t) \wedge k_i^-(t) \geq g_i^-(t) \wedge k_i^+(t) \leq g_i^+(t)$. Now, if $f_i^-(t) > g_i^+(t)$, considering that $k_i^-(t) \geq f_i^-(t)$, we obtain $k_i^-(t) > g_i^+(t)$. This fact, combined with $k_i^+(t) \geq k_i^-(t)$ (for the first validity condition of abstract states), results in $k_i^+(t) > g_i^+(t)$, which cannot hold, because $k_i^+(t) \leq g_i^+(t)$. We obtained a contradiction. A similar reasoning can be done when $g_i^-(t) > f_i^+(t)$ (instead of $f_i^-(t) > g_i^+(t)$). Since we reached a contradiction, k must be $\bot^\sharp$ and then $\bot^\sharp = k \subseteq^\sharp h = \bot^\sharp$.

   Now consider case (b) (the result of the glb did not respect the second validity condition - line 29). We will prove that any other lower bound k of f and g does not respect the second validity condition, either. For the sake of a contradiction, suppose that $\exists k : k$ is a lower bound of f and g and $k \neq \bot^\sharp$. The hypothesis $[h_i^-(t_{i+1}), h_i^+(t_{i+1})] \cap [h_{i+1}^-(t_{i+1}), h_{i+1}^+(t_{i+1})] = \emptyset$ (violation of

the second validity condition by h) can be rewritten, by construction of h (lines 13, 15, 18, 21 and 25, where we assign $h_i^- = max(f_i^-, g_i^-, \cdot)$ and $h_i^+ = min(f_i^+, g_i^+, \cdot)$), as $[A, B] \cap [C, D] = \emptyset$ where $A = \max(f_i^-(t_{i+1}), g_i^-(t_{i+1}))$, $B = \min(f_i^+(t_{i+1}), g_i^+(t_{i+1}))$, $C = \max(f_{i+1}^-(t_{i+1}), g_{i+1}^-(t_{i+1}))$, $D = \min(f_{i+1}^+(t_{i+1}), g_{i+1}^+(t_{i+1}))$ (see also Figure 4.11 for the notation of A, B, C, D). This means that $B < C \vee A > D$. Suppose that $A > D$, exactly like it happens in Figure 4.11 (the reasoning is symmetrical if $B < C$). The abstract state k, being a lower bound, must be less or equal than f and g. So, for example, $k_i^-(t_{i+1})$ must be greater or equal than $f_i^-(t_{i+1})$ and $g_i^-(t_{i+1})$. Thus, $k_i^-(t_{i+1}) \geq A$. For the same reason (k is a lower bound of f, g), $k_{i+1}^+(t_{i+1})$ must be less or equal than both $f_{i+1}^+(t_{i+1})$ and $g_{i+1}^+(t_{i+1})$: thus, $k_{i+1}^+(t_{i+1}) \leq D$. Then, since we know that $A > D$, by transitivity we have $k_{i+1}^+(t_{i+1}) < k_i^-(t_{i+1})$, and this violates the second validity condition. The TSF domain does not include abstract states which do not respect such condition, so k becomes $\bot^\sharp$. Trivially, it follows $\bot^\sharp = k \subseteq^\sharp h = \bot^\sharp$.

□

4.5.7 Least Upper Bound

Given two elements x and y of the abstract domain, the least upper bound operator defines the least element z that over-approximates both x and y. In TSF, this means that we have to create a sequence of trapezoids that are as narrow as possible and that, at the same time, contain the two given sequences of trapezoids.

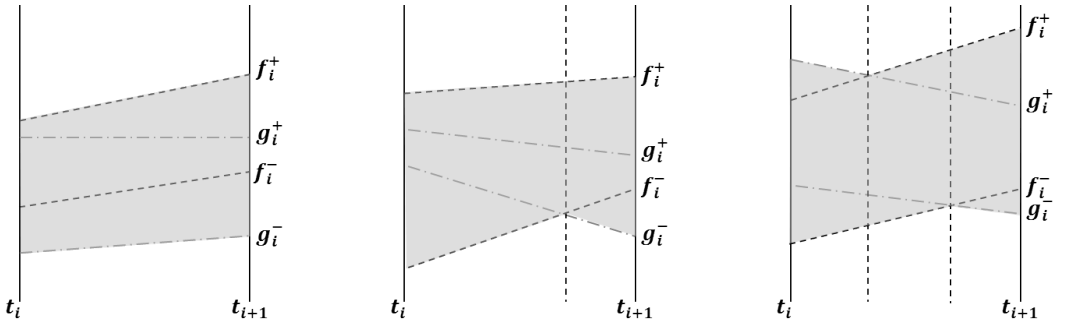
Let $f' = \bigwedge_{0 \leq k \leq N} \{x_k : \mathbf{v_k}\}$ and $g' = \bigwedge_{0 \leq j \leq M} \{u_j : \mathbf{w_j}\}$ be two abstract states. In order to define the least upper bound of f' and g', we use an algorithm very similar to the one presented for the glb in Section 4.5.6. First, we refine f' and g' on the same set of steps, obtaining $f = Refine(f', U)$ and $g = Refine(g', X)$ where X and U are the step set of f' and g', respectively. Then, for each step of f and g we look at the two trapezoids and check if there are intersections either between the two lower sides (f_i^-, g_i^-) or between the two upper sides (f_i^+, g_i^+). We split the step with respect to such intersections; if there are none, the step remains unsplit. In each of these new steps, we are sure that neither the upper sides nor the lower sides intersect each other. So, the resulting trapezoid for each new step is made by the greatest of the upper sides and the lowest of the lower sides. For some examples, see Figure 4.12. We formalize the procedure in Algorithm 4.

The algorithm is very similar to the one introduced for the glb, with the following differences:

- min and max are reversed, since here we keep the lowest line between f_i^-, g_i^- and the greatest line between f_i^+, g_i^+ in order to over-approximate f_i and g_i.
- we never return $\bot^\sharp$ because the lub between two values is always possible and it also satisfies, by construction, the validity conditions (see also Lemma 4.5.6 for the formal proof).

Algorithm 4 Pseudo-code algorithm for lub computation

1: $f \leftarrow Refine(f', U)$
2: $g \leftarrow Refine(g', X)$
3: $T \leftarrow StepIndexes(f) = \{t_i\}$ // StepIndexes(f) = StepIndexes(g)
4: $P \leftarrow |T|$
5: $result \leftarrow \emptyset$
6: **for** i from 0 to $P-1$ **do**
7: **if** f_i^- intersects g_i^- in $t_A \in]t_i..t_{i+1}[$ **then**
8: **if** f_i^+ intersects g_i^+ in $t_B \in]t_i..t_{i+1}[$ **then**
9: **if** $t_A < t_B$ **then**
10: $resSteps \leftarrow t_i : (\min(f_i^-, g_i^-, [t_i..t_A]), \max(f_i^+, g_i^+, [t_i..t_A])) \wedge \wedge t_A : (\min(f_i^-, g_i^-, [t_A..t_B]), \max(f_i^+, g_i^+, [t_A..t_B])) \wedge \wedge t_B : (\min(f_i^-, g_i^-, [t_B..t_{i+1}]), \max(f_i^+, g_i^+, [t_B..t_{i+1}]))$
11: **else**
12: //the same but reversing t_A and t_B
13: **end if**
14: **else**
15: $resSteps \leftarrow t_i : (\min(f_i^-, g_i^-, [t_i..t_A]), \max(f_i^+, g_i^+, [t_i..t_A])) \wedge \wedge t_A : (\min(f_i^-, g_i^-, [t_A..t_{i+1}]), \max(f_i^+, g_i^+, [t_A..t_{i+1}]))$
16: **end if**
17: **else if** f_i^+ intersects g_i^+ in $t_B \in]t_i..t_{i+1}[$ **then**
18: $resSteps \leftarrow t_i : (\min(f_i^-, g_i^-, [t_i..t_B]), \max(f_i^+, g_i^+, [t_i..t_B])) \wedge \wedge t_B : (\min(f_i^-, g_i^-, [t_B..t_{i+1}]), \max(f_i^+, g_i^+, [t_B..t_{i+1}]))$
19: **else**
20: $resSteps \leftarrow t_i : (\min(f_i^-, g_i^-, [t_i..t_{i+1}]), \max(f_i^+, g_i^+, [t_i..t_{i+1}]))$
21: **end if**
22: $result \leftarrow result \wedge resSteps$
23: **end for**
24: **return** Norm(result)



(a) No intersections, the step remains unsplit, resulting in one trapezoid (colored in gray)

(b) One intersection between f_i^- and g_i^-, resulting in two sub-steps and, thus, two trapezoids (colored in gray)

(c) Two intersections, one between f_i^-, g_i^- and one between f_i^-, g_i^-, resulting in three sub-steps and, thus, three trapezoids (colored in gray)

Figure 4.12: Examples of the lub sub-step splitting

Lemma 4.5.5 ($\cup^\sharp$ is the least upper bound operator)**.** *$\cup^\sharp$ is a least upper bound operator. Let $h = f \cup^\sharp g$ be the least upper bound of f and g. Then:*

1. $f \subseteq^\sharp h \wedge g \subseteq^\sharp h$
2. $h \subseteq^\sharp k \; \forall k$ *upper bound of* f *and* g

Proof. Let f, g, h be defined on the same set of steps $T = \{t_i : i \in [0, N]\}$: $f = \bigwedge_{i \in I}\{t_i : \mathbf{f_i}\}, g = \bigwedge_{i \in I}\{t_i : \mathbf{g_i}\}$ and $h = \bigwedge_{i \in I}\{t_i : \mathbf{h_i}\}$. We can obtain this situation by refining the three abstract states on the union of their step sets.

1. By definition of $\subseteq^\sharp$, we have that $f \subseteq^\sharp h \wedge g \subseteq^\sharp h \Leftrightarrow \forall i \in [0, N] : \mathbf{f_i} \sqsubseteq_{[t_i, t_{i+1}]} \mathbf{h_i} \wedge \mathbf{g_i} \sqsubseteq_{[t_i, t_{i+1}]} \mathbf{h_i}$. Let $[t_i, t_{i+1}]$ be a generic step, where $i \in [0, N]$. By construction of h, we know that f_i^+ and g_i^+ do not intersect each other in $[t_i, t_{i+1}]$, as well as f_i^- and g_i^- do not intersect (lines 7, 8, 17). Then:

   - For the upper line, we know by construction that h_i^+ is the greatest between the two non-intersecting sides f_i^+ and g_i^+ (lines 10, 12, 15, 18, 20), so $\forall t \in [t_i, t_{i+1}], h_i^+(t) \geq f_i^+(t) \wedge h_i^+(t) \geq g_i^+(t)$.
   - For the lower line, we know by construction that h_i^- is the lowest between the two non-intersecting sides f_i^- and g_i^- (lines 10, 12, 15, 18, 20), so $\forall t \in [t_i, t_{i+1}], h_i^-(t) \leq f_i^-(t) \wedge h_i^-(t) \leq g_i^-(t)$.
   - From $\forall t \in [t_i, t_{i+1}], h_i^+(t) \geq f_i^+(t)$ and from $\forall t \in [t_i, t_{i+1}], h_i^-(t) \leq f_i^-(t)$, by definition of $\mathbf{u} \sqsubseteq_{[a,b]} \mathbf{v}$ (Equation 4.3), it follows that $\mathbf{f_i} \sqsubseteq_{[t_i, t_{i+1}]} \mathbf{h_i}$. From $\forall t \in [t_i, t_{i+1}], h_i^+(t) \geq g_i^+(t) \wedge h_i^-(t) \leq g_i^-(t)$, it follows that $\mathbf{g_i} \sqsubseteq_{[t_i, t_{i+1}]} \mathbf{h_i}$.

- Then, $\forall i \in I, \mathbf{f_i} \sqsubseteq_{[t_i,t_{i+1}]} \mathbf{h_i} \wedge \mathbf{g_i} \sqsubseteq_{[t_i,t_{i+1}]} \mathbf{h_i}$ and so we conclude that $f \subseteq^\sharp h \wedge g \subseteq^\sharp h$ by definition of $\subseteq^\sharp$.

2. Assume that k is defined on the same set of steps $T = \{t_i : i \in [0, N]\}$ as that of f, g, h (as said before, if this is not true, we can make it true by refining each abstract state on the union of the four step sets, maintaining all the relationships among such states). By contradiction, assume that $h \not\subseteq^\sharp k$, that is $\exists i \in [0, N] : \mathbf{h_i} \not\sqsubseteq_{[t_i,t_{i+1}]} \mathbf{k_i}$. Since $h_i^-, h_i^+, k_i^-, k_i^+$ are straight lines, and by the definition of $\sqsubseteq_{[a,b]}$ (Equation 4.3), at least one of the following equations must be true (otherwise it would hold $\mathbf{h_i} \sqsubseteq_{[t_i,t_{i+1}]} \mathbf{k_i}$):

   - $k_i^+(t_i) < h_i^+(t_i)$
   - $k_i^+(t_{i+1}) < h_i^+(t_{i+1})$
   - $k_i^-(t_i) > h_i^-(t_i)$
   - $k_i^-(t_{i+1}) > h_i^-(t_{i+1})$

   Suppose that $k_i^+(t_i) < h_i^+(t_i)$ is true (the reasoning is symmetrical for the other three equations). By construction of h, we know that $h_i^+ = f_i^+ \vee h_i^+ = g_i^+$ (lines 10, 12, 15, 18, 20, where $h_i^+ = max(f_i^+, g_i^+, \cdot)$). So we can rewrite $k_i^+(t_i) < h_i^+(t_i)$ as $k_i^+(t_i) < f_i^+(t_i) \vee k_i^+(t_i) < g_i^+(t_i)$. If $k_i^+(t_i) < f_i^+(t_i)$ holds, then $\mathbf{f_i} \not\sqsubseteq_{[t_i,t_{i+1}]} \mathbf{k_i}$, and this implies that $f \not\subseteq^\sharp k$. We have reached a contradiction, because k is an upper bound of f. On the other hand, if $k_i^+(t_i) < g_i^+(t_i)$ holds, then $\mathbf{g_i} \not\sqsubseteq_{[t_i,t_{i+1}]} \mathbf{k_i}$, and this implies that $g \not\subseteq^\sharp k$. We have reached a contradiction in this case, too, because k is also an upper bound of g.

□

Lemma 4.5.6 ($\cup^\sharp$ maintains the validity conditions)**.** *Let f, g be two abstract states which both respect the validity conditions enunciated in Section 4.5 and which are defined on the same set of indices $T = \{t_i : i \in [0, N]\}$ (if it is not the case, we can use the refine operator). Let $f \cup^\sharp g = h$ be their least upper bound. Then, the abstract state h satisfies the two validity conditions.*

Proof. Let $h = \bigwedge_{j \in [0,M]} \{x_j : \mathbf{v_j} = (h_j^-, h_j^+)\}$ be the lub of f, g. By construction of h, we know that $T \subseteq X = \{x_j : j \in [0, M]\}$, that is, the steps indices of f and g are a subset of those of h. In fact, h has a number of steps which is greater or equal to the one of f, g, since each step of f, g generates from one to three steps in h, depending on the sides intersections (lines 10, 12, 15, 18, 20). The validity conditions applied to h are the following ones:

1. $\forall j \in [0, M], h_j^-(x_j) \leq h_j^+(x_j) \wedge h_j^-(x_{j+1}) \leq h_j^+(x_{j+1})$
2. $\forall j \in [0, M-1], [h_j^-(x_{j+1}), h_j^+(x_{j+1})] \cap [h_{j+1}^-(x_{j+1}), h_{j+1}^+(x_{j+1})] \neq \emptyset$

1. Consider the generic step $[x_j, x_{j+1}]$, where $j \in [0, M]$ and (h_j^-, h_j^+) is the value of h in such step. By construction of h, we know that each step range $[x_j, x_{j+1}]$ of h is a subset of a step range from f, g, that is, $\exists i : [x_j, x_{j+1}] \subseteq [t_i, t_{i+1}]$ (lines 10, 12, 15, 18, 20). Let $(f_i^-, f_i^+), (g_i^-, g_i^+)$ be the values of f and g in such step, respectively. Regarding (h_j^-, h_j^+), we know that h_j^- corresponds to the lowest line between f_i^- and g_i^-, and that h_j^+ corresponds to the greatest line between f_i^+ or g_i^+ (lines 10, 12, 15, 18, 20, where we assign $h_j^+ = max(f_i^+, g_i^+, \cdot)$ and $h_j^- = min(f_i^-, g_i^-, \cdot)$). For this reason, it holds that $h_j^+(x_j) \geq f_i^+(x_j) \wedge h_j^+(x_j) \geq g_i^+(x_j)$ (for the upper side) and that $h_j^-(x_j) \leq f_i^-(x_j) \wedge h_j^-(x_j) \leq g_i^-(x_j)$ (for the lower side). By hypothesis, f satisfies the first validity condition, so it holds also $f_i^+(x_j) \geq f_i^-(x_j)$. Combining $h_j^+(x_j) \geq f_i^+(x_j) \wedge f_i^+(x_j) \geq f_i^-(x_j) \wedge h_j^-(x_j) \leq f_i^-(x_j)$ we obtain, by transitivity, that $h_j^+(x_j) \geq h_j^-(x_j)$. The same reasoning can be done about the value of h in x_{j+1}, obtaining $h_j^+(x_{j+1}) \geq h_j^-(x_{j+1})$. Then, h satisfies the first validity condition.

2. As before, consider the generic step $[x_j, x_{j+1}]$, where $j \in [0, M-1]$ and (h_j^-, h_j^+) is the value of h in such step. By construction of h, $\exists i : [x_j, x_{j+1}] \subseteq [t_i, t_{i+1}]$ (lines 10, 12, 15, 18, 20). Let $(f_i^-, f_i^+), (g_i^-, g_i^+)$ be the values of f and g in such step respectively. Moreover, let (h_{j+1}^-, h_{j+1}^+), $(f_{i+1}^-, f_{i+1}^+), (g_{i+1}^-, g_{i+1}^+)$ be the values of f, g and h in $[x_{j+1}, x_{j+2}]$ respectively. Using a similar notation to the one introduced in the second part of Lemma 4.5.4, let $A = \min(f_i^-(x_{j+1}), g_i^-(x_{j+1})), B = \max(f_i^+(x_{j+1}), g_i^+(x_{j+1}))$, $C = \min(f_{i+1}^-(x_{j+1}),$ $g_{i+1}^-(x_{j+1}))$, $D = \max(f_{i+1}^+(x_{j+1}), g_{i+1}^+(x_{j+1}))$ be the lower and upper values assumed by h in x_{j+1} with respect to the trapezoids of steps $[x_j, x_{j+1}]$ and $[x_{j+1}, x_{j+2}]$. The second validity condition ($[h_j^-(x_{j+1}), h_j^+(x_{j+1})] \cap [h_{j+1}^-(x_{j+1}),$ $h_{j+1}^+(x_{j+1})] \neq \emptyset$) can be then rewritten, by construction of h (lines 10, 12, 15, 18, 20, where we assign $h_j^+ = max(f_i^+, g_i^+, \cdot)$ and $h_j^- = min(f_i^-, g_i^-, \cdot)$), as $[A, B] \cap [C, D] \neq \emptyset$. By contradiction, suppose that $[A, B] \cap [C, D] = \emptyset$. Then it must be that $A > D$ or $B < C$. Suppose that $A > D$ is true (the same reasoning applies to $B < C$ symmetrically). Since $A = \min(f_i^-(x_{j+1}), g_i^-(x_{j+1}))$ and $D = \max(f_{i+1}^+(x_{j+1}), g_{i+1}^+(x_{j+1}))$, this means that $f_i^-(x_{j+1}) > f_{i+1}^+(x_{j+1})$. This is a contradiction, because either f has a step starting at x_{j+1} (and in this case f would not respect the second validity condition at the border x_{j+1}) or not. In this second case, $[x_{j+1}, x_{j+2}] \subseteq [t_i, t_{i+1}]$ and thus $f_i^- = f_{i+1}^- \wedge f_i^+ = f_{i+1}^+$. Then, $f_i^-(x_{j+1}) > f_{i+1}^+(x_{j+1})$ can be rewritten as $f_{i+1}^-(x_{j+1}) > f_{i+1}^+(x_{j+1})$ which violates the first validity condition inside step $[t_i, t_{i+1}]$. We reached a contradiction, so we must discard the absurd hypothesis that the second validity condition is not respected by $\cup^\sharp$.

□

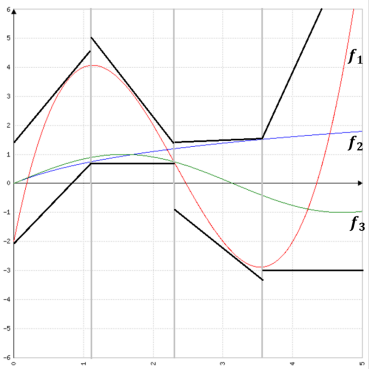


Figure 4.13: Concretization function

4.5.8 Abstraction and Concretization Functions

The abstract step function f defined by $\bigwedge_{0\leq i\leq N}\{t_i : \mathbf{v_i}\}$, where $\mathbf{v_i} = (f_i^-, f_i^+) = (m_i^-, q_i^-, m_i^+, q_i^+)$, represents the set of continuous, differentiable functions that are bounded by the lines $f_i^-(t) = m_i^- t + q_i^-$ and $f_i^+(t) = m_i^+ t + q_i^+$ for any time $t \in [t_i, t_{i+1}]$. The concretization function γ is thus defined by:

$$\gamma(\bigwedge_{0\leq i\leq N}\{t_i : \mathbf{v_i}\}) = \{g \in C^2_+ | \forall i \in [0, N], \forall t \in [t_i, t_{i+1}], g(t) \in [f_i^-(t), f_i^+(t)]\}$$

where t_{N+1} is either $+\infty$ if $dom(f) = \mathbb{R}^+$, or k if $dom(f) = [0, k]$, with k constant.

Figure 4.13 depicts an example of an abstract state defined on the domain $[0, 5]$ with 4 steps (note that here $t_{N+1} = 5$). In this Figure we can see three possible concrete functions ($f_1 = x^3 - 7x^2 + 12x - 2$, $f_2 = ln(x + 1)$ and $f_3 = sin(x)$) that are all approximated by such abstract state.

The definition of an abstraction is not as direct as the concretization. As in the case of the polyhedral domain [62], we cannot define the best one: it is always possible to increase the quality of the abstraction by increasing the number of steps. Thus, for now, we only give a criterion (the same as [24]) for a function α to be a safe abstraction (later, in Section 4.6, we will also give the definition of a sound abstraction function). Let us first define the lower- and upper-functions for a given set of continuous real functions. Let $\mathcal{Y}$ be a continuous function ($\mathcal{Y} \in \mathcal{D}$). We define two functions $\underline{\mathcal{Y}}$ and $\overline{\mathcal{Y}}$ to be the inf- and sup-functions of $\mathcal{Y}$: $\underline{\mathcal{Y}} = \lambda t.inf\{y(t) : y \in \mathcal{Y}\}$ and $\overline{\mathcal{Y}} = \lambda t.sup\{y(t) : y \in \mathcal{Y}\}$. Equivalently we define the lower- and upper-functions of a Trapezoid Step Function. Let f be $\in \mathcal{D}^\sharp$, the real-valued step functions $\underline{f}$ and $\overline{f}$ are $\underline{f} = \lambda t.f_i^-(t)$ and $\overline{f} = \lambda t.f_i^+(t)$ where, in both cases, $i = \max(\{j \in \mathbb{N} : t_j \leq t\})$. These four functions are used to define the *Validity Condition*.

Lemma 4.5.7 (Validity Condition)**.** *A function $\alpha : \mathcal{D} \to \mathcal{D}^\sharp$ satisfies the Validity Condition (V.C.) if and only if for all $\mathcal{Y} \in \mathcal{D}$, it holds that:*

$$\forall t \in \mathbb{R}^+, \underline{\alpha(\mathcal{Y})}(t) \leq \underline{\mathcal{Y}}(t) \leq \overline{\mathcal{Y}}(t) \leq \overline{\alpha(\mathcal{Y})}(t)$$

This property states that the computed Trapezoid Step Function indeed encloses the set $\{y(t) : y \in \mathcal{Y}\}$ for all $t \in \mathbb{R}^+$. The V.C. is a necessary and sufficient condition for the abstraction α to be sound.

We now formulate the theorem that guarantees the soundness of the abstraction.

Theorem 4.5.8. *If α satisfies the V.C., then for every $\mathcal{Y} \in \mathcal{D}$, $\mathcal{Y} \subseteq \gamma(\alpha(\mathcal{Y}))$.*

Proof. Let $\mathcal{Y}$ be $\in \mathcal{D}$ and $f = \alpha(\mathcal{Y})$ be $\in \mathcal{D}^\sharp$. We have to prove that $\mathcal{Y} \subseteq \gamma(f)$. Since α satisfies the V.C., we know that $\forall t \in \mathbb{R}^+, \forall y \in \mathcal{Y}, y(t) \in f(t)$. Let us take a $y \in \mathcal{Y}$. y is a continuous function that verifies $\forall t \in \mathbb{R}^+, y(t) \in f(t)$, thus $y \in \gamma(f)$. Therefore, we have that $\mathcal{Y} \subseteq \gamma(f)$.

□

4.5.9 *Compact* Operator

The dual operator of *refine* is *compact*. This operator reduces the number of steps contained in an abstract state, and it is useful in order to keep it below a given threshold and make the analysis convergent (throughout the widening which exploits this operator). *Compact* works by merging a pair of steps into a single one, and repeating the same procedure until the threshold is reached. While *refine* leaves the precision of an abstraction unchanged, the *compact* operator induces some loss of precision, since it merges together some steps.

Let $f = \bigwedge_{0 \leq i \leq N} \{t_i : \mathbf{v_i}\}$ be an abstract state, composed by $N+1$ steps, and let M be the threshold to reach, with $M < N+1$. The algorithm:

1. chooses the step with the minimum width ($w_i = t_{i+1} - t_i$)
2. merges it with the successive one
3. repeats 1 and 2 iteratively until the threshold M is reached.

We choose the step to be merged as the smallest one (i.e., the one with the smallest width), but this choice is arbitrary: alternative solutions are possible (for example considering the similarity of values of successive steps) and can be supported by our approach as well. Another possibility would be to minimize the difference between the slopes and intercepts of corresponding lines, that is, the following quantity:

$$(|m_i^+ - m_{i+1}^+|) + (|q_i^+ - q_{i+1}^+|) + (|m_i^- - m_{i+1}^-|) + (|q_i^- - q_{i+1}^-|)$$

As for the creation of the merged step, let A_i, B_i be the two extremes (the left and right ones, respectively) of f_i^+ in $[t_i, t_{i+1}]$, and let A_{i+1}, B_{i+1} be the two extremes of f_{i+1}^+ in $[t_{i+1}, t_{i+2}]$. Then the upper side f'^+ of the merged step will have the slope of the side linking A_i and B_{i+1}. If the point $P = \max(B_i, A_{i+1})$ is greater than such side, the intercept will be such that the side covers exactly P, otherwise it is kept the original intercept of the side linking A_i and B_{i+1}. Figure 4.14 depicts

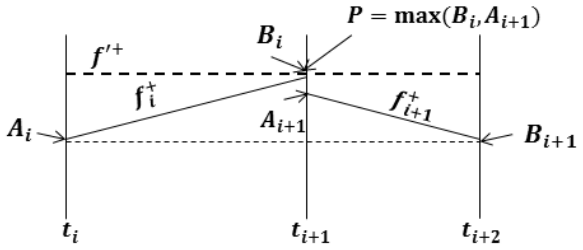


Figure 4.14: Merging of two steps within the *compact* operation

this situation. The same applies symmetrically for the lower side: we consider the minimum between the two points at t_{i+1} and we check if such point is lower than the side linking the two extremes at t_i and t_{i+2}.

A slightly different process is required if the selected step is next to last (that is, $i = N - 1$), since in such case we cannot rely on t_{i+2}. For the upper side, we consider f_N^+ and we increase its intercept if one of the extremes of f_{N-1}^+ in $[t_{N-1}, t_N]$ is greater than such side. The same procedure applies symmetrically for the lower side.

In addition, we can specify a list of steps which we do *not* want to remove from the state. Let T be the set of steps of the abstract state f, and let $X \subseteq T$ be the set of steps of f that have to be preserved. Obviously, if M is the number of steps we want in the resulting abstract state, $|X| \leq M$ holds,. Then, $g = Compact_X(f, M)$ is an abstract state obtained by compacting f to M steps, and discarding only steps coming from $T \setminus X$. The algorithm presented above can be applied in this case as well by considering only the steps in $T \setminus X$ when selecting the steps to remove.

Lemma 4.5.9 (Soundness of *compact*)**.** *Let f be $\in D^\sharp$ and M be $\in \mathbb{N}$. If $g = Compact(f, M)$, then $f \subseteq^\sharp g$.*

Proof. Let N be the number of steps of the abstract state f, where $N > M$. In order to prove the proposition above, we have to prove that $f \subseteq^\sharp Compact(f, N-1)$, that is, the execution of one step of the compaction, going from N steps to $N - 1$. If the reduction of one step decreases the size of the abstract function, then our proposition is proved to be valid, since the reduction function works by reducing repeatedly one step at a time (being equivalent to an iterative process) and the ordering is transitive. In fact, from $f = Compact(f, N) \subseteq^\sharp Compact(f, N - 1) \subseteq^\sharp \ldots \subseteq^\sharp Compact(f, M + 1) \subseteq^\sharp Compact(f, M) = g$, we derive (by transitivity of the partial ordering) $f \subseteq^\sharp g$.

Let i be the index of the step to be removed, and suppose that $i < N - 2$ (that is, the last step is not involved in the reduction; that case can be trivially proved with slight modifications to this proof). The steps we are joining together are then f_i and f_{i+1}; the result will be the step g_i of the new abstract function.

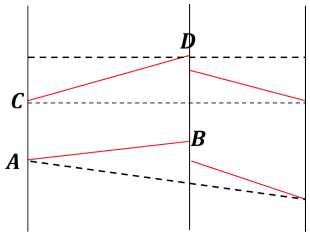


Figure 4.15: Notation

Since the steps before and after i remain unchanged, in order to prove that $f \subseteq^\sharp g$ we just need to prove that $\forall t \in [t_i, t_{i+1}] : [f_i^-(t), f_i^+(t)] \subseteq [g_i^-(t), g_i^+(t)]$ and $\forall t \in [t_{i+1}, t_{i+2}] : [f_{i+1}^-(t), f_{i+1}^+(t)] \subseteq [g_i^-(t), g_i^+(t)]$. We consider the two cases separately, first $[f_i^-(t), f_i^+(t)] \subseteq [g_i^-(t), g_i^+(t)]$ and then $[f_{i+1}^-(t), f_{i+1}^+(t)] \subseteq [g_i^-(t), g_i^+(t)]$.

We use the following notation, referring to a generic step $[t_i, t_{i+1}]$: $A = f_i^-(t_i)$, $B = f_i^-(t_{i+1})$, $C = f_i^+(t_i)$, $D = f_i^+(t_{i+1})$. See Figure 4.15 for a graphical representation of this notation.

- $([f_i^-(t), f_i^+(t)] \subseteq [g_i^-(t), g_i^+(t)])$ For the upper side, we have two distinct cases:
  - In the first one, the upper side passes for C, and D is below such side; then trivially $\forall t \in [t_i, t_{i+1}] : f_i^+(t) \leq g_i^+(t)$.
  - Otherwise, the upper side passes for D, and C is below such side (since the upper side is produced by lifting up the side passing for C, so C necessarily remains below; this case is depicted in Figure 4.15); in this case we have $\forall t \in [t_i, t_{i+1}] : f_i^+(t) \leq g_i^+(t)$ as well.

  The same can be said for the lower side: or the side that passes through A and B is above, or the side passes for B and A is above (since the side passing for A has been lowered). In both cases, we have that $\forall t \in [t_i, t_{i+1}] : f_i^-(t) \geq g_i^-(t)$. Combining the two equations we obtain that $\forall t \in [t_i, t_{i+1}] : (f_i^-(t) \geq g_i^-(t) \wedge f_i^+(t) \leq g_i^+(t))$ and this, combined with the constraint of Equation 4.1 (f_i^- is always below f_i^+ in step i-th), implies that $\forall t \in [t_i, t_{i+1}] : [f_i^-(t), f_i^+(t)] \subseteq [g_i^-(t), g_i^+(t)]$.
- $([f_{i+1}^-(t), f_{i+1}^+(t)] \subseteq [g_i^-(t), g_i^+(t)])$ this case is symmetrical to the previous one.

□

4.5.10 Widening

The widening operator is parametrized on:

- k_S, corresponding to the maximum number of steps allowed in an abstract state;

- k_M, k_Q, corresponding to the maximum value allowed for the slope and intercept of trapezoid sides respectively;
- k_I, k_L, corresponding to the increment constants for the slope and intercept, respectively.

All these parameters have to be positive. Thanks to these parameters, we can tune the widening operators at different levels of precision and efficiency.

The widening operator $\nabla_{D^\sharp}$ is then defined as follows.

$$\nabla_{D^\sharp} : (D^\sharp, D^\sharp) \to D^\sharp$$
$$f \nabla_{D^\sharp} g = \begin{cases} \top^\sharp & \text{if } |U| > k_S \\ f & \text{if } g \subseteq^\sharp f \\ Norm(Compact_U(h_{MQ}, k_S)) & \text{otherwise} \end{cases}$$

where U is the step set of the abstract state f.

We distinguish three cases:

$|U| > k_S$: f exceeds the maximum number of steps allowed in an abstract state, k_S, and we return $\top^\sharp$.

$g \subseteq^\sharp f$: We do not have an ascending chain and we simply return f, which is already normalized, being an element of $D^\sharp$.

Otherwise: We return the normalized and compacted version of h_{MQ}, keeping all the steps U of f (we know that $|U| \leq k_S$, otherwise we would have returned $\top^\sharp$). In this way, we are sure that U will be a subset of the step set of the result $(f \nabla_{D^\sharp} g)$. The abstract state h_{MQ} is built as follows. Let f be defined on the indices set U, and g be defined on the indices set V. Let $f' = Refine(f, V)$ be the refined version of f with the addition of the indices of g, and $g' = Refine(g, U)$ the refined version of g with the addition of the indices of f. Then f' and g' are defined on the same set of steps $T = U \cup V$. So we have that $f' = \bigwedge_{0 \leq i \leq N} \{t_i : \mathbf{v_i} = (f_i^-, f_i^+)\}$ and $g' = \bigwedge_{0 \leq i \leq N} \{t_i : \mathbf{w_i} = (g_i^-, g_i^+)\}$. We define h_{MQ} as:

$$h_{MQ} = \bigwedge_{0 \leq i \leq N} \{t_i : \mathbf{z_i} = (h_i^-, h_i^+)\}$$

where (h_i^-, h_i^+) are defined as follows:

$$h_i^-(x) = \begin{cases} g_i^-(x) & \text{if } f_i^- = g_i^- \\ -\infty & \text{if } (m_{g_i^-} \leq -k_M) \vee (q_{g_i^-} \leq -k_Q) \vee (m_{f_i^-} \leq -k_M) \vee (q_{f_i^-} \leq -k_Q) \\ (g_i^-)^\bullet(x) & \text{otherwise} \end{cases}$$
$$h_i^+(x) = \begin{cases} g_i^+(x) & \text{if } f_i^+ = g_i^+ \\ +\infty & \text{if } (m_{g_i^+} \geq k_M) \vee (q_{g_i^+} \geq k_Q) \vee (m_{f_i^+} \geq k_M) \vee (q_{f_i^+} \geq k_Q) \\ (g_i^+)^\circ(x) & \text{otherwise} \end{cases}$$

and

$$
\begin{aligned}
(g_i^-)^\bullet(t) &= (m_{MIN_i^-} - k_I) \times t + (q_{MIN_i^-} - k_L) \\
(g_i^+)^\circ(t) &= (m_{MAX_i^+} + k_I) \times t + (q_{MAX_i^+} + k_L) \\
m_{MIN_i^-} &= \min(m_{f_i^-}, m_{g_i^-}) \\
q_{MIN_i^-} &= \min(q_{f_i^-}, q_{g_i^-}) \\
m_{MAX_i^+} &= \max(m_{f_i^+}, m_{g_i^+}) \\
q_{MAX_i^+} &= \max(q_{f_i^+}, q_{g_i^+})
\end{aligned}
$$

The computation is symmetric for the lower and upper side, so let us focus on $h_i^+(x)$. For each step t_i of f' and g' we consider three distinct cases:

- $f_i^+ = g_i^+$: the side is the same in f' and g', so we keep it unchanged.
- $(m_{g_i^+} \geq k_M) \vee (q_{g_i^+} \geq k_Q) \vee (m_{f_i^+} \geq k_M) \vee (q_{f_i^+} \geq k_Q)$: the slope (or the intercept) of the side of one abstract state (g' or f') exceeds the threshold k_M (k_Q), so we move the side to $+\infty$.
- Otherwise: we keep the maximum slope and intercept between their values in f_i^+ and g_i^+ and then we increase them both by a predefined constant quantity (k_I for the slope, k_L for the intercept).

Intuitively, the convergence is guaranteed by the combination of:

- the application of *compact* with the parameter k_S;
- the parameters k_M and k_Q that limit the maximal values allowed for the slope and the intercept of a line, respectively. If a certain line exceeds one of these two values, then it goes to $\pm\infty$, stopping its possible growth in that direction;
- the $\bullet$ and $\circ$ operators, that shift a side down ($\bullet$) and up ($\circ$) for a predefined amount respectively. In particular, referring to $\circ$, the maximum slope $m_{MAX_i^+}$ between the ones of f_i^+ and g_i^+ is increased by k_I, while the maximum intercept $q_{MAX_i^+}$ between the one of f_i^+ and g_i^+ is increased by k_L. k_I and k_L are positive values and ensure that we will reach the convergence in a finite number of steps.

For the soundness of this operator we refer to the definition of [40, 43], through the two properties of covering and termination.

Lemma 4.5.10 (Correctness of $\nabla_{D^\sharp}$)**.** *The widening operator $\nabla_{D^\sharp}$ is correct, that is, it respects the properties of covering and termination:*

1. *Covering: $\forall f, g \in D^\sharp : f \subseteq^\sharp (f \nabla_{D^\sharp} g) \wedge g \subseteq^\sharp (f \nabla_{D^\sharp} g)$*
2. *Termination: for every ascending chain $\{f_j\}_{j \geq 0}$, the ascending chain defined as $g_0 = f_0; g_{j+1} = g_j \nabla_{D^\sharp} f_{j+1}$ stabilizes after a finite number of terms.*

Proof. We prove first the covering property, then the termination one.

Covering By definition of $\nabla_{D^\sharp}$, we have three possible cases:

1. If $|U| > k_S$ we return $\top^\#$, and trivially $f \subseteq^\sharp \top^\# \wedge g \subseteq^\sharp \top^\#$
2. If $g \subseteq^\sharp f$ then we return f and trivially $f \subseteq^\sharp f$ by the reflexivity property of the partial order and $g \subseteq^\sharp f$ by hypothesis.
3. In the last case, suppose that the two functions are defined on the same step set (we can obtain that with the *refine* operator, without changing their ordering by Lemma 4.5.2), $f = \bigwedge_{0\leq i\leq N}\{t_i : \mathbf{v_i}\} \wedge g = \bigwedge_{0\leq i\leq N}\{t_i : \mathbf{w_i}\}$. Let $h = \bigwedge_{0\leq i\leq N}\{t_i : \mathbf{z_i}\}$ be the result of the widening of f and g. We must prove that $\forall i, \mathbf{w_i} \sqsubseteq_{[t_i,t_{i+1}]} \mathbf{z_i} \wedge \mathbf{v_i} \sqsubseteq_{[t_i,t_{i+1}]} \mathbf{z_i}$. Consider first $\mathbf{w_i} \sqsubseteq_{[t_i,t_{i+1}]} \mathbf{z_i}$. By definition of $\sqsubseteq_{[t_i,t_{i+1}]}$, we have that $\mathbf{w_i} \sqsubseteq_{[t_i,t_{i+1}]} \mathbf{z_i} \Leftrightarrow [g_i^-(t_i), g_i^+(t_i)] \subseteq [h_i^-(t_i), h_i^+(t_i)] \wedge [g_i^-(t_{i+1}), g_i^+(t_{i+1})] \subseteq [h_i^-(t_{i+1}), h_i^+(t_{i+1})] \Leftrightarrow g_i^-(t_i) \geq h_i^-(t_i) \wedge g_i^+(t_i) \leq h_i^+(t_i) \wedge g_i^-(t_{i+1}) \geq h_i^-(t_{i+1}) \wedge g_i^+(t_{i+1}) \leq h_i^+(t_{i+1})$. Regarding $h_i^-(t)$, we know that it could be:
   - $h_i^-(t) = g_i^-(t)$. Then $\forall t \in [t_i, t_{i+1}] : g_i^-(t) \geq g_i^-(t)$
   - $h_i^-(t) = -\infty$. Then $\forall t \in [t_i, t_{i+1}] : g_i^-(t) \geq -\infty$
   - $h_i^-(t) = (g_i^-)^\bullet(t)$ that is $h_i^-(t) = (m_{MIN_i^-} - k_I) \times t + (q_{MIN_i^-} - k_L) = m_{MIN_i^-} \times t - k_I \times t + q_{MIN_i^-} - k_L$. Since k_I, k_L are both ≥ 0 by definition and $m_{MIN_i^-} \leq m_{g_i^-} \wedge q_{MIN_i^-} \leq q_{g_i^-}$ also by definition, we can say that $h_i^-(t) = m_{MIN_i^-} \times t + q_{MIN_i^-} - k_I \times t - k_L \leq m_{g_i^-} \times t + q_{g_i^-} = g_i^-(t)$ holds $\forall t \geq 0$.

   The same happens symmetrically for $h_i^+(x)$. We have then proved that $\forall i : \mathbf{w_i} \sqsubseteq_{[t_i,t_{i+1}]} \mathbf{z_i}$. The same reasoning can be made considering f instead of g, ending in proving that $\forall i : \mathbf{v_i} \sqsubseteq_{[t_i,t_{i+1}]} \mathbf{z_i}$. Thus $g \subseteq^\sharp h \wedge f \subseteq^\sharp h$.

Termination We want to prove that the ascending chain $\{g_j\}_{j\geq 0}$ stabilizes after a finite number of terms. Consider a generic element of the chain, g_j. Since the chain is ascending, $g_{j-1} \subseteq^\sharp g_j$. If $g_{j-1} = g_j$, the chain has stabilized and the convergence is reached. Otherwise, we have that $g_{j-1} \subset^\sharp g_j$. By definition of $\subseteq^\sharp$, this means that the area of (at least) one trapezoid of g_j has increased with respect to the corresponding trapezoid of g_{j-1}. Let (s_i^-, s_i^+) be the lower and upper sides of such trapezoid in g_j. The area of the step is increased if the two lines (s_i^- and s_i^+) drift apart with respect to their values at the preceding iteration, that is, in g_{j-1}. In particular, or s_i^- went down (that is, the slope, the intercept or both decreased), or s_i^+ went up (that is, the slope, the intercept or both increased). By definition of widening, we know that the lines will stop increasing or decreasing when their slope or intercept reach the threshold values (k_M, k_Q) they will be approximated by $+\infty$ or $-\infty$. We also know that the growth in the number of steps is limited to k_S, and that the

steps of the previous value in the chain will be preserved (the widening only adds indices, but it does not remove them).

Suppose that the chain ascends of the minimum possible quantity (with respect to our parameters k_I, k_L and the float numbers representation of the computer).

Let us define

$$\begin{aligned} maxIterationsToStopGrowth(i) = &\min(\frac{|m_i^+ - k_M|}{k_I}, \frac{|q_i^+ - k_Q|}{k_L}) + \\ &\min(\frac{|m_i^- - k_M|}{k_I}, \frac{|q_i^- - k_Q|}{k_L}) \end{aligned}$$

This value is an upper bound of the number of iterations needed for step t_i to reach the threshold after which it stops growing. In particular, consider a generic step t_i and the increase of its upper side. Since the slope is increased of at least k_I and, at the same time, the intercept is increased of at least k_L, the side will stop to grow after, at most, $\min(\frac{|m_i^+ - k_M|}{k_I}, \frac{|q_i^+ - k_Q|}{k_L})$ iterations (after that, the side is set to $+\infty$, its maximum value). The same happens for the lower side, which stops to grow after, at most, $\min(\frac{|m_i^- - k_M|}{k_I}, \frac{|q_i^- - k_Q|}{k_L})$ iterations. Thus, each step t_i is abstracted to top after at most $\min(\frac{|m_i^+ - k_M|}{k_I}, \frac{|q_i^+ - k_Q|}{k_L}) + \min(\frac{|m_i^- - k_M|}{k_I}, \frac{|q_i^- - k_Q|}{k_L})$ iterations, that is, the quantity $maxIterationsToStopGrowth(i)$.

Let $g = \bigwedge_{i \in [0,N]} \{t_i : \mathbf{v_i} = (m_i^-, q_i^-, m_i^+, q_i^+)\}$ be a generic element of the ascending chain $\{g_j\}$. Then, an upper bound of the maximum number of iterations needed to converge is:

$$\begin{aligned} K = \; & \textstyle\sum_{i=0}^{N} maxIterationsToStopGrowth(i) + \\ & + (k_S - N) \times maxIterationsToStopGrowth(iMAX) \end{aligned} \tag{4.5}$$

where $iMAX = \{i \in [0, N] : \nexists j \in [0, N] : maxIterationsToStopGrowth(j) > maxIterationsToStopGrowth(i)\}$ is the step index which maximizes the quantity $maxIterationsToStopGrowth$.

Equation 4.5 is composed by two parts:

- $\sum_{i=0}^{N} maxIterationsToStopGrowth(i)$ represents an upper bound of the number of iterations needed for *all* the steps of g to reach the threshold after which they stop growing. In fact, at each iteration of the widening, at least one side of one step is moved up or down and this means that its slope and intercept move closer to the thresholds k_M, k_Q. Each step t_i of g will stop to grow after, at most, $maxIterationsToStopGrowth(i)$ iterations. Summing this quantity for all the N steps of g, we obtain the first part of Equation 4.5.

– $(k_S - N) \times maxIterationsToStopGrowth(iMAX)$ represents an upper bound of the number of iterations needed for all the (possibly) added steps to reach the threshold after which they stop growing. In fact, we know that the maximum number of steps allowed in an abstract state by the widening is k_S. The state g is composed by N steps, so at most $k_S - N$ steps could be added to g along the chain. Nevertheless, we know that the chain $\{g_j\}$ is increasing, so these added steps cannot have values that are smaller (i.e., far from the thresholds that stop the growth) than the already existing values in g. In order to have an upper bound of the iterations needed by these added step to stop growing, we then suppose that all these steps lie within the step of g which is farthest from the growth stop. More precisely, let $iMAX$ be the index of the step which maximizes the quantity $maxIterationsToStopGrowth(\cdot)$. Then, each added step stops to grow after, at most, $maxIterationsToStopGrowth(iMAX)$ iterations. Multiplying this value by $k_S - N$ (the maximum number of steps we can add to g) we obtain the second part of Equation 4.5.

Since K is the result of the summation of two finite quantities, we can say that the ascending chain $\{g_j\}_{j \geq 0}$ stabilizes after a finite number of terms, which is at most K.

□

4.5.11 The Lattice $D^\sharp$

Theorem 4.5.11. $(D^\sharp \cup \{\bot^\sharp\}, \subseteq^\sharp, \bot^\sharp, \top^\sharp, \cup^\sharp, \cap^\sharp)$ *is a lattice.*

Proof. By Lemma 4.5.1 we get that $\subseteq^\sharp$ is a partial order, by Lemma 4.5.5 that $\cup^\sharp$ is the least upper bound operator, by Lemma 4.5.4 that $\cap^\sharp$ is the greatest lower bound operator. □

We also proved the correctness of the widening operator $\nabla_{D^\sharp}$ in Lemma 4.5.10. Therefore, we have defined and proved the correctness of all the operators required to define an abstract domain in the Abstract Interpretation framework.

4.6 Abstraction of a Continuous Function

In this section, we show how to compute the approximation of $\mathcal{C}^2_+$ functions in IVSF and TSF. We consider IVSF as well since [24] did not define its abstraction function, because they relied on a particular type of ODE solver [23]. For both domains, we consider two different approaches: when the step width is constant and fixed, and when we automatically determine the step distribution. Note that we abstract only one concrete function; this approach can be generalized to the abstraction of a

countable set of concrete functions C by computing the abstraction of each function in the set and then returning the least upper bound of all the resulting abstract states.

In the following subsections, we will denote by:

- $f \in \mathcal{C}^2_+$ the continuous function we want to abstract;
- f', f'' its first and second derivatives;
- F'_0 the set containing the points of the domain where $f'(x) = 0$ (the minimum and maximum points of the function), that is, $F'_0 = \{t : f'(t) = 0\}$;
- F''_0 the set containing the points of the domain where $f''(x) = 0$ (the inflection points), that is, $F''_0 = \{t : f''(t) = 0\}$;
- $G'^{[a,b]}_0$ the set containing the maximum and minimum points of f restricted to the domain interval $[a, b]$, that is, $G'^{[a,b]}_0 = \{f(t) : t \in ([a, b] \cap F'_0)\}$;
- $G''^{[a,b]}_0$ the same set but for the inflection points, that is, $G''^{[a,b]}_0 = \{f(t) : t \in ([a, b] \cap F''_0)\}$.

4.6.1 IVSF Abstraction Function, Fixed Step Width

Given a fixed step width w, suppose that $[a, b]$ is a generic interval ($b - a = w \wedge a = k \times w \wedge b = (k + 1) \times w \wedge k \geq 0, w > 0$). $M = \max(\{f(a), f(b)\} \cup G'^{[a,b]}_0)$ is the maximum point of the function in the interval $[a, b]$, extremes included, and $m = \min(\{f(a), f(b)\} \cup G'^{[a,b]}_0)$ is the minimum point of the function in the interval $[a, b]$, extremes included. The best abstraction in IVSF of this step is the interval $[m, M]$. To build the abstraction of the function f, we repeat this procedure for each step of the abstract state.

4.6.2 IVSF Abstraction Function, Arbitrary Step Width

For the IVSF abstract domain, we cannot find *a priori* the best way to split the domain of f in sub-intervals. We could use different techniques, for example:

- splitting in correspondence of the inflection points of the function, and
- splitting in correspondence of the intermediate point between each pair of minimum/maximum points.

There is no best choice that works in any case, since this choice depends on the shape of the function. In some cases it is more precise to use the inflection points for splitting, in other cases the intermediate points yield more precision. In addition we could split also in correspondence of the minimum/maximum points; sometimes (more often than not) this increases the precision of the representation.

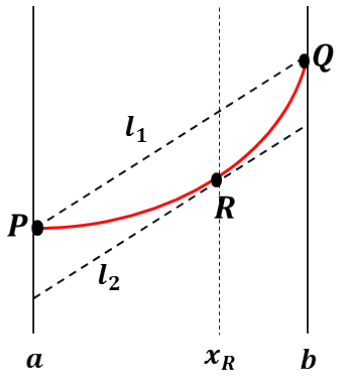


Figure 4.16: The abstraction on the step $[a, b]$

Once we have chosen a technique for splitting the domain in steps, we can apply the procedure introduced in Section 4.6.1 to compute the abstraction of each single step.

4.6.3 TSF Basic Abstraction Function, Arbitrary Step Width

In TSF a very good trade-off between complexity and precision of the abstraction can be achieved by splitting the domain in correspondence of (i) the maximum and minimum points F'_0, and (ii) the inflection points F''_0.

Assume that $[a, b]$ is a generic sub-interval obtained using this schema. Then the two sides which compose the value of such step are the following ones (see Figure 4.16):

1. the side l_1 linking the points $P = (a, f(a))$ and $Q = (b, f(b))$ (the points of f in correspondence of the extremes of the interval $[a, b]$).

2. the side l_2 which has the same slope as l_1 and is tangent to f inside $[a, b]$. Since we already know the slope of this side, we just need to compute its intercept. The procedure is the following one:

   (a) find the point $x_R \in [a, b]$ where the first derivative of f is equal to the slope of l_1: $f'(x_R) = m_{l_1}$. This point can be computed by bisection in $[a, b]$.

   (b) let R be the point with coordinates $(x_R, f(x_R))$. Then l_2 is the side that goes through the point R and with slope equal to the one of l_1 ($m_{l_2} = m_{l_1}$). The intercept is computed as follows: $q_{l_2} = f(x_R) - m_{l_2} \times x_R$.

Note that the resulting sides l_1 and l_2 are parallel, as they have the same slope. Moreover, $l2$ is a tangent of f.

4.6.4 TSF Basic Abstraction Function, Fixed Step Width

Also in the case of TSF we can define the abstraction on a fixed step width. Suppose that $[a, b]$ is a generic interval determined by a fixed width w. First of all, we split the interval in sub-intervals, following the schema introduced in Section 4.6.3. Then for each sub-interval, we compute the upper and lower sides as specified in Section 4.6.3. Finally, we have to "join" these sub-intervals into a single one (with range $[a, b]$) through the *compact* operator (see Section 4.5.9).

Observe that the abstraction function of IVSF is less restrictive than the one of TSF, since TSF's abstraction needs the second derivative of the function to know the inflection points.

4.6.5 Dealing with Floating Point Precision Issues in TSF

Unfortunately, the abstraction technique presented in Section 4.6.3 is theoretically sound but it is not computable on a finite precision machine, due to the rounding issues of floating point representation. The abstraction function depends on various values: the stationary points (F'_0), the inflection points (F''_0), the point $x_r \in [a, b]$ such that $f'(x_R) = m_{l1}$. Even knowing exactly all the points in F'_0 and F''_0 by mathematical analysis, we could not be able to precisely represent them in a machine (e.g., $\sqrt{2}$). Therefore, we can only compute an approximation of such points and not their exact value. In this section, we introduce some restrictions on the functions we can manipulate and a refinement of the basic abstraction function proposed in Section 4.6.3 to enforce the soundness of the resulting abstraction function in a floating point computation.

We assume that f respects the following property. If x_0 is a point such that $f'(x_0) = 0$, then, for each interval $[\overline{x}, \overline{x} + \epsilon]$ such that $x_0 \in [\overline{x}, \overline{x} + \epsilon]$, we have:

$$\forall x \in [\overline{x}, \overline{x} + \epsilon] : f(x) \in [f(\overline{x}) - \tau, f(\overline{x}) + \tau]$$

where τ is a parameter of the analysis and ϵ is a constant depending on the machine in use. Intuitively, we ask that function's values change at most of τ around stationary points. We impose the same constraint on the inflection points (x s.t. $f''(x) = 0$). Note that the value of τ has to be set by the user: the smaller the value, the more precise the abstraction. The value of ϵ, instead, should be set based on the standard in use on the machine (for instance, IEEE 754 for floating points), in order to choose the smallest possible value. In fact, the smaller is ϵ, the smaller we can set τ (because, in the context of continuous functions, smaller changes in the input imply also smaller changes in the output).

Like in Section 4.6.3, we split the domain in steps with respect to the stationary points (F'_0) and the inflection points (F''_0). If we cannot pinpoint those points exactly, we introduce an additional step of width ϵ in correspondence of them. The exact location of the step $[t_i, t_{i+1}] = [\overline{x}, \overline{x}+\epsilon]$ depends on the numerical representation of the machine and it obviously must contain the exact value of the considered

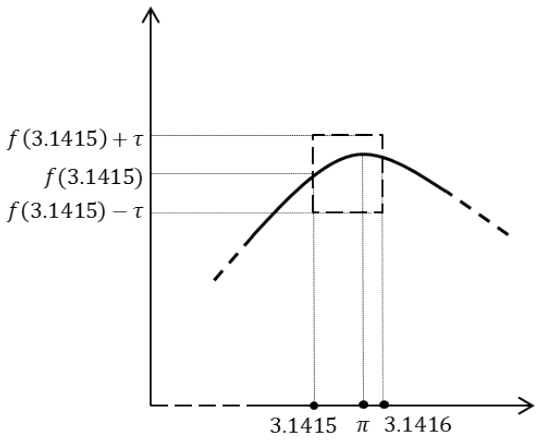


Figure 4.17: Creation of the step around stationary point π

stationary or inflection point. Intuitively, $\overline{x}$ will be the greatest representable under-approximation of the stationary/inflection point. The value of such additional step is $\mathbf{v_i} = (0, f(\overline{x}) - \tau, 0, f(\overline{x}) + \tau)$. For the condition imposed above, we are sure that this trapezoid (which is a rectangle, since the two sides are horizontal) soundly contains the abstracted function in the considered step.

Let us see a simple example to better understand how these kinds of steps are computed. Consider a function f in the restricted domain $[0, 4]$. Assume we know by mathematical analysis that, in such domain, the function has only one stationary point (in correspondence of $x = \pi = 3.14159\ldots$) and no inflections points. More formally, we know that: $G'^{[0,4]}_0 = \{\pi\}$ and $G''^{[0,4]}_0 = \emptyset$. Suppose that the machine in use has a precision such that $\epsilon = 0.0001$. This means that we can represent values like $0, 0.0001, 0.0002, \ldots, 3.1415, 3.1416, \ldots$. So, we are not able to represent any other number between 3.1415 and 3.1416. Since $3.1415 \leq \pi \leq 3.1416$, we choose $\overline{x} = 3.1415$. Then we have $[\overline{x}, \overline{x} + \epsilon] = [3.1415, 3.1416]$. The resulting step function constraint around the stationary point π is

$$t_i = 3.1415 : \mathbf{v_i} = (0, f(3.1415) - \tau, 0, f(3.1415) + \tau)$$

where τ is a parameter given by the user and specific with respect to the function which we want to abstract. We have created a step that starts at 3.1415 and whose trapezoid lower and upper sides are horizontal lines at the height of $f(3.1415) - \tau$ and $f(3.1415) + \tau$, respectively. We can see this step depicted in Figure 4.17. In this case the trapezoid is a rectangle. The next step will start at $t_{i+1} = 3.1416$.

For the steps that do not contain stationary and inflection points, the computational schema of Section 4.6.3 is refined as follows. The side l_1 (the one which links the extremes of f in the step) is moved up (or down, depending on the concavity of f in the step) of ϵ. This compensates for potential errors in the evaluation of the

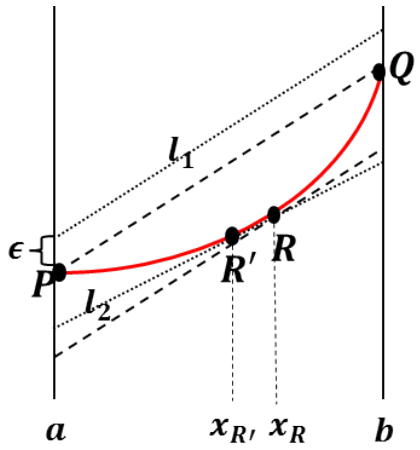


Figure 4.18: Creation of steps without stationary/inflection points

function values at the extremes. The other side l_2 goes through the point $x_R{}'$ such that $f'(x_R{}')$ is the closest value to m_{l1} that we can reach (given the precision of the machine). The slope of l_2 is $f'(x_R{}')$ so that l_2 is tangent to the function. Since we know that the function is concave (or convex) in the considered subinterval, we are sure that one of its tangents leaves the function always above (or below), resulting in a safe approximation. This refined computational schema is represented in Figure 4.18.

The round dotted lines l_1 and l_2 represent the final result of the computation. The intermediate process is as follows. First, we create the dashed line joining $P = (a, f(a))$ and $Q = (b, f(b))$, then we move it up of ϵ, resulting in l_1. Then, since we cannot find the dashed line going through the point R (a line tangent to the function and having the same slope as l_1) because of rounding approximation, we use instead the round dotted line going through R', that is, the closest point to R with respect to the machine precision. The slope of the line l_2 is $f'(x_{R'})$, that is, l_2 is tangent in R' to the function f being abstracted.

4.6.6 Dealing with Floating Point Precision Issues in IVSF

The same issues about floating point rounding issues arise for the abstraction function presented in Section 4.6.1. In this case, though, it is very easy to make the abstraction sound and we do not need to impose any requirement on the function to abstract. After having determined the minimum and maximum values (m and M) assumed by the function in the considered step, it is sufficient to return $[m-\epsilon, M+\epsilon]$ (instead of $[m, M]$) as step value. This compensates for possible approximation errors due to the finite precision of the machine.

4.7 Abstract Semantics

In this section we are going to define only the abstract version of the concrete operations presented in Section 4.4 which deal with functional expressions. For the remaining language constructs (statement operations, logic combination of boolean values, boolean comparisons), we refer to the usual abstract semantics of the classical Abstract Interpretation framework. This means that we are going to specify the abstract semantics of `newFun`, `constFun`, `valueAt`, modulus, sum, difference, product, division, composition, minimum, maximum.

The first three operators are simple:

- the abstract semantics of `newFun` is the abstraction function defined in Section 4.6;
- the creation of a constant function (`constFun(c)`) returns a step function with a single step with value $\mathbf{v_i} = (0, c, 0, c)$;
- the abstract semantics of $\texttt{valueAt}_\mathbf{x}$ finds the step t_i such that $t_i \leq x < t_{i+1}$ and returns the interval of values (m, M) such that $m = f_i^-(x)$ and $M = f_i^+(x)$. In other words, it returns the interval delimited by the two sides of the trapezoid containing the input x.

Now we are going to consider the arithmetic operators on functions. One of them (the modulus) is unary, while all the others are binary operators. For the rest of the section, let f, g be two abstract values defined on the same set of steps $T = \{t_i : i \in [0, N]\}$:

$$f = \bigwedge_{i \in [0,N]} \{t_i : \mathbf{f_i}\}$$

$$g = \bigwedge_{i \in [0,N]} \{t_i : \mathbf{g_i}\}$$

If the abstract operation receives in input two abstract functions defined on different sets of steps, we can always execute a preliminary refinement of the two abstract states on the union of their step sets.

The result of the abstract arithmetic operation is another abstract value, h, defined on the same set of steps as the input functions:

$$h = \bigwedge_{i \in [0,N]} \{t_i : \mathbf{h_i}\}$$

For each operation we are going to define only how to compute the value $\mathbf{h_i}$ associated the generic step t_i, that is the step $([t_i, t_{i+1}])$, as the result of the abstract semantic operation applied to the two corresponding steps t_i from f and g. We assume that, at t_i, f has value $\mathbf{f_i} = (m, q, n, r) = (f_i^-, f_i^+)$ and g has value $\mathbf{g_i} = (m', q', n', r')$

$= (g_i^-, g_i^+)$. If the operation is unary (i.e., the modulus), then we refer only to $t_i : \mathbf{f_i} = (m, q, n, r) = (f_i^-, f_i^+)$ from f.

The abstract semantics of arithmetic operators is inspired by the interval arithmetic presented in Section 2.2.

Last step

Note that, for the abstract operations of *product*, *division* and *composition* there is the problem of the last step. In fact, such operations use the information about t_{i+1} (the right border of the interval), but the last step does not have this information, so we are forced to return $\top$ in such step. The *sum* and *difference* operations do not need the information about t_{i+1} so the computation of the last step is identical to the one of the other steps. The *modulus*, *min* and *max* operation, instead, would require such information, but, with a special treatment of the last step case, we are able to return a value different from $\top$ in such step.

Sum

The abstract sum operation returns, for each step, the two linear functions obtained by summing the coefficients of, respectively, the lower and upper functions from f and g:

$$\mathbf{h_i} = (h_i^-, h_i^+) = (m + m', q + q', n + n', r + r')$$

where:

$$\begin{aligned} h_i^-(t) &= f_i^-(t) + g_i^-(t) &&= (mt + q) + (m't + q') = (m + m')t + (q + q') \\ h_i^+(t) &= f_i^+(t) + g_i^+(t) &&= (nt + r) + (n't + r') = (n + n')t + (r + r') \end{aligned}$$

Difference

The abstract difference operation works similarly to the sum one, but we have to be careful to subtract the upper function of g from the lower function of f (and, respectively, subtract the lower function of g from the upper function of f), so that we are sure to obtain the minimum (respectively, maximum) values possible. Formally:

$$\mathbf{h_i} = (h_i^-, h_i^+) = (m - n', q - r', n - m', r - q')$$

where:

$$\begin{aligned} h_i^-(t) &= f_i^-(t) - g_i^+(t) &&= (mt + q) - (n't + r') = (m - n')t + (q - r') \\ h_i^+(t) &= f_i^+(t) - g_i^-(t) &&= (nt + r) - (m't + q') = (n - m')t + (r - q') \end{aligned}$$

Product

The basic idea of the product abstract operation is to compute the product of all four pairs of functions made by a function of f (f_i^-, f_i^+) and one of g (g_i^-, g_i^+), overapproximate their quadratic components and finally computing the values in

correspondence of the extremities of the step (keeping only the minimum and maximum values). More formally:

- We compute the products of functions pairs obtained by taking one function from f and one from g. We obtain four parabolas with the following values:

$$\begin{array}{lll} A(t) & = f_i^-(t) \times g_i^-(t) & = (mt+q) \times (m't+q') = mm't^2 + (mq'+m'q)t + qq' \\ B(t) & = f_i^-(t) \times g_i^+(t) & = (mt+q) \times (n't+r') = mn't^2 + (mr'+n'q)t + qr' \\ C(t) & = f_i^+(t) \times g_i^+(t) & = (nt+r) \times (n't+r') = nn't^2 + (nr'+n'r)t + rr' \\ D(t) & = f_i^+(t) \times g_i^-(t) & = (nt+r) \times (m't+q') = nm't^2 + (nq'+m'r)t + q'r \end{array}$$

- In each parabola, we overapproximate the quadratic term (t^2) with the two possible values t_i and t_{i+1}, which are the minimum and maximum values that the variable t can assume in the considered step. We then obtain eight lines, which we call $A_i, A_{i+1}, B_i, B_{i+1}, C_i, C_{i+1}, D_i, D_{i+1},$, where the pedix indicates the values substituted to t^2 (i.e., $B_{i+1}(t) = mn'(t_{i+1})^2 + (mr'+n'q)t + qr'$).

- We look for the minimum and maximum values assumed by the eight lines in correspondence of the two extremities of the step (t_i and t_{i+1}). We call S, T the minimum and maximum values in correspondence of t_i, and U, V those in correspondence of t_{i+1}:

$$\begin{array}{l} S = \min\left(A_i(t_i), B_i(t_i), C_i(t_i), D_i(t_i), A_{i+1}(t_i), B_{i+1}(t_i), C_{i+1}(t_i), D_{i+1}(t_i)\right) \\ T = \max\left(A_i(t_i), B_i(t_i), C_i(t_i), D_i(t_i), A_{i+1}(t_i), B_{i+1}(t_i), C_{i+1}(t_i), D_{i+1}(t_i)\right) \\ U = \min\left(A_i(t_{i+1}), B_i(t_{i+1}), C_i(t_{i+1}), D_i(t_{i+1}), A_{i+1}(t_{i+1}), B_{i+1}(t_{i+1}), C_{i+1}(t_{i+1}), D_{i+1}(t_{i+1})\right) \\ V = \max\left(A_i(t_{i+1}), B_i(t_{i+1}), C_i(t_{i+1}), D_i(t_{i+1}), A_{i+1}(t_{i+1}), B_{i+1}(t_{i+1}), C_{i+1}(t_{i+1}), D_{i+1}(t_{i+1})\right) \end{array}$$

- We return the trapezoid made by two lines $\overline{SU}$ and $\overline{TV}$, i.e. the lines which connect, respectively, the two minimum values (S, U) and the two maximum ones (T, V):

$$\mathbf{h_i} = (h_i^-, h_i^+) = (m_{\overline{SU}}, q_{\overline{SU}}, m_{\overline{TV}}, q_{\overline{TV}})$$

  where $m_{\overline{XY}}$ and $q_{\overline{XY}}$ represent, respectively, the slope and intercept of the line connecting two points X, Y.

Division

The procedure to compute the division between two abstract functions works very similarly to the one explained above for the product. However, here we must pay particular attention to the possible division by/of zero: if one of the abstract functions contains the value zero (i.e., it could assume the value zero in one or more points of the step), then we will have to treat the situation appropriately. If the abstract function which contains the zero is the denominator, then we return $\perp$ (we signal a possible computation error, a division by zero); if, instead, the abstract function which contains the zero is the numerator, then we return $\top$ (since the division of "zero by something" results in ∞). If the zero is not present in any of the

abstract functions, then we apply the same reasoning used for the product. More formally:

- We compute the division of functions pairs obtained by taking one function from f and one from g. We obtain four functions:

$$\begin{aligned} A(t) &= \frac{f_i^-(t)}{g_i^-(t)} &= \frac{mt+q}{m't+q'} \\ B(t) &= \frac{f_i^-(t)}{g_i^+(t)} &= \frac{mt+q}{n't+r'} \\ C(t) &= \frac{f_i^+(t)}{g_i^+(t)} &= \frac{nt+r}{n't+r'} \\ D(t) &= \frac{f_i^+(t)}{g_i^-(t)} &= \frac{nt+r}{m't+q'} \end{aligned}$$

- In each of the four functions, we overapproximate the variable contained in the *denominator* with the two possible values t_i and t_{i+1}, which are the minimum and maximum values that the variable t can assume in the considered step. We then obtain eight lines, which we call $A_i, A_{i+1}, B_i, B_{i+1}, C_i, C_{i+1}, D_i, D_{i+1},$, where the pedix indicates the values substituted to t in the denominator (for example, $B_{i+1}(t) = \frac{mt+q}{n't_{i+1}+r'}$ and $D_i(t) = \frac{nt+r}{m't_i+q'}$).

- We look for the minimum and maximum values assumed by the eight lines in correspondence of the two extremities of the step (t_i and t_{i+1}). Exactly as we did for the product, we call S, T the minimum and maximum values in correspondence of t_i, and U, V those in correspondence of t_{i+1}:

$$\begin{aligned} S &= \min\left(A_i(t_i), B_i(t_i), C_i(t_i), D_i(t_i), A_{i+1}(t_i), B_{i+1}(t_i), C_{i+1}(t_i), D_{i+1}(t_i)\right) \\ T &= \max\left(A_i(t_i), B_i(t_i), C_i(t_i), D_i(t_i), A_{i+1}(t_i), B_{i+1}(t_i), C_{i+1}(t_i), D_{i+1}(t_i)\right) \\ U &= \min\left(A_i(t_{i+1}), B_i(t_{i+1}), C_i(t_{i+1}), D_i(t_{i+1}), A_{i+1}(t_{i+1}), B_{i+1}(t_{i+1}), C_{i+1}(t_{i+1}), D_{i+1}(t_{i+1})\right) \\ V &= \max\left(A_i(t_{i+1}), B_i(t_{i+1}), C_i(t_{i+1}), D_i(t_{i+1}), A_{i+1}(t_{i+1}), B_{i+1}(t_{i+1}), C_{i+1}(t_{i+1}), D_{i+1}(t_{i+1})\right) \end{aligned}$$

- We return the trapezoid made by two lines $\overline{SU}$ and $\overline{TV}$, i.e. the lines which connect, respectively, the two minimum values (S, U) and the two maximum ones (T, V).

Considering also the possible division by/of zero, the final result returned by the abstract division operation for the step t_i is then the following:

$$\mathbf{h_i} = (h_i^-, h_i^+) = \begin{cases} \bot & \text{if } \exists t \in [t_i, t_{i+1}) : 0 \in [g_i^-(t), g_i^+(t)] \\ \top & \text{if } \exists t \in [t_i, t_{i+1}) : 0 \in [f_i^-(t), f_i^+(t)] \\ (m_{\overline{SU}}, q_{\overline{SU}}, m_{\overline{TV}}, q_{\overline{TV}}) & \text{otherwise} \end{cases}$$

Composition

First of all, note that for this operation we do not need the two abstract values to be defined on the same set of steps, because, given an input $\bar{t}$, we do not compute the values of both f, g in $\bar{t}$: instead, we compute the value of only g in $\bar{t}$ and then

pass the result as input to f. In this case, then, the domain of f is related to the *codomain* of g. We consider $g = \bigwedge_{i\in[0,M]}\{t_i : \mathbf{g_i}\}$ and $f = \bigwedge_{j\in[0,N]}\{u_j : \mathbf{f_j}\}$. The result of the composition $f \circ g$ will be an abstract function $h = \bigwedge_{i\in[0,M]}\{t_i : \mathbf{h_i}\}$ (the domain is the same of g).

The procedure to compute the composition of two abstract functions is the following: for each step t_i of the inner function (g), we check its possible values; then, we consider all the steps of f which domain includes at least one of such values and we keep the lowest and greatest values assumed by f in those steps. More formally:

- We compute g_{min}, g_{max} which are, respectively, the minimum and maximum values assumed by g_i^-, g_i^+ throughout the interval $[t_i, t_{i+1}]$:

$$g_{min} = \min_{t\in[t_i,t_{i+1}]} g_i^-(t)$$

$$g_{max} = \max_{t\in[t_i,t_{i+1}]} g_i^+(t)$$

- We find all the steps of f which domain intersects the interval $[g_{min}, g_{max}]$:

$$J = \{j : [u_j, u_{j+1}] \cap [g_{min}, g_{max}] \neq \emptyset\}$$

- We compute f_{min}, f_{max} which are, respectively, the lowest and greatest value assumed by f^-, f^+ in all the steps f_j (where $j \in J$):

$$f_{min} = \min_{j\in J} \min_{u\in[u_j,u_{j+1}]} f_j^-(u)$$

$$f_{max} = \max_{j\in J} \max_{u\in[u_j,u_{j+1}]} f_j^+(u)$$

- We return the trapezoid made by two horizontal lines in correspondence of f_{min}, f_{max}:

$$\mathbf{h_i} = (h_i^-, h_i^+) = \begin{cases} (0, f_{min}, 0, f_{max}) & \text{if } g_{max} \geq 0 \\ \bot & \text{otherwise} \end{cases}$$

  Note that, if g_{max} is less than zero, we return bottom, since there are no possible values assumed by g which are correct inputs for f (the domain of our abstract function is $\mathbb{R}^+$), so the composition of such functions cannot be done in this step ($J = \emptyset$).

Note also that, to make this operator more precise, when dealing with the steps of f intersected by $[g_{min}, g_{max}]$, we could consider only the fraction of the step domain which effectively intersects $[g_{min}, g_{max}]$ instead of all the step. More formally, for each step f_j we should consider the values it assumes in the intersection $[u_j, u_{j+1}] \cap [g_{min}, g_{max}]$.

Minimum

To achieve more precision, for this operation (and for the next one, the *maximum*) we will split each interval into subintervals, depending on (potential) intersections between f_i^- and g_i^- and between f_i^+ and g_i^+. Thus, the number of steps of the resulting abstract function is potentially increased with respect to the number of steps of the two input functions. The procedure to compute the result is as follows:

- We look for intersections between the two upper lines and the two lower ones: if f_i^- intersects g_i^-, we split the interval with respect to such intersection point. Moreover, if f_i^+ intersects g_i^+, we split the interval with respect to such intersection point, too. As a result of these two (potential) splittings, we could obtain two subintervals (in case of one intersection), three subintervals (in case of two intersections) or no subintervals (the step remains the same because there are no intersection).

- For each subinterval obtained, let $fg_{min}^- = \min(f_i^-, g_i^-)$ be the function between f_i^-, g_i^- which is *always below* the other one (we are sure it exists because, by construction, they do not intersect in the subinterval). Also, let $fg_{min}^+ = \min(f_i^+, g_i^+)$ be the function between f_i^+, g_i^+ which is always below the other one.

- For each subinterval i', return the pair of function fg_{min}^-, fg_{min}^+:

$$\mathbf{h_{i'}} = (h_{i'}^-, h_{i'}^+) = (fg_{min}^-, fg_{min}^+)$$

Note that this procedure can be used also for the last step, since it does not need information from the *next step* to perform its computations.

Maximum

The procedure to compute this abstract operation is the same as the one described for the pointwise minimum; the only difference is that, in this case, we return the line *always above* the other:

- We look for intersections between the two upper lines and the two lower ones: if f_i^- intersects g_i^-, we split the interval with respect to such intersection point. Moreover, if f_i^+ intersects g_i^+, we split the interval with respect to such intersection point, too. As a result of these two (potential) splittings, we could obtain two subintervals (in case of one intersection), three subintervals (in case of two intersections) or no subintervals (the step remains the same because there are no intersection).

- For each subinterval obtained, let $fg_{max}^- = \max(f_i^-, g_i^-)$ be the function between f_i^-, g_i^- which is *always above* the other one (we are sure it exists because, by construction, they do not intersect in the subinterval). Also, let

$fg^+_{max} = \max(f_i^+, g_i^+)$ be the function between f_i^+, g_i^+ which is always above the other one.

- For each subinterval i', return the pair of function fg^-_{max}, fg^+_{max}:

$$\mathbf{h_{i'}} = (h^-_{i'}, h^+_{i'}) = (fg^-_{max}, fg^+_{max})$$

In this case, too, we can use the same procedure for the last step.

Modulus

This operation is unary, so we only consider one function, f. For each step $[t_i, t_{i+1}]$, where f has value $(f_i^-, f_i^+) = (m, q, n, r)$, we create the corresponding step of the resulting function h as follows:

$$\mathbf{h_i} = (h_i^-, h_i^+) = \begin{cases} (0, 0, m_{ST}, q_{ST}) & \text{if } A \leq 0 \vee B \leq 0 \vee C \leq 0 \vee D \leq 0 \\ (m, q, n, r) & \text{otherwise} \end{cases}$$

where

$$\begin{array}{rl} A & = f_i^-(t_i) \\ B & = f_i^+(t_i) \\ C & = f_i^-(t_{i+1}) \\ D & = f_i^+(t_{i+1}) \\ S & = \max(|A|, |B|) \\ T & = \max(|C|, |D|) \end{array}$$

The result is derived as follows: if the area enclosed by the trapezoid is all *above* the horizontal axis (that is, the "otherwise" case of our result), then we return the same coefficients of the input value, since the values are all positive already. In the other case, we use the x-axis as lower function ($m = q = 0$) and the line linking the two maximum absolute values at the borders (S at t_i, T at t_{i+1}) as upper function.

In this operation, we need to define a special treatment for the last step:

- Let t_N be the last step, with value $\mathbf{f_N} = (m_N, q_N, n_N, r_N)$. Let A_N, B_N be the values of the lower and upper line in correspondence of the step beginning: $A_N = f_N^-(t_N), B_N = f_N^+(t_N)$.

- Find the coefficients for the new lower line, m', q' as follows:

$$\begin{array}{rl} m' & = \begin{cases} m_N & \text{if } A_N \geq 0 \wedge m_N \geq 0 \\ 0 & \text{otherwise} \end{cases} \\ q' & = \begin{cases} q_N & \text{if } A_N \geq 0 \wedge m_N \geq 0 \\ 0 & \text{otherwise} \end{cases} \end{array}$$

If the line is always above the t-axis, we return the same line, otherwise we return the constant line having value 0.

- Find the coefficients for the new upper line, n', r' as follows:

$$\begin{aligned} n' &= \max(|m_N|, |n_N|) \\ r' &= \begin{cases} r_N & \text{if } B_N \geq |A_N| \\ r_N + (|A_N| - B_N) & \text{otherwise} \end{cases} \end{aligned}$$

  The slope of the new upper line is the biggest between the absolute values of two original slopes m_N, n_N. The new upper intercept, instead, is *at least* the same of the old upper one (r_N), but it could be increased by $(|A_N| - B_N)$, to consider the fact that the absolute value of A_N could now be the higher point in correspondence of t_N (instead of B_N).

- We return the value:

$$\mathbf{h_N} = (h_N^-, h_N^+) = (m', q', n', r')$$

4.8 Experimental Results

In this section we present some experimental results about the use of TSF, and we compare them with the ones obtained by IVSF. First of all, we explore how the precision of TSF varies with the number of steps of the representation when analyzing some representative functions. Then, we consider the embedded software case study introduced in Section 4.2.

4.8.1 Varying the Number of Steps

Let us first analyze how the precision of TSF and IVSF scales with respect to the number of steps which compose the abstract values. We apply the abstraction function to a set of representative functions (namely, $\sin(x)$, x^3, e^x, and $\ln(x+1)$[4]) in the interval $[0, 10]$ varying the number of steps from 4 to 128. We measure the precision of a representation by computing the area covered by the abstract states in the Cartesian plan: the wider the area, the rougher the abstraction. Table 4.2 reports the results of this computation. The first column reports the number of steps. Then, for each analyzed function, we report the area of the TSF and IVSF abstractions, and the ratio between the two areas. For instance, if the ratio is 50%, it means that the TSF area is half of the IVSF one (i.e., it is twice more precise).

We implemented the computation of TSF in Java and we ran it on an Intel Core 2 Quad CPU 2.83 GHz with 4 GB of RAM, running Windows 7, and the Java SE Runtime Environment 1.6.0_16-b01. The execution is always extremely fast: in the worst case (function e^x), TSF requires 40 msec to compute the approximation and the area of the function for *all* the different numbers of steps. This result is not

[4]Note that, since $\ln(x)$ is not continuous in $x = 0$, we apply it to $x + 1$ in order to have a continuous function in the domain $[0, 10]$

Table 4.2: Precision of TSF and IVSF varying the number of steps

#s	$\sin(x)$			x^3			e^x			$\ln(x+1)$		
	TSF	IVSF	Ratio	TSF	IVSF	Ratio	TSF	IVSF	Ratio	TSF	IVSF	Ratio
4	4.14	15.10	27.4%	235.04	2500.00	9.4%	15894.07	55063.66	28.9%	0.63	5.99	10.5%
8	1.15	7.99	14.4%	58.64	1250.00	4.7%	4211.62	27531.83	15.3%	0.17	3.00	5.7%
16	0.30	4.01	7.6%	14.65	625.00	2.3%	1069.68	13765.92	7.8%	0.04	1.50	2.9%
32	0.08	2.04	3.7%	3.66	312.50	1.2%	268.50	6882.96	3.9%	0.01	0.75	1.5%
64	0.02	1.02	1.8%	0.92	156.25	0.6%	67.19	3441.48	2.0%	2.77E-03	0.37	0.7%
128	4.70E-03	0.51	0.9%	0.23	78.13	0.3%	16.80	1720.74	1.0%	6.93E-04	0.19	0.4%

particularly surprising since the computation mainly performs arithmetic operators for whom modern processors are quite efficient. Since the times of executions are so low, we could not notice any significant difference between TSF and IVSF even if we would expect that IVSF is faster. In addition, we did not notice any relevant memory consumption by the computation since it does not need to allocate any memory.

In all cases, TSF is more precise than IVSF. In the worst case, TSF is almost 4 times more precise (since the ratio is $\approx 29\%$) and this happens when using a small number of intervals (4). In the best case, it is approximately 330 times more precise (the ratio is 0.3%), and this happens when using a lot of intervals (128).

Why does the precision of TSF get more and more precise with respect to the one of IVSF when we increase the number of steps? IVSF uses rectangles to approximate portions of the curve, so its precision is greater when the curve is "flat" (i.e., similar to a horizontal line), while it is lower when the curve's slope is high. So, the amount of precision depends more on the kind of function than on the steps width. The precision of TSF, instead, does not depend on the curve slope, since the trapezoids are able to well approximate various kinds of slope. The precision of TSF depends only on how much the curve differs from a straight line *within a single step*. If in a single step the curve is similar to a straight line, then the error is near to zero; if in a single step the curve is very concave/convex then there is a lack of precision. When increasing the number of steps in a given domain, each step has a smaller width: for this reason, the bigger the number of steps, the more the function resembles a straight line in each single step (instead that a convex/concave curve) and the more the TSF precision increases. On the other hand, the decrease in a step width does not modify the slope of the curve in each step, and this is why the precision of IVSF does not increase as much as in the TSF domain.

4.8.2 The Integrator Case Study

Consider the case study presented in Section 4.2. Table 4.3 reports the intervals of the values of `intgrx` computed by TSF and IVSF after 104 iterations of the `while` loop. The smaller the interval, the more precise the analysis. The last column reports the ratio between the widths of the two intervals. TSF obtains more precise

Table 4.3: Values computed by TSF and IVSF on `intgrx`

# steps	TSF	IVSF	Ratio (%)
4	[-1.0263, 1.7367]	[-4.8750, 4.8750]	28
8	[-0.2772, 0.3778]	[-0.4760, 0.4760]	69
16	[-0.0740, 0.0870]	[-0.1237, 0.1237]	65
32	[-0.0188, 0.0204]	[-0.0312, 0.0312]	63
64	[-0.0047, 0.0049]	[-0.0078, 0.0078]	62
128	[-0.0012, 0.0012]	[-0.0020, 0.0020]	61

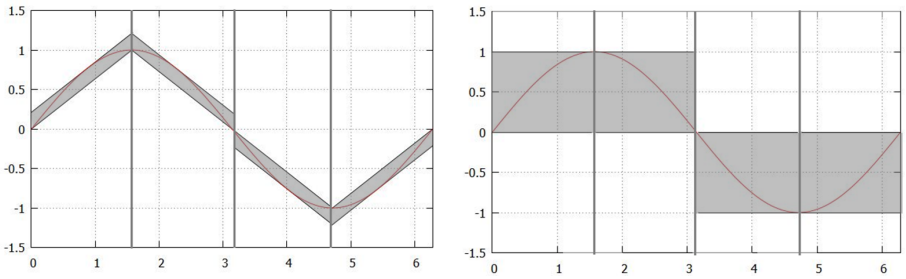


Figure 4.19: TSF (left) and IVSF (right) abstractions of $\sin(x)$, with 4 steps, on the domain $[0, 2\pi]$

results in all the cases. Note that augmenting the numbers of steps in the abstraction improves the precision of both domains, and the error ratio of TSF vs. IVSF stabilizes around 60% even if it is slightly better when augmenting the number of steps.

4.8.3 Combination of TSF with IVSF

In the two case studies, we have seen that the TSF domain is able to approximate more closely the shape of the abstracted function than IVSF. Moreover we noticed that our abstraction gets more and more precise (with respect to IVSF) every time we reduce the width of the steps in the representation. Consider, as a graphical example, Figures 4.19 and 4.20. They compare the 4-steps abstraction of $f = sin(x)$ by TSF (on the left) and by IVSF (on the right) in the intervals $[0, 2\pi]$ and $[0, \frac{\pi}{2}]$ respectively.

On the one hand, these plots make clear that in general TSF better approximates the shape of the function using trapezoids rather than the rectangles obtained with IVSF. On the other hand, IVSF gives more precise bounds on the maximum and minimum values assumed by the function because it has the advantage to preserve the stationary points of a function throughout its entire domain. Unfortunately, TSF does not preserve such information, since the trapezoids vertices might exceed these values.

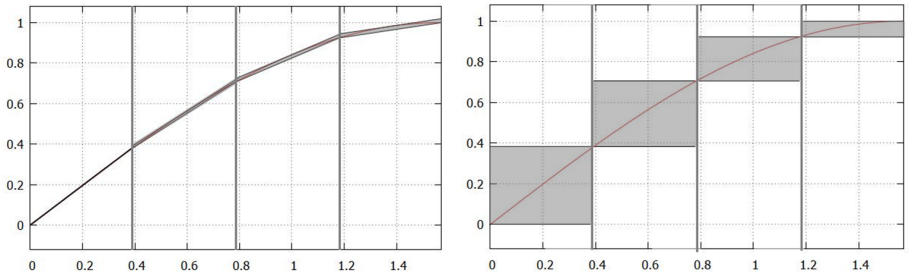


Figure 4.20: TSF (left) and IVSF (right) abstractions of $\sin(x)$, with 4 steps, on the domain $[0, \frac{\pi}{2}]$

Sometimes the right side of a trapezoid (defined on $[t_i, t_{i+1}]$) and the left side of the next one (defined on $[t_{i+1}, t_{i+2}]$) have an intersection made up by only one point: in these cases we know for sure which is the value assumed by the function in correspondence of t_{i+1} (the border between the two trapezoids). In some cases, this phenomenon happens also for IVSF, and not necessarily in correspondence of the same points as TSF. In Figure 4.19 you can see this happening at $t = \pi$ for both TSF and IVSF: the information we have is that the function has precisely value 0 in that point.

Finally, to compute the abstraction of IVSF it is necessary to know only the first derivative (other than, obviously, the original function). TSF instead requires also the second derivative, in order to locate the inflection points of the function.

To summarize, TSF is more precise than IVSF but it requires more information on the function and it does not necessarily preserve the minimum/maximum bounds (while IVSF does). Since both domains track interesting information, it could be useful in some applications (especially the ones where the stationary points of the function have some relevancy), to consider a combination (i.e., the Cartesian product or the reduced product presented in Section 2.11) of these two domains, to improve the precision of the overall analysis and in particular to precisely bound the minimal and maximal values of the function. For instance, consider the Cartesian product of these two domains: the application of TSF $\times$ IVSF to the example of Figure 4.19 discovers that, at $\frac{\pi}{2}$, the abstracted function has exactly value 1, since (i) TSF tracks that its minimal value is 1, and (ii) IVSF tracks that its maximal value is 1. The same happens in the analysis of our case study: we can *precisely* abstract the values of $\sin(x)$ when it is at its maximum (or minimum) by taking the intersection of the values approximated by TSF (that computes that the values are greater or equal to 1 in the maximum, and less or equal than -1 in the minimum) and IVSF (that computes that the values are less or equal to 1 in the maximum, and greater or equal than -1 in the minimum).

4.9 Related Work

As already explained in Section 4.1, hybrid systems are models for complex physical systems where both discrete and continuous behaviours are important. Many applications are safety-critical, including car, railway, and air traffic control, robotics, physical-chemical process control, and biomedical devices. Hybrid systems verification is then an important and very challenging problem, because of the interaction between discrete and continuous dynamics.

Continuity is not a new topic in static analysis, even if it has been only partially (and, most of all, separately from *hybrid systems*) explored so far. In fact, there exists some work regarding continuous functions in general, but it is not thought for the specific purpose of verifying the behaviour of hybrid systems. For example, a useful domain theoretic characterization of continuous function can be found in [70], but this work only describes the continuous functions at the concrete level, and there is nothing involving their sound abstraction.

Regarding continuity analysis of programs, [97] was the first to argue for a testing methodology for Lipschitz-continuity of software. [135] introduced a type system that verifies the Lipschitz-continuity of functional programs. This system does not handle control flow and does not consider any application other than differential privacy. The paper [29] recently proposed a qualitative program analysis to automatically determine if a program implements a continuous function. The practical motivation they address is the verification of robustness properties of programs whose inputs can have small amounts of error and uncertainty. This work was further extended by [30] to quantify the robustness of a program to uncertainty in its inputs. Since they decompose the verification of robustness into the independent sub-problems of verifying continuity and piecewise robustness, they do not add new insights to the specific problem of program continuity with respect to [29]. Our treatment of continuous functions should be applicable to this particular setting (continuity of programs) as well.

Note also that a Trapezoid Step Function is a sequence of trapezoids, one for each step. But a TSF can be seen as a pair of PieceWise Linear (PWL) functions as well, where one PWL function bounds the approximated continuous functions from above and the other one bounds them from below. There exists an extensive literature about PWL functions, since they played an important role in approximation, regression and classification. One of the biggest problems concerns their explicit representation in a closed form [37, 107]. Another important issue is to find a PWL approximation of a certain function to minimize or bound the overall area (or the distance in each point) between the original function and the approximation [34, 102, 145]. Instead, our approach provides a *sound* approximation of a function rather than bounding the error of its representation.

On the other hand, the verification of hybrid systems is certainly a common topic, but previous works on Abstract Interpretation-based strategies for such systems mainly involve the analysis of hybrid automata [92, 95] without specifically

considering the continuous environment of the system. [71] introduced domain-specific abstract domains for digital filters (in the context of ASTREE [18]), but did not provide a generic treatment of continuous functions and their abstraction.

Given this context, to the best of our knowledge, IVSF has been the first formalism to allow the integration of the continuous environment in an abstract interpretation of embedded software. A static analyzer based on that formalism has been implemented [22] in order to consider the interactions between the program and the physical environments on which it acts. The analyzer (HybridFluctuat) is based upon Fluctuat, a tool developed by CEA which aims at analyzing the numerical precision and stability of complex algorithms. However, as far as we know, IVSF theoretical framework has not been refined further since then.

In Section 4.8 we compared extensively the precision of our approach with respect to IVSF and we showed that TSF is quite more precise than IVSF when abstracting a representative set of continuous functions and when applied to a representative case study.

4.10 Discussion

In this chapter we tackled the issue of approximating the physical environment of embedded software through Abstract Interpretation, improving the current state-of-the art defined by [24].

To this purpose, we introduced the Trapezoid Step Functions domain (TSF), a new abstract domain aimed at approximating the behaviours of continuous functions. We formally defined the structure of the domain together with the lattice and widening operators, proving the soundness of our approach. We also introduced a sound abstraction function that, given a concrete function, builds up its abstract representation in TSF. Finally, we presented some experimental results about the precision of TSF, and we compared them with the ones obtained by the Interval Valued Step Functions domain of [24] (IVSF), that, at the best of our knowledge, is the most refined domain in the context of the abstraction of functions in continuous environments. The experimental results underline that TSF obtains abstractions that are more precise than the ones obtained by IVSF.

Note that we also defined the abstract semantics of arithmetic operations involving continuous functions, even though such operations have not been used in the section about experimental results and their correctness has not been proved (yet). This semantics has been developed for the specific purpose of applying our domain to the cost analysis of code [11] and it is the first step in that direction. However, in the context of cost analysis, it is necessary to extend our approach to multivariate functions (TSF only considers univariate ones) in order to allow more complex cost models. We already started working on this new abstract domain and are very close to its completion.

5

The Parametric Hypercubes Abstract Domain

In this chapter we focus on the last of the three goals of our thesis, i.e. creating a new approach to deal with relationships between variables, keeping into account the specific features of physics simulations inside games software.

Computer Games Software is a fast growing industry, with more than 200 million units sold every year, and annual revenue of more than 10 billion dollars. According to the Entertainment Software Association (ESA), more than 25% of the software played concerns sport, action, and strategy games, where physics simulations are the core of the product, and compile-time verification of behavioural properties is particularly challenging for developers. Physics simulations are especially demanding to analyze because they manipulate a large amount of interleaving floating point variables. Therefore, this application domain is an interesting workbench to stress the trade-off between accuracy and efficiency of abstract domains for static analysis.

In this chapter, we introduce Parametric Hypercubes, a novel disjunctive non-relational abstract domain. Its main features are: (i) it combines the low computational cost of operations on (selected) multidimensional intervals with the accuracy provided by lifting to a power-set disjunctive domain, (ii) the compact representation of its elements aids the space complexity of the analysis, and (iii) the parametric nature of the domain provides a way to tune the accuracy/efficiency of the analysis by just setting the widths of the hypercubes sides. The first experimental results on a representative Computer Games case study outline both the efficiency and the precision of the proposal.

Similarly to Chapters 3 and 4, this chapter follows the structure explained in Section 1.6: Section 5.1 introduces the problem and some basic terminology specific of games software (even though an entire section of notation is not required), Section 5.2 presents the case study which we use to experiment with our approach and Section 5.3 defines the language syntax supported by our analysis. Section 5.4 explains the concrete domain of reference, while Sections 5.5 and 5.6 formally define the abstract domain and semantics, respectively. Section 5.7 deals with issues related to the practical use of the analysis. Section 5.8 contains the experimental results of our analysis applied to the case study. Section 5.9 presents the related work. In Section 5.10 we hint at other possible applications of our abstract domain outside

[0]This chapter is partially derived from [48].

games software, by making a concrete example. Finally, Section 5.11 concludes.

Contents

5.1 Introduction

Context

The video game industry (sometimes referred to as the interactive entertainment industry) is the economic sector involved with the development, marketing and sales of video games. It encompasses dozens of job disciplines and employs thousands of people worldwide. Today, the video game industry has a major impact on the economy through the sales of major systems and games: for example, when the game *Call of Duty: Black Ops* was launched, it took in over $650 million of sales in the game's first five days and set a five-day global record for a movie, book or videogame. According to industry statistics [10], the U.S. interactive entertainment software publishing industry achieved retail sales of $10.5 billion in 2009. Unit sales of computer and video games have increased from 226.3 million in 2005 to more than 273 million in 2009.

A *video game* can be seen as a simulation of a fictional world: every game is populated by objects and characters which interact in some way with the user and between themselves. An important feature of such entities is their movement: a game where everything is always still would be quite uninteresting. Physics is an integral part of most modern video game design, especially if the game is in 3D. Computer animation physics (or *game physics*) involves the introduction of the laws of physics into a simulation or game engine, particularly in 3D computer graphics, for the purpose of making the effects appear more realistic to the observer. Typically, simulation physics is only a close approximation of real physics, and computation is performed using discrete values. Implementing game physics is quite an important phase in the development process of a game, and for this reason there are a lot of resources (books, websites, etc.) which handle this subject, for example [26, 68, 119].

Video game development is the software development process by which a video game is produced. Game development is undertaken by a game developer - ranging from an individual to a large company. In the early era of home computers and the first video game consoles, a single programmer could handle almost all the tasks of developing a game (programming, graphical design, sound effects, etc), taking only a few weeks to complete its development. However, as computing and graphics power increased, so did the size of development teams, as larger staff were needed to address increasing graphical and programming complexity. Now, budgets for games production can easily reach tens of millions of dollars, even if middleware and pre-built game engines are used to reduce development time. Most professional games require one to three years to develop, further increasing the strain on budgets. A consistent part of the budget of a videogame production is spent on testing, and on consequent bug fixing. In fact, testing and quality assurance are vital for modern, complex games: a game shipped with bugs can result in customers dissatisfaction and failure to meet sales goals, causing critical economic damage for game software manufacturers. In addition, given the formats that game titles are shipped in, such

as ROM cartridge and DVD-ROM formats, they do not allow for bug fixes on the fly. Until some years ago, the ROM cartridges and DVDs needed to be physically collected and updated, making any bugs found post-release extremely costly to fix. Now, consoles are all connected to the internet, so patches and updates can be simply downloaded: however, the size of such patches is usually quite big (up to 2GB) and players are never happy to spend a lot of time waiting before they can play again. For these reasons, if a bug makes it into the market, it can severely tarnish a game vendor's brand. [8]

The quality assurance is carried out by game testers. In the early days of computer and video games, no more than one or two testers were required due to the limited scope of games. In some cases, the programmers alone could handle all the testing. As games became more and more complex, though, the testing phase increased in importance and with it the role of the game tester. A game tester analyzes video games to document software defects as part of a quality control. Testing is a highly technical field requiring computing expertise, and analytic competence. The testers ensure that the game falls within the proposed design: it both *works* and *is entertaining*. This involves testing of all features, compatibility, localization, etc. However, testing is very expensive and for this reason it is often actively utilized only towards the completion of the project, while it would be necessary throughout the whole development process. At SEGA [1], in fact, developers researched advanced software development tools that could help them to find and fix defects early. To help get ahead of defects and ensure code quality in their games prior to release, SEGA chose to employ *Coverity Static Analysis* [5] as a source code analysis tool (after examining it together with two other static analysis tools). Since the initial deployment, Coverity's solution has been implemented in five additional SEGA projects and now forty developers are using Coverity Static Analysis to analyze development programs on a daily basis. According to the case study report [8], by using Coverity Static Analysis, SEGA developers can find and fix bugs at an early stage, which saves time and labor in the test process, improving the productivity of the entire development process by approximately 20%. This example clearly shows how static analysis can be effective in significantly reducing testing time on games software. Another testimony from the game software field about the importance of static analysis comes from an in-depth blog article written by John Carmack [2] [9]. In this article, he describes his experiences in trying various static analysis tools to find bugs in a large code base. A brief recap of his findings and opinions follows:

- *Coverity* has a very high signal to noise ratio: most of the issues highlighted were clearly incorrect code that could have serious consequences. However, the

[1] SEGA Corporation is a Japanese multinational video game software developer.

[2] John Carmack is an American game programmer and the co-founder of *id Software*. Carmack was the lead programmer of the *id* video games Commander Keen, Wolfenstein 3D, Doom, Quake, Rage and their sequels. Carmack is best known for his innovations in 3D graphics, and is also the founder and lead engineer of Armadillo Aerospace.

cost of the licence is very high (thus affordable only if one has a big budget available).

- *Microsoft /analyze*, a static analysis functionality which has been incorporated into the Xbox 360 SDK, making it available to every Xbox 360 developer at no extra charge. This tool only performs local analysis, but it poured out mountains of errors nonetheless (far more than Coverity). However, there were also lots of false positives. The disadvantage of this tool is that it works only on the Xbox 360 code (so it does not cover the PC and PS3 specific platform code, and the code for all the tools and utilities that only ran on the PC).

- *PVS-Studio* has a good integration with Visual Studio and a convenient demo mode. When tried by Carmack, it pointed out a number of additional important errors (even on code that was already completely clean to /analyze). As a nice additional feature, PVS-Studio also points out a number of things that are common patterns of programmer errors. The problem of this tool is its performance: it is terribly slow.

- *PC-Lint* (coupled with Visual Lint for IDE integration) is a tool which on the one hand can be configured to do just about anything, but on the other hand is not very friendly and requires a lot of hassle to make it work. However, it found out new errors of significance.

As we can see from Carmack's overview, every tool has its features and its downsides. The common point is that every analyzer found (a lot of) significant errors. In the words of Carmack: "*It is impossible to do a true control test in software development, but I feel the success that we have had with* ***code analysis*** *has been clear enough that I will say plainly it* ***is irresponsible not to use it****. If you are developing commercial software, buying static analysis tools is money well spent.*".

State of the art

The main limitation of the static analysis tools cited above (Microsoft /analyze, PSV-Studio, etc.) is that they can only verify syntactic *structural properties* of programs, for example: buffer overruns, uninitialized memory, null pointer dereferences, memory and resource leaks, conformance to coding guidelines, race conditions, deadlocks, needless synchronization, and so on. While verifying this kind of properties is certainly very important for any kind of program, for physics simulations it also crucial to consider *behavioural properties*, i.e., if to check if the program behaves according to a given specification. Interesting properties on physical programs are, for example, the insurance that a rocket reaches a stable orbit, or that a bouncing ball arrives at a certain destination.

It is not easy to find tools that understand automatically *what* a physical simulation does, because, usually, these kind of programs are characterized by:

- a `while` loop which goes on endlessly: most of the times, a simulation consists in the initialization of the state (i.e., the variables which compose the simulated world) followed by an infinite `while` loop which computes the numerical integration over time (i.e., the inductive step of the simulation). Such loop is executed until the game is stopped.

- a complex state made up by multiple real-valued variables. The variables of a physics simulation are real-valued, because they represent continuous values that map directly to physical aspects of the real world, like positions, velocities (speed plus direction), and accelerations.

- strong dependencies among variables. The variables of a simulation are strongly inter-related, because the simulation often makes decisions based on the values of particular variables. For example, the velocity of an object changes abruptly when there is a collision, which depends on the object position. Similarly, the position changes accordingly to the velocity, which in turn depends on the acceleration which may derive from the position (for a gravitational field) or from other parameters.

Given these features, to statically prove interesting properties on physical simulations we need to precisely track relationships between variables. However, traditional approaches to static analysis are not best suited to deal with these kind of properties. On the one hand, non-relational domains are too approximate. On the other hand, the computational cost of sophisticated relational domains like Polyhedra [62] or Parallelotopes [14] is too high, and their practical use in this context becomes unfeasible. Model Checking techniques, too, have performance issues (due to the well-known state explosion problem). In Section 5.9 we will make a more detailed comparison between our proposal and other existing techniques, coming from both model checking and Abstract Interpretation.

Contribution

In this chapter, we introduce Parametric Hypercubes, a novel disjunctive non-relational abstract domain. Its main features are: (i) it combines the low computational cost of operations on (selected) multidimensional intervals with the accuracy provided by lifting to a power-set domain, (ii) the compact representation of its elements allows to limit the space complexity of the analysis, and (iii) the parametric nature of the domain provides a way to tune the trade-off between accuracy and efficiency of the analysis by just setting the widths of the hypercubes sides. The domain can be seen as the combination of a suite of well-known techniques for numerical abstract domain design, like disjunctive power-set, and conditional partitioning. The most interesting points of our work are: (i) the approach: the design of the domain has as starting point the features of the application domains, (ii) the self-adaptive parametrization: a recursive algorithm is applied to refine the initial

```
1  px = rand([0.0, 10.0]), py = rand([0.0, 50.0])
2  vx = rand([0.0, 60.0]), vy = rand([-30.0, -25.0])
3  dt = 0.05, g = -9.8, k = 0.8
4
5  while (true) do
6     if( py >= 0.0 ) then
7       (px, py) = (px + vx * dt, py + vy * dt)
8       (vx, vy) = (vx, vy + g * dt)
9     else
10      (px, py) = (px + vx * dt, 0.0)
11      (vx, vy) = (vx, -vy) * k
```

Figure 5.1: Bouncing ball case study

set of parameters in order to improve the accuracy of the analysis without sacrificing the performance, and (iii) the novel notion of "offset" that allows to narrow the lack of precision due to the fixed width of intervals. The analysis has been implemented, and it shows promising results in terms both of efficiency and precision when applied to a representative case study of Computer Games Software.

5.2 Case Study

In this chapter, we chose to use the case study reported in Figure 5.1: it is inspired by the bouncing ball hybrid system described in Section 4.1.

The program generates a bouncing ball that starts at the left side of the screen (even though the exact initial position is not fixed), and it has a random initial velocity. The horizontal direction of the ball is always towards the right of the screen, since $\texttt{vx} \geq 0$.Whenever the ball reaches the bottom of the screen, it bounces (i.e., its vertical velocity is inverted). When the ball reaches the right border of the screen, it disappears. We want to verify that T seconds after the generation of the ball, such ball has already exited from the screen (we call this *Property 1*).

The structure of this program respects the generic structure of a physics simulation, as explained in Section 5.1. The meanings of the variables are as follows:

- $(\texttt{px}, \texttt{py})$ represents the current position of the ball in the screen, and its initial values are generated randomly;
- $(\texttt{vx}, \texttt{vy})$ represents the current velocity of the ball, and its initial values are generated randomly as well;
- `dt` represents the time interval between iterations of the loop. This value is constant and known at compile time ($\texttt{dt} = 1/20 = 0.05$ considering a simulation running at 20 frames per second);

```
1  balls = Set.empty
2  dt = 0.05, creationInterval = 3.0, timeFromLastCreation = 0.0
3  while (true) do
4          foreach ball in balls
5                  updateBall(ball)
6          if(timeFromLastCreation >= creationInterval)
7                  generateNewBall()
8                  timeFromLastCreation = 0.0
9          else
10                 timeFromLastCreation += dt
```

Figure 5.2: Bouncing ball generation

- `g` represents the force of gravity (-9.8);
- `k` represents how much the impact with the ground decreases the velocity of the ball.

The `while` loop updates the ball position and velocity. To simulate the bouncing, we update the horizontal position according to the rule of uniform linear motion, while we force the vertical position to zero when the ball touches the ground and we invert the vertical velocity. In addition, we decrease both the horizontal and vertical velocity through the constant factor `k`, to consider the force which is lost in the impact with the ground.

Verifying Property 1 on this program has a significant practical interest, since it is a basic physics simulation which can be used in many contexts [69]. For instance, consider the program in Figure 5.2, where `updateBall(b)` moves the ball `b` (through the body of the while loop of Figure 5.1) and `generateNewBall()` creates a new ball (with the values of the initialization of Figure 5.1). It discretely generates bouncing balls on the screen. The interval between the creation of two balls (`creationInterval`) is constant and known at compile time.

Proving *Property 1* on the program in Figure 5.1 means that a single ball will have exited the screen after T seconds. In addition, in the program of Figure 5.2, we generate one ball each `creationInterval` seconds. This means that, having verified *Property 1*, we can guarantee that *a maximum of* $\lceil \frac{T}{\texttt{creationInterval}} \rceil$ *balls will be on the screen at the same time.* Such information may be useful for performance reasons (crucial in a game), since each ball requires computations for its rendering and updating.

Non-disjunctive or non-relational static analyses are not properly suited to verify *Property 1.* Consider for example the Interval domain where every variable of the program is associated to a single interval. After a few iterations, when the vertical position possibly goes to zero, the analysis is not able to distinguish which branch of the `if` $-$ `then` $-$ `else` to take anymore. In this case, the lub operator makes the

$$V \in \mathcal{V}, I \in \mathcal{I}, c \in \mathbb{R}$$
$$E := c|\texttt{rand}(I)|V|E \texttt{ aop } E \text{ where } \texttt{aop} \in \{+, -, \times, \div\}$$
$$B := E \texttt{ bop } E|B \texttt{ and } B|\texttt{not } B|B \texttt{ or } B \text{ where } \texttt{bop} \in \{\geq, >, \leq, <, \neq\}$$
$$P := V = E|\texttt{if}(B) \texttt{ then } P \texttt{ else } P|\texttt{while}(B)\ P|P; P$$

Figure 5.3: Syntax

vertical velocity interval quite wide, since it will contain both positive and negative values. After that, the precision gets completely lost, since the velocity variable affects the position and vice-versa. On the other hand, the accuracy that would be ensured by using existing disjunctive domains has a computational cost that makes this approach unfeasible for practical use.

5.3 Language Syntax

Let $\mathcal{V}$ be a finite set of variables, and $\mathcal{I}$ the set of all real-valued intervals. Figure 5.3 defines the language supported by our analysis.

We focus on programs dealing with mathematical computations over real-valued variables. Therefore, we consider expressions built through the most common mathematical operators (sum, subtraction, multiplication, and division). An arithmetic expression can be a constant value ($c \in \mathbb{R}$), a non-deterministic value in an interval ($\texttt{rand}(I)$ where $I \in \mathcal{I}$), or a variable ($V \in \mathcal{V}$). We also consider boolean conditions built through the comparison of two arithmetic expressions. Boolean conditions can be combined as usual with logical operators (and, or, not). As for statements, we support the assignment of an expression to a variable, $\texttt{if} - \texttt{then} - \texttt{else}$, `while` loops, and concatenation. Even though this syntax is simple and limited, many physical simulations can be built through it [20], since their complexity lies mostly in their logic and not in the used constructs.

5.4 Concrete Domain and Semantics

Since we want to abstract a set of real-valued variables (i.e., all the non-constant variables of the program as a whole), the concrete domain $\mathcal{R}$ is defined as the power-set of environments of real values. As explained in Section 2.4, when the lattice is the power-set of a set, the other operators immediately follow and the complete definition of the lattice $\mathcal{R}$ is:

$$\mathcal{R} = \langle \wp(\mathsf{Vars} \to \mathbb{R}), \subseteq, \cup, \cap, \mathsf{Vars} \to \mathbb{R}, \emptyset \rangle$$

Table 5.1: Concrete semantics

$$
\begin{array}{ll}
\mathbb{V}[\![\mathtt{c}]\!]() & = \{\mathtt{c}\} \\
\mathbb{V}[\![\mathtt{rand_I}]\!]() & = \{\mathtt{r} : \mathtt{r} \in \mathtt{I}\} \\
\mathbb{V}[\![\mathtt{V}]\!](\Sigma) & = \{\sigma[\mathtt{varIndex(V)}] : \sigma \in \Sigma\} \\
\mathbb{V}[\![\mathtt{aop}]\!](\mathsf{R_1}, \mathsf{R_2}) & = \{\mathtt{r} : \exists \mathtt{r_1} \in \mathsf{R_1}, \mathtt{r_2} \in \mathsf{R_2} \wedge \mathtt{r} = \mathtt{r_1\ aop\ r_2}\} \\
 & \text{where } \mathtt{aop} \in \{+, -, \times, \div\}
\end{array}
$$

We can now define the concrete semantics of the language introduced in Section 5.3. For the statements operations and boolean conditions we refer to the usual semantics of the classical Abstract Interpretation framework, applied to environments. We specify here only the concrete semantics of mathematical expressions, which is formalized in Table 5.1.

We define the semantics $\mathbb{V}$ that, given the statement and eventually the values of the arguments of the statement, returns a set of real values resulting from that operation. In particular:

- the evaluation of a constant $\mathtt{c}$ returns a singleton containing only $\mathtt{c}$ itself;
- the extraction of a random value from an interval ($\mathtt{rand}(\mathrm{I})$) returns the set containing all values of the interval in input;
- the extraction of the value of a specific variable V returns all the values which V assumes in each environment σ of the concrete set of environments Σ (where $\sigma[i]$ represents the value of the i-th variable of the environment, and $varIndex(x)$ returns the environment index associated to the variable x);
- $\mathtt{aop} \in \{+, -, \times, \div\}$ returns, for every pair of real values (one from the first set in input, one from the second one), the value resulting from their combination through $\mathtt{aop}$. For example, when $\mathtt{aop}$ is $+$, we return a set containing all the summations of all possible pairs from the two input sets.

5.5 Abstract Domain

Intuitively, an abstract state of the Parametric Hypercubes domain ($\mathcal{H}$) tracks disjunctive information relying on floating-point intervals of fixed width. A state of $\mathcal{H}$ is made by a set of hypercubes of dimension $|\mathsf{Vars}|$. Each hypercube has $|\mathsf{Vars}|$ sides, one for each variable, and each side contains an abstract non-relational value for the corresponding variable. Each hypercube represents a set of admissible combinations of values for all variables.

The name *Hypercubes* comes from the geometric interpretation of the elements of $\mathcal{H}$. The concrete state of a program with variables in Vars is an environment

in $\mathsf{Vars} \to \mathbb{R}$. This can be isomorphically represented by a tuple of values where each item of the tuple represents a program variable. Seen in this way, the concrete state corresponds, geometrically, to *a point* in the $|\mathsf{Vars}|$-dimensional space. Each dimension of the space represents the possible values that the corresponding variable of the program can assume. The concrete trace of a program is a sequence of points in such space (one for each state of the trace). The hypercubes of our domain $\mathcal{H}$ are *volumes* in the same $|\mathsf{Vars}|$-dimensional space. Each side of the hypercube is the concretization of the abstract value of the corresponding variable, and thus it corresponds to a set of values in that dimension of the space. The concretization of an hypercube is the set of all the points contained in its volume. A state in $\mathcal{H}$ is composed by a set of hypercubes: its concretization is the union of all the volumes of its hypercubes.

5.5.1 Lattice Structure

An abstract state of $\mathcal{H}$ tracks a *set* of hypercubes, and each hypercube is represented by a tuple of abstract values. The dimension of these tuples is equal to the number of program variables: this means that each variable is associated to a given item of the tuple (i.e., to a specific side of the hypercube). Consider, for instance, a program in which $\mathsf{Vars} = \{x_1, x_2\}$. In this case, the hypercubes of $\mathcal{H}$ are 2D-rectangles. In particular, the two sides of a single hypercube are two abstract values, one for x_1 and one for x_2.

A priori, our approach is modular w.r.t. the non-relational abstract domain we adopt to approximate the values of single variables inside an hypercube. However, since we are targeting the analysis of physics simulations, we focus on the abstraction of floating-point variables through intervals of real values.

A set of hypercubes allows us to track disjunctive information, and this is useful when the values of a variable are clustered in different ranges: instead of having a very big interval to cover them all (and which would cover also a lot of invalid values), we use two (or more) smaller intervals. Since it would be particularly expensive to perform all the lattice operators pointwisely, we partition the possible values into intervals of fixed width. As an example, suppose that the initial vertical velocity of the balls of our case study ranges between 50.0 and 60.0 or between -60.0 and -50.0. A single interval would approximate these values with $[-60.0..60.0]$, while with our approach we track two intervals, $[-60.0..-50.0]$ and $[50.0..60.0]$ (with fixed width 10.0), which distinguish between balls thrown downwards and balls thrown upwards.

The performance of this domain, though, becomes a crucial point, because the number of possible hypercubes in the space is potentially exponential with respect to the number of partitions along each spatial axis. First of all, the complexity is reduced by: (i) the use of a *fixed* width for each variable; (ii) partitioning the possible intervals; and (iii) by the efficiency of set operators on tuples. Then, another performance booster is the use of a smart representation for intervals: in order to

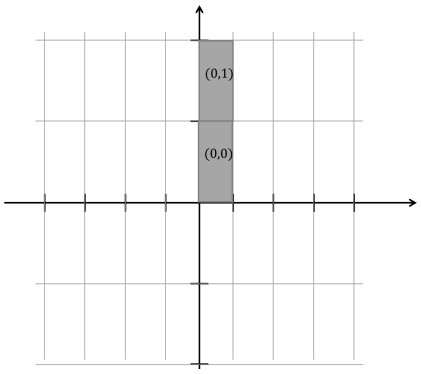


Figure 5.4: The abstract state of the case study after the initialization of the variables (focusing the attention only on `px`, `py`, when their widths are, respectively, 10.0 and 25.0)

store the specific interval range we just use a single integer representing it. This is possible because each variable x_i is associated to an interval width (specific only for that variable), which we call w_i and which is a parameter of the analysis. Each width w_i represents the width of all the possible abstract intervals associated to x_i. More precisely, given a width w_i and an integer index m, the interval uniquely associated to the variable x_i is

$$[m \times w_i..(m+1) \times w_i]$$

Notice that the smaller the width associated to a variable, the more granular and precise the analysis on that variable (and the heavier computationally the analysis). In Section 5.6 we will show how to compute and adjust automatically the widths.

Example

Consider the case study of Section 5.2 and in particular the two variables `px` and `py`. Suppose that the widths associated to such variables are, respectively, $w_1 = 10.0$ and $w_2 = 25.0$. The hypercubes in this case are 2D-rectangles that can be represented on the Cartesian plane. Each side of a hypercube is identified by an integer index, and a 2D hypercube is then uniquely identified by a pair of integers. For instance, the hypercube $h_1 = (0, 1)$ represents `px` $\in [0.0..10.0]$ and `py` $\in [25.0..50.0]$, while the hypercube $h_2 = (0, 0)$ associates `px` to $[0.0..10.0]$ and `py` to $[0.0..25.0]$. Figure 5.4 depicts the two hypercubes h_1 and h_2 associated to the initialization of the case study. Instead, Figure 5.5 depicts the six hypercubes obtained after executing the first iteration of the `while` loop. The ball is moving towards the right of the screen and is going downwards: this is coherent with the fact that the horizontal velocity is certainly positive (between 0.0 and 60.0), while the vertical velocity is certainly negative (between -30.0 and -25.0).

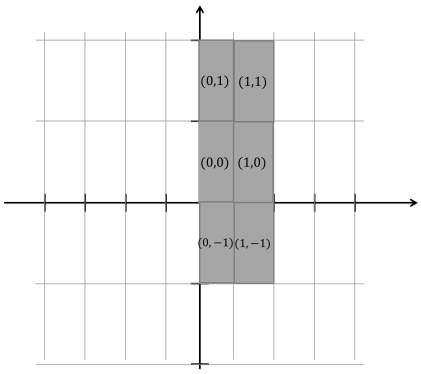


Figure 5.5: The abstract state of the case study after the first iteration of the loop (focusing the attention only on px, py, when their widths are, respectively, 10.0 and 25.0)

Domain Definition

We now formalize our abstract domain. Each abstract state is a set of hypercubes, where each hypercube is composed by $|\mathsf{Vars}|$ integer numbers. The abstract domain is then defined by

$$\mathcal{H} = \wp(\mathbb{Z}^n)$$

where $n = |\mathsf{Vars}|$. The definition of lattice operators relies on set operators (as explained in Section 2.4 about the power-set lattice): the *partial order* is defined through set inclusion, the *lub* and *glb* are set union and set intersection, respectively, while the *bottom* and *top* elements are the empty set and the set containing all possible n-dimensional hypercubes, respectively. Formally:

$$\widehat{\mathcal{H}} = \langle \wp(\mathbb{Z}^n), \subseteq, \cup, \cap, \emptyset, \mathbb{Z}^n \rangle$$

Lemma 5.5.1. *$\widehat{\mathcal{H}} = \langle \wp(\mathbb{Z}^n), \subseteq, \cup, \cap, \emptyset, \mathbb{Z}^n \rangle$ is a complete lattice.*

Proof. The proof follows immediately by basic properties of set operators. □

5.5.2 Abstraction and Concretization Functions

We now define the abstraction and concretization functions in a parametric way, i.e. independently from the kind of abstract domain (denoted here by $\mathcal{A}$) used to abstract the single variables of the program. Let n be the number of variables of the program ($n = |\mathsf{Vars}|$). Let $\sigma \in \mathbb{R}^n$ be a tuple and $\sigma_i \in \mathbb{R}$ be the i-th element of such tuple.

The *abstraction function* maps a set of concrete environments into a set of hypercubes, i.e. a set of tuples of abstract values belonging to the generic abstract domain $\mathcal{A}$:

$$\alpha_{\mathcal{H}} : \wp(\mathsf{Vars} \to \mathbb{R}) \to \wp(\mathcal{A}^n)$$

The formal definition of $\alpha_{\mathcal{H}}$ follows:

$$\alpha_{\mathcal{H}}(E) = \{h : \exists e \in E : \forall v \in \mathsf{Vars} : \alpha_{\mathcal{A}}(e[v]) = h_{varIndex(v)}\}$$

Given a set of environments E, we associate to it the set of hypercubes $\{h\}$ such that there is at least one environment e for which the abstraction of the concrete value of each variable ($\alpha_{\mathcal{A}}(e[v])$, where $e \in E$ is an environment, $v \in \mathsf{Vars}$ is a variable and $\alpha_{\mathcal{A}}$ is the abstraction function of $\mathcal{A}$) corresponds to the side of the hypercube assigned to such variable ($h_{varIndex(v)}$, where the function $varIndex : \mathsf{Vars} \to \mathbb{N}$, given a variable, returns its index in the tuples which compose the elements of $\mathcal{H}$).

We now define the *concretization function*. Let

$$\gamma_{\mathcal{A}} : \mathcal{A} \to \wp(\mathbb{R})$$

be the concretization function of abstract values of the non-relational abstract domain $\mathcal{A}$, and

$$getAbsValue_v : \mathbb{N} \to \mathcal{A}$$

be the function that, given an integer index, returns the abstract value (in the domain $\mathcal{A}$) which corresponds to that index inside the hypercube tuple v. Then, the function

$$\gamma_{\mathsf{Val}} : \wp(\mathcal{A}^n) \to \wp(\mathbb{R}^n)$$

concretizes a set of hypercubes to a set of vectors of n floating point values. The formal definition of γ_{Val} follows:

$$\gamma_{\mathsf{Val}}(\mathsf{V}) = \{\sigma : \exists v \in \mathsf{V} : \forall i \in [1..n] : \sigma_i \in \gamma_{\mathcal{A}}(getAbsValue_v(i))\}$$

where $\mathsf{V} \in \wp(\mathcal{A}^n)$ is a set of hypercubes. Finally, based on γ_{Val}, we can define the function $\gamma_{\mathcal{H}}$, which maps a subset V of $\wp(\mathcal{A}^n)$ into an environment. The function

$$\gamma_{\mathcal{H}} : \wp(\mathcal{A}^n) \to \wp(\mathsf{Vars} \to \mathbb{R})$$

concretizes the hypercubes domain. Formally:

$$\gamma_{\mathcal{H}}(\mathsf{V}) = \{[\mathtt{x} \mapsto \sigma_{varIndex(\mathtt{x})} : \mathtt{x} \in \mathsf{Vars}] : \sigma \in \gamma_{\mathsf{Val}}(\mathsf{V})\}$$

The function $\gamma_{\mathcal{H}}$ maps the vectors returned by γ_{Val} into concrete environments relying on the function *varIndex* defined above for the abstraction function.

5.5.3 Widening Operator

The domain described so far does not ensure the convergence of the analysis. In fact, a `while` loop may add new hypercubes with increased indices at each iteration, and the dimension of the abstract state (i.e., the hypercubes set) would increase at each iteration without converging. Thus, we need a way to force the convergence of the analysis.

Given our abstract state representation, we fix for each variable of the program a maximum integer index n_i such that n_i represents the interval

$$[n_i \times w_i .. + \infty]$$

The same happens symmetrically for negative values (i.e., the index $-n_i$ represents the interval $[-\infty .. - n_i \times w_i]$). In this way, the set of indices of a given variable is finite: then, the resulting domain has finite height and the analysis is convergent without the need to employ a specific widening operator.

This approach may seem too rough since we establish the bounds of intervals before running the analysis. However, this allows us to control the number of possible intervals in our hypercubes, and this is particularly important for the efficiency of the overall analysis. In addition, when analysing physics simulations we can use the initialization of variables and the property to verify in order to establish convenient bounds for the intervals. For instance, in the case study presented in Section 5.2 we are interested in checking if a ball stays in the screen, that is, if `px` is greater than zero and less than a given value `w` representing the width of the screen. Since we are only interested in proving that the balls exit the screen, we can abstract together all the values that are greater than `w`.

Observe that more sophisticated widening operators could be defined as an alternative to the adopted solution described above, but this could affect the performance of the resulting analysis.

5.5.4 Enhancing Precision: Offsets

In this subsection we present a modification of the abstract elements definition, in order to increase the precision of the domain. In fact, a big loss of precision may occur due to the fact that hypercubes proliferate too much, even using small widths. Consider, for example, the statement `x = x + 0.01` (which is repeated at each iteration of the `while` loop) with 1.0 as the width associated to `x`. If the initial interval associated to `x` was [0.0..1.0], after the first iteration we would obtain two intervals ([0.0..1.0] and [1.0..2.0]) because the resulting interval would be [0.01..1.01], which spans over two fixed-width intervals. For the same reason, after the second iteration we would obtain three intervals ([0.0..1.0], [1.0..2.0] and [2.0..3.0]) and so on: at each iteration we would add one interval.

In order to overcome these situations, we further improve the definition of our domain: in each hypercube, each variable v_i (associated to width w_i) is related to (other than an integer index i representing the fixed-width interval $[i \times w_i .. (i+1) \times w_i]$) a specific offset

$$(o_m, o_M)$$

inside such an interval. In this way, we use a sub-interval (of arbitrary width) inside the fixed-interval width, thereby restricting the possible values that the variable can assume. Both o_m and o_M must be smaller than w_i, greater than or equal to 0 and

$o_m \leq o_M$. Then, if i and (o_m, o_M) are associated to v_i, this means that the possible values of v_i belong to the interval

$$[(i \times w_i) + o_m..(i \times w_i) + o_M]$$

An element of our abstract domain is then stored as a *map* from hypercubes to tuples of offsets. In this way, we can keep the original definition of a hypercube as a tuple of integers, but we also map each hypercube to a tuple of offsets (one for each variable). Now an **abstract state** is defined by

$$M : \mathbb{Z}^{|Vars|} \to (\mathbb{R} \times \mathbb{R})^{|Vars|}$$

that is a map where the domain is the set of hypercubes, and the codomain is the set of tuples of offsets.

The new **partial order** has to check, other than the set inclusion between the two sets of hypercubes, also that the offsets of the presumed-smaller element are all included in the offsets of the bigger element. Formally:

$$M_1 \leq_{\mathcal{H}} M_2 \Leftrightarrow dom(M_1) \subseteq dom(M_2) \wedge \forall h \in dom(M_1) : M_1(h) \leq_O M_2(h)$$

where $\leq_O$ is the partial order between tuples of offsets and is defined as follows:

$$\rho^1 \leq_O \rho^2 \Leftrightarrow \forall i \in [0, |Vars| - 1] : [o_i^1, O_i^1] \subseteq [o_i^2, O_i^2]$$

where ρ^1, ρ^2 are two tuples of pairs of offsets defined such that $\rho_i^1 = (o_i^1, O_i^1) \forall i \in [0, |Vars| - 1]$ and $\rho_i^2 = (o_i^2, O_i^2) \forall i \in [0, |Vars| - 1]$. The partial order $\leq_O$ checks if the interval defined by each pair of offsets of the first tuple ρ^1 is included in the interval defined by the corresponding offset of the second tuple ρ^2.

The **greatest lower bound** between two abstract states ($M = M_1 \sqcup M_2$) is defined by

$$dom(M) = \{h : h \in (dom(M_1) \cap dom(M_2)) \wedge intersected(M_1(h), M_2(h))\}$$

and

$$\forall h \in dom(M) : M(h) = intersection(M_1(h), M_2(h))$$

where $intersected(\rho_1, \rho_2)$ is a boolean function which checks if all pairs of offsets from the tuples in input have an intersection:

$$intersected(\rho_1, \rho_2) = \texttt{true} \Leftrightarrow \forall i \in [0, |\mathsf{Vars}| - 1] : [o_i^1, O_i^1] \cap [o_i^2, O_i^2] \neq \emptyset$$

and $intersection(\rho_1, \rho_2)$ is the function which actually computes the tuple of offsets resulting from the intersection of offsets from ρ_1, ρ_2:

$$intersection(\rho_1, \rho_2) = \rho : \forall i \in [0, \mathsf{Vars}] - 1] : \rho_i = [o_i^1, O_i^1] \cap [o_i^2, O_i^2]$$

The **least upper bound** between two abstract states ($M = M_1 \sqcup M_2$) is defined by $dom(M) = dom(M_1) \cup dom(M_2)$, and

$$\forall h \in dom(M) : M(h) = \begin{cases} M_1(h) & \text{if } h \in dom(M_1) \wedge h \notin dom(M_2) \\ M_2(h) & \text{if } h \in dom(M_2) \wedge h \notin dom(M_1) \\ merge(M_1(h), M_2(h)) & \text{otherwise} \end{cases}$$

where $merge(o_1, o_2)$ creates a new tuple of offsets by merging the two tuples of offsets in input: for each pair of corresponding offsets (for example (m_1, M_1) and (m_2, M_2)), the new offset is the widest combination possible (i.e., $(\min(m_1, m_2)$ and $\max(M_1, M_2))$). Note that this definition corresponds to the pointwise application of the least upper bound operator over intervals.

The **widening** operator is extended in the same way: it applies the standard widening operators over intervals pointwisely to the elements of the vector representing the offsets.

As for the **abstraction and concretization functions**, $\alpha_{\mathcal{H}}$ and $\gamma_{\mathcal{H}}$ have been defined in a generic way with respect to the abstraction and concretization functions of the non-relational domain used to abstract the single variables. Then, we just have to explicit how to modify the abstraction and concretization function of intervals of real values by considering also an offset inside the interval. Both new definitions are very simple:

- for the concretization function, given a fixed-width interval $[a, b]$ and a pair of offsets (o, O), instead of returning the interval $[a, b]$, we return the sub-interval $[a + o, a + O]$;

- for the abstraction function, given a value $r \in \mathbb{R}$, suppose that $r \in [i \times w..(i + 1) \times w]$ (where w is the width associated to the variable we are abstracting). Then, we return also the offset pair $(r - i \times w, r - i \times w)$ together with the index i which represents the interval containing r.

5.6 Abstract Semantics

In this section, we are going to define the abstract versions of the operations presented in Section 5.3. For the most part, the abstract semantics applies existing semantic operators of boxed Intervals [51]. Here we sketch how these operators are used to define the semantics on $\mathcal{H}$. In particular, we will define:

- the abstract semantics $\mathbb{I}$ of arithmetic expressions, which receives an expression and a single hypercube in input and returns an *interval of real values* resulting from the execution of that expression when the variable values belong to that hypercube.

- the abstract semantics $\mathbb{B}$ of Boolean comparisons, which receives a single hypercube and two expressions in input and returns an *abstract value of the boolean domain* (namely, *true*, *false*, or $\top$) obtained by comparing the two expressions (through $\geq, >, \leq, <$ or $\neq$) when the variable values belong to that hypercube. Boolean conditions can be combined through logical operators (and, or, not) in the usual way (i.e., exploiting the abstract semantics of the boolean abstract domain).

- the abstract semantics $\mathbb{S}$ of statements, which receives in input a set of hypercubes (the current abstract state) and returns a new set of hypercubes (the new abstract state after the execution of the statement).

5.6.1 The Abstract Semantics of Arithmetic Expressions, $\mathbb{I}$

Note that the semantics $\mathbb{I}$ returns an interval of real values which width is *not* fixed. The restriction on the interval width will be enforced by the semantics of statements (i.e., by the variable assignment).

Constants

We define a constant as a variable which gets assigned only once with a constant value, or a numerical value which appears in some statements (without being assigned to a specific variable). To simplify the treatment of constants, we execute a preprocessing on the program with constant propagation, to remove constant variables and replace their uses with their numerical value.

The abstract semantics of an expression made up by a constant numeric value is, simply, an interval of zero width: the extremes of the interval are the same and they are equal to the value of the constant. Then, the abstract semantics of a constant is:

$$\mathbb{I}[\![c]\!]\ h = [c, c]$$

Note that the value of the hypercube in input (h) is ignored because it is not needed to compute the result.

Intervals

The abstract semantic of an expression made up by a random value inside an interval of real values is exactly that interval, without modifications. We ignore the hypercube passed in input.

$$\mathbb{I}[\![\texttt{rand}([m, M])]\!]\ h = [m, M]$$

Variables

When the expression is made up by a variable, we must consider the abstract value of that variable in the hypercube passed in input. Let h_i be the integer index associated to the i-th dimension of the hypercube, V_i the i-th variable defined in the program and w_i the width associated in the analysis to such variable. Then, the abstract semantics of a variable is:

$$\mathbb{I}[\![V_i]\!]\ h = [h_i \times w_i..(h_i + 1) \times w_i]$$

If we also using offsets and (o_i, O_i) is the pair of offsets associated to V_i, then the abstract semantics is:

$$\mathbb{I}[\![V_i]\!]\ h = [(h_i \times w_i) + o_i..(h_i \times w_i) + O_i]$$

Arithmetic Operations

We considered only the most used arithmetic operators. In particular, we considered sum ($+$), subtraction ($-$), product ($\times$) and division ($\div$). These operators should suffice for most physics simulations (for example, our case study requires only sum and product - the change of sign being a multiplication for -1). Anyway, our framework can be easily extended to support other operations (for example modulus), by simply defining their abstract semantics when working on intervals of values.

The expression $E := E_1$ `aop` E_2 returns the interval obtained using the interval arithmetic defined in Section 2.2, applied on the two intervals resulting from the execution of the abstract semantics on the expressions E_1, E_2.

5.6.2 The Abstract Semantics of Boolean Conditions, $\mathbb{B}$

We use the semantics $\mathbb{I}$ of arithmetic expressions to define the abstract semantics $\mathbb{B}$ of Boolean comparisons. This semantics returns an abstract value of the boolean domain (*true*, *false*, or $\top$). As it happened with $\mathbb{I}$, also in this case we work with a single hypercube (it will be the semantics of statements to deal with sets of hypercubes).

Given a hypercube h and a Boolean comparison E_1 `bop` E_2 (where `bop` $\in \{\geq, >, \leq, <, \neq\}$), $\mathbb{B}$ returns the boolean abstract value obtained by comparing the intervals returned by the execution of the semantics $\mathbb{I}$ on E_1 and E_2. For example, if $\mathbb{I}[\![E_1]\!]h$ returns the interval $i_1 = [a, b]$ and $\mathbb{I}[\![E_2]\!]h$ returns the interval $i_2 = [c, d]$, then we have to compare the intervals i_1, i_2. The abstract comparison between them depends on the specific comparison operator present in the statement:

- $i_1 \neq i_2$ returns true if $b < c \vee a > d$, false if $b = c = a = d$, and $\top$ otherwise.
- $i_1 < i_2$ returns true if $b < c$, false if $a > d$, top otherwise.
- $i_1 > i_2$ returns true if $a > d$, false if $b < c$, top otherwise.

- $i_1 \leq i_2$ returns true if $b \leq c$, false if $a \geq d$, top otherwise.
- $i_1 \geq i_2$ returns true if $a \geq d$, false if $b \leq c$, top otherwise.

$\mathbb{B}$ will be used by the statement semantics $\mathbb{S}$ when computing the semantics of `if` and `while` statements to discard the hypercubes that surely do not satisfy the condition. Note that in this way we lose some precision. For instance, imagine that in a given hypercube we know that $\texttt{x} \in [0..5]$ (because the fixed width of intervals associated to `x` is 5), and we check if $\texttt{x} \leq 3$ when computing the semantics of an `if` statements. The answer of $\mathbb{B}$ will be $\top$, and so this hypercube will be used to compute the semantics of both the branches. Indeed, we would know that $\texttt{x} \in [0..3]$ in the `then` branch, and $\texttt{x} \in [4..5]$ in the else branch. Nevertheless, we cannot track this information in our hypercube, since the width of the interval associated to `x` is 5. Anyway, we will present (Section 5.7.3) how we can recursively modify the widths of the analysis to improve precision in these cases.

5.6.3 The Abstract Semantics of Statements, $\mathbb{S}$

Assignment

Usually, with non-disjunctive domains, the abstract semantics of an assignment is straightforward: you have to compute the abstract semantics of the expression and update the abstract value of the variable with the result. In our domain, though, we track a different kind of information: we represent possible values of all variables together (through hypercubes) and we consider disjunctive information (a set of valid hypercubes instead of a single hypercube). Therefore, we must devise a specific abstract semantics to deal with the assignment statement.

Let the assignment be $V_i = e$, where e is an arithmetic expression. Our approach can then be sketched as follows:

- we consider, separately, each hypercube h of the current state;
- we compute the abstract semantics of the arithmetic expression e passing to it the hypercube h ($\mathbb{I}[\![e]\!]h$).
- we create a new hypercube (or *some* new hypercubes, depending on the width of the resulting interval), where the abstract value of V_i is the abstraction of the interval resulting from e.

This approach does not necessarily produce a single hypercube, since the interval to assign could have a greater width than the fixed width of the assigned variable (for example, the interval $[0..6]$ when $w = 5$). It could also happen that the resulting interval width is smaller than the fixed width, but the interval spans over more than one hypercube side, due to the fixed space partitioning (for example, the interval $[3..6]$ when $w = 5$, because the space is partitioned in $[0..5], [5..10]$, etc.). In these

cases, we build up several hypercubes that cover the resulting interval. Then, the cardinality of $\mathcal{H}$ can increase after a statement execution, because each hypercube could produce many new hypercubes. On the other hand, if many hypercubes of the initial state map to the same hypercube in the resulting state, it could also happen that the cardinality of $\mathcal{H}$ decreases (or remains unmodified) after the execution of an assignment.

Formally:

$$\mathbb{S}[\![\mathtt{V_i} = \mathtt{e}]\!]\ H = \bigcup_{h \in H} assign(h, V_i, \mathbb{I}[\![e]\!]\ h)$$

where h is a hypercube, V_i is the assigned variable, and the *assign* function is defined as:

$$assign(h, V_i, [a, b]) = \{h[i \mapsto m] : [m \times w_i..(m+1) \times w_i] \cap [a..b] \neq \emptyset\}$$

where $[a..b]$ is the interval we are assigning (which depends on the hypercube h, since we use its variables values to compute the result of the expression). The output of this function is the set of hypercubes covering all the intervals that overlap with the interval assigned to the given variable.

We repeat this process for each hypercube h in the abstract state by using it as input for the computation of *assign*. In this way, we are able to over-approximate the assignment while also keeping the fixed widths of the intervals, which are very important for performance issues.

Assignment and Offsets

Offsets allow us to recover some precision when computing the abstract semantics of assignment. In particular, as the expression semantics $\mathbb{I}$ returns intervals of arbitrary widths, we can use such exact result to update the offsets of the abstract state.

Formally, the semantics of the assignment considering offsets is modified as follows:

$$assign(h, V_i, [a..b]) = \{h[i \mapsto (m, o_m, o_M)] : [m \times w_i..(m+1) \times w_i] \cap [a..b] \neq \emptyset\}$$

where h is a hypercube, V_i is the assigned variable, $[a..b]$ is the interval we are assigning and o_m, o_M are computed as:

$$o_m = \begin{cases} 0 & \text{if } a \leq (m \times w_i) \\ a - (m \times w_i) & \text{otherwise} \end{cases}$$

$$o_M = \begin{cases} w_i & \text{if } b \geq ((m+1) \times w_i) \\ b - (m \times w_i) & \text{otherwise} \end{cases}$$

Consider the evaluation of statement $\texttt{x} = \texttt{x} + \texttt{0.01}$ inside a while loop with 1.0 as width of `x` and [0..1] as initial value of `x`. After the first iteration, the abstract semantics computes [0.0..1.0] and [1.0..2.0] with offsets [0.01..1.0] and [1.0..1.01], respectively. In this way, at the following iteration we would obtain again the same two intervals with the offsets changed to [0.02..1.0] and [1.0..1.02]. This results is strictly more precise than the one obtained without offsets, and it is an essential feature of our abstract domain. For instance, in the case study of Figure 5.1 offsets will allow us to discover if a bouncing ball exits the screen after N iterations of the `while` loop.

If-then-else

To precisely deal with branches of `if` statements, we partition the abstract state $\mathcal{H}$ with respect to the evaluation of the branching condition. In particular, we compute the abstract semantics $\mathbb{B}$ of the boolean condition on each hypercube of $\mathcal{H}$ and we assign each hypercube to a specific partition, based on the result of the condition semantics. Therefore, we obtain three partitions:

- the hypercubes for which the condition evaluates to `true` (p_t);
- the ones for which the condition evaluates to `false` (p_f);
- the ones for which we do not have a definitive answer ($p_\top$).

Once obtained these three partitions, we can compute selectively the abstract semantics of the two branches, and in particular the `then` branch with $p_t \cup p_\top$, and the `else` branch with $p_f \cup p_\top$.

Formally, the semantics of the `if` statements follows:

$$\mathbb{S}[\![\texttt{if}(B)\ \texttt{then}\ P_1\ \texttt{else}\ P_2]\!]\ H = (\mathbb{S}[\![P_1]\!]\ (p_t \cup p_\top)) \cup (\mathbb{S}[\![P_2]\!]\ (p_f \cup p_\top))$$

where

$$\begin{aligned} p_t &= \{h \in H : \mathbb{B}[\![B]\!]\ h = \texttt{true}\} \\ p_f &= \{h \in H : \mathbb{B}[\![B]\!]\ h = \texttt{false}\} \\ p_\top &= \{h \in H : \mathbb{B}[\![B]\!]\ h = \top\} \end{aligned}$$

Concatenation of Statements

The abstract semantics of the concatenation of two statements is straightforward: it executes the abstract semantics of the first statement, it takes the result and it passes it as input to the abstract semantics of the second statement. Formally:

$$\mathbb{S}[\![P_1; P_2]\!]\ H = \mathbb{S}[\![P_2]\!]\ (\mathbb{S}[\![P_1]\!]\ H)$$

While Loop

The semantics of the while loop $\texttt{while}(B)P$ exploits the standard abstract semantics for loops, with one peculiarity: at each abstract execution of the loop iteration, the semantics of the loop body is applied only on a restricted subset of hypercubes, i.e. the ones for which the boolean condition of the loop evaluates to `true` or $\top$.

Formally, let $\widehat{S} = \mathbb{S}[\![P]\!]$ be the abstract semantics of the loop body P and let H_0 be the pre-loop state (i.e., a set of hypercubes). Then, the abstract iterations to compute the semantics of the loop are defined as follows:

$$\begin{aligned} H_1 &= H_0 \cup (\widehat{S}\ H_0^T) \\ H_2 &= H_1 \cup (\widehat{S}\ H_1^T) \\ &\dots \\ H_n &= H_{n-1} \cup (\widehat{S}\ H_{n-1}^T) \end{aligned}$$

where $H_i^T = \{h \in H_i : \mathbb{B}[\![B]\!]\ h = (\texttt{true} \vee \top)\}$, that is, H_i^T is the subset of hypercubes of H_i for which the abstract evaluation of the loop condition B returns `true` or $\top$. We exclude from the computation of the loop body semantics the hypercubes which certainly do not satisfy the loop condition B.

The abstract semantics of the loop is then obtained as the fixpoint of the iterations defined above: we continue computing such iterations until $H_n = H_{n-1}$ for some $n \in \mathbb{N}$ or until we reach a maximum number $n^{\#}$ of iterations (after which we employ the widening to ensure convergence in a finite number of steps).

5.7 Tuning the Analysis

In this section we are going to talk about some practical issues regarding the analysis using the Hypercubes domain. In particular, we are going to explain: (i) how to initialize the hypercubes set at the beginning of the analysis, (ii) how to keep track of the source of each hypercube (to give more interesting information other than the boolean answer to the verification of the property), and (iii) how to select the interval widths in the hypercubes.

5.7.1 Initialization

Before starting the analysis we have to determine the number of sides each hypercube will have. To do this, we must find all the variables ($Vars$) of the program which are not constants (i.e., assigned only once at the beginning of the program).

We require the program to initialize all the variables at the beginning of the program. Note that physics simulations, like our case study, satisfy this requirement because they are made up by an initialization of all variables, followed by a `while` loop which contains the core of the program (i.e., the update of the simulated world).

Otherwise, we can consider a dummy initialization (i.e., 0.0) for all variables which are not initialized at the beginning of the program. The actual initialization of the variables will be treated as a normal assignment, without any loss of precision.

The initialization of the analysis is then made in two steps:

1. For each initialized variable, we compute its abstraction in the non-relational domain chosen to represent the single variables. The resulting set of abstract values could contain more than one element. Let us call $\alpha(V)$ the set of abstract values associated to the initialization of the variable $V \in Vars$.

2. We compute the Cartesian product of all sets of abstracted values (one for each variable). The resulting set of tuples (where each tuple has the same cardinality as $Vars$) is the initial set of hypercubes of the analysis.

Formally:

$$\mathcal{H} = \bigtimes_{V \in Vars} \alpha(V)$$

As an example, consider the code of our case study in Figure 5.1. First of all, we must identify the variables which are not constants: dt, g, k are assigned only during the initialization, so we do not include them in $Vars$. The set of not-constant variables is then $Vars = \{V_1 = px, V_2 = py, V_3 = vx, V_4 = vy\}$, and so $|Vars| = 4$. Suppose that the widths associated to the variables are $w_1 = 10.0, w_2 = 25.0, w_3 = 30.0, w_4 = 5.0$. Then, the abstraction of each variable is $\alpha(V_1) = \{0\}$, $\alpha(V_2) = \{0, 1\}$, $\alpha(V_3) = \{0, 1\}$, and $\alpha(V_4) = \{-6\}$. The Cartesian product of these abstractions brings us to the following initial set of hypercubes:

$$\mathcal{H} = \{h_1 = (0, 0, 0, -6), h_2 = (0, 0, 1, -6), h_3 = (0, 1, 0, -6), h_4 = (0, 1, 1, -6)\}$$

If we use offsets, we also associate each hypercube to a tuple of pairs of offsets (one for each variable). For example, the hypercube h_1 would be associated to the offsets tuple $O = \langle(0.0, 10.0), (0.0, 25.0), (0.0, 30.0), (0.0, 5.0)\rangle$.

5.7.2 Tracking the Origins

During the analysis of a program we also track, for each hypercube of the current abstract state, the initial hypercubes (*origins*) from which it is derived. To store such information, we proceed as follows.

Let H_i be the set of hypercubes obtained for the i-th statement of the program. The data structure of a hypercube h contains also an additional set of hypercubes, h^{or}, which are its origins and are always a subset of the initial set of hypercubes, i.e., $\forall h : h^{or} \subseteq H_0$.

At the first iteration, each hypercube contains only itself in its origins set:

$$\forall h \in H_0 : h^{or} = \{h\}$$

When we execute a statement of the program, each hypercube produces some new hypercubes: at this stage, the origins set is simply propagated. For example, if h generates h_1, h_2, then $h_1^{or} = h_2^{or} = h^{or}$. When merging all the newly produced hypercubes in a single set (the abstract state associated to the point of the program just after the executed statement), we also merge through set union the sets of origins of any repeated hypercube.

For example, consider $H_i = \{h_a, h_b\}$ and let h_1, h_2 be the hypercubes produced by h_a executing statement i-th and h_2, h_3 be those produced by h_b. Then, $H_{i+1} = \{h_1, h_2, h_3\}$ and $h_1^{or} = h_a^{or}$, $h_2^{or} = h_a^{or} \cup h_b^{or}$ and $h_3^{or} = h_b^{or}$.

5.7.3 Width Choice

The choice of the interval widths is quite important, because it influences both the precision and efficiency of the analysis. The widths influence the granularity of the space partitioning with respect to each variable. On the one hand, if we use smaller widths we certainly obtain more precision, but the analysis risks being too slow. On the other hand, with bigger widths the analysis will be surely faster, but we could not be able to verify the desired property.

The width selection can be automatized. We implemented a recursive algorithm which adjusts the widths automatically, starting with bigger ones and then decreasing them only in the portions of space where it is really needed, to avoid compromising the performance.

We start with wide intervals (i.e., coarse precision, but fast results) and we run the analysis for the first time. We track, for each hypercube of the final abstract state, the initial hypercubes (*origins*) from which it is derived (as explained in the previous section).

At the end of the analysis, we check, for each hypercube of the final set, if the desired property is verified. We associate to each origin its final result by merging the results of its derived final hypercubes: some origins will certainly verify the property (i.e., they produce only final hypercubes which satisfy the property), some will not, and some will not be able to give us a definite answer (because they produce both hypercubes which verify the property and hypercubes which do not verify it). We can partition the starting hypercubes set with respect to this criterion (obtaining, respectively, the *yes* set, the *no* set and the *maybe* set).

We run the analysis again, but *only* on the origins which did not give a definite answer (the *maybe* set). To obtain more precise results in this specific space portion, the analysis is now run with halved widths. Note that this step is only performed until we reach a specific threshold, i.e., the *minimum width* allowed for the analysis. This parameter can be specified by the user (together with the starting widths). The smaller this threshold is, the more precise (but slower) the analysis becomes.

At the end of this recursive process, we obtain three final partitions of the variable space: a set of starting hypercubes which certainly verify the property (*yes* set), a set of starting hypercubes which certainly do not verify the property (*no* set), and

a set of starting hypercubes which, at the minimum width allowed for the analysis, still do not give a definite answer (*maybe* set). The analysis is then able to tell us which initial values of the variables permit to verify the property (the union of all the *yes* sets encountered during the recursive algorithm) and which do not. Thanks to these results, the user can modify the initial values of the program, and run the analysis again, until the answer is that the property is verified *for all initial values*. In our case study, for example, we can adjust the possible initial positions and velocities until we are sure that the ball will exit the screen in a certain time frame.

The formalization of this recursive algorithm is presented in Algorithm 5.

Algorithm 5 The width adjusting recursive algorithm

function ANALYSIS($currWidth, minWidth, startingHypercubes$)
  return $(yes \cup yes', no \cup no', maybe')$
  where
  $(yes, no, maybe) = hypercubesAnalysis(currWidth, startingHypercubes)$
  if $currWidth/2.0 \geq minWidth$ **then**
    $(yes', no', maybe') = Analysis(currWidth/2.0, minWidth, maybe)$
  else
    $(yes', no', maybe') = (Set.empty, Set.empty, maybe)$
  end if
end function

The overall analysis takes as input the starting width, the minimum width allowed and the set of starting hypercubes (obtained from the initialization of the program as described in Section 5.7.1). It executes the analysis on such data with the function `hypercubesAnalysis`, which returns three sets of hypercubes (`yes`, `no`, `maybe`) with the meaning explained above. Then, if the halved width is still greater than the minimum one allowed, the algorithm performs a recursive step by repeating the analysis function only on the `maybe` hypercubes set (with halved width).

Note that the three final hypercubes sets (the *yes,no,maybe* partitions) will contain hypercubes of different sizes: this happens because each hypercube can come from a different iteration of the analysis, and each iteration is associated to a specific hypercube size. A certain portion of the variable space could give a definite answer even at coarse precision (for example, when the horizontal velocity of the ball is sufficiently high, the values of other variables do not matter so much), while another portion could need to be split in much smaller hypercubes to give interesting results.

5.8 Experimental Results

In this section we present some experimental results on the case study introduced in Section 5.2. We want to check if *Property 1* is verified on the program of Figure 5.1 and, in particular, we want to know which subset of starting values brings to verify it. We implemented our analysis in the F# language with Visual Studio 2012. We ran the analysis on an Intel Core i5 CPU 1.60 GHz with 4 GB of RAM, running Windows 8 and the F# runtime 4.0 under .NET 4.0.

5.8.1 Setting Up

We set the initial widths associated to all variables to 100.0 and the minimum width allowed to 5.0. As for *Property 1*, we set $T = 5$, i.e., we want to verify if the ball is surely out of the screen within 5 seconds from its generation. Since $dt = 0.05$, a simulation during 5 seconds corresponds to $5/0.05 = 100$ iterations of the `while` loop. To verify this property, we apply trace partitioning [118] to track one abstract state per loop iteration until the 100-th iteration (we do not need to track precise information after the 100th iteration). The position which corresponds to the exiting from the screen is 100.0: if after 100 iterations the position `px` is surely greater than 100.0, then *Property 1* is verified. The whole of these values (starting variables values and widths, minimum width allowed, number of iterations, position to reach) make up our *standard workbench data*. We will experiment to study how efficiency and precision change when modifying some parameters of the analysis, and for each test we will specify only the values which are different with respect to the standard workbench data. We will start by running the analysis on the standard workbench data, and then we will experiment by changing other values, one at a time, to study how the efficiency and precision change. In particular, we will start by showing how precision and performance are affected by changing the minimum width allowed. After that, we will concentrate on the horizontal velocity variable (`vx`) and we will show how we can use our analysis as a form of "assisted debugging" to understand which starting values of a variable bring to verify the property and which not.

For each test, the analysis returns three sets of starting hypercubes:

- the initial values of the variables which satisfy the property (*yes* set);
- the initial values of the variables which surely do not satisfy the property (*no* set);
- the initial values of the variables which may or not satisfy the property (*maybe* set).

To make the results more immediate and clearer, we computed for each *yes* and *no* set the corresponding volume covered in the space by their hypercubes. We also consider the *total* volume of the variable space, i.e., the volume covered by

Table 5.2: Varying the minimum width allowed (MWA)

MWA	Time (sec.)	*yes*+*no* volume	Precision
3	530	131934	88%
5	77	99219	66%
12	11	40625	27%
24	1	25000	17%
45	0.2	0	0%

all possible values with which the program variables are initialized. In the case of the standard workbench data, the *total* volume is $10.0 \times 50.0 \times 60.0 \times 5.0 = 150000$. Dividing the sum of *yes* and *no* volumes by the *total* volume, we obtain the percentage of the cases for which the analysis gives a definite answer. We will call this percentage the *precision* of the analysis.

Note that the *total* volume refers to the initial variable space, i.e., to the space defined by the intervals assigned to the variables at the beginning of the program. The *yes*, *maybe*, and *no* volumes refer to the same space, since they are defined in terms of *starting* hypercubes. These volumes are all finite, since we suppose that all the variables are initialized by a bound interval, and this is always the case in Computer Games Software.

5.8.2 Varying the Minimum Width Allowed

First of all, we run the analysis modifying the *minimum width allowed* (MWA) parameter, to see how the precision and performance are affected. In Table 5.2 and Figure 5.6 we report the results of these tests. In Figure 5.6 the horizontal axis represents the value of the MWA parameter. On the vertical axis we reported, respectively, the precision of the analysis (as defined in the previous paragraph) and the execution time.

From these results, we can clearly see the trade-off between performance and precision: the performance decreases when we set small widths, and it is instead very good on bigger ones. On the other hand, by decreasing the MWA we also gain more precision. For instance, when MWA = 45 we do not have any certain answer, while with MWA = 3 the certain answers cover the 88% of the volume space, a quite precise result.

5.8.3 Finding Appropriate Starting Values

In Table 5.3 we reported the results of a series of successive tests obtained by changing the horizontal velocity of the ball (`vx`). In particular, we made up a series of tests simulating the behaviour of a developer using our analysis to debug his code.

- Let us suppose that we wrongly inserted a starting interval of negative values

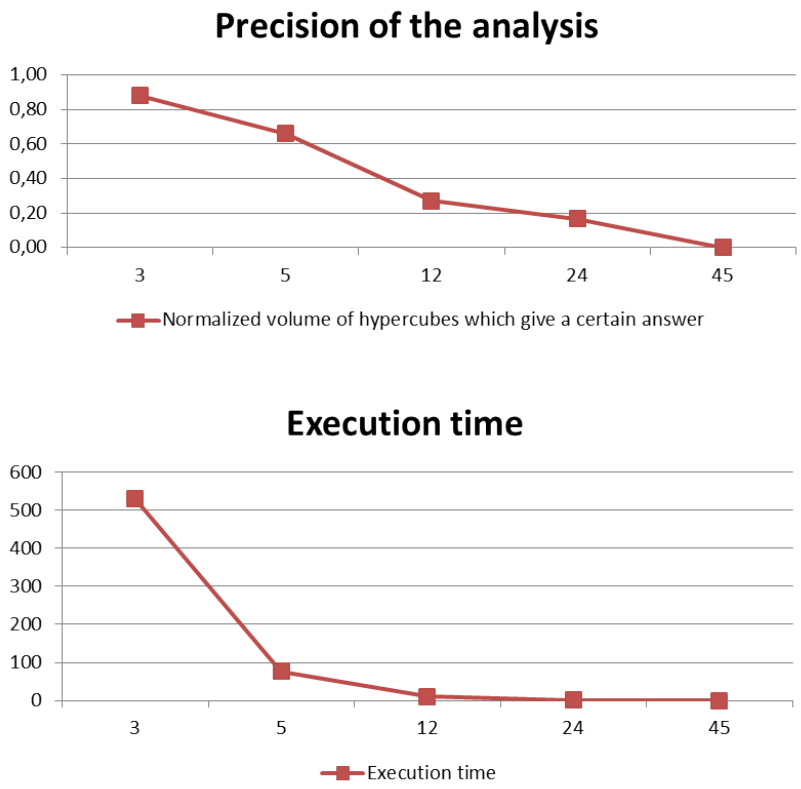


Figure 5.6: Varying the minimum width allowed - Plots

(between -120 and 0) for the horizontal velocity variable `vx`. The first test (# 1) shows us that the program does not work correctly, since the *no* volume is 100%. Also, to give this answer, the analysis is very quick because a low MWA (45) suffices.

- After that, we try (test # 2) with very high positive velocities (between 60 and 120) and we obtain (also very quickly) a 100% of positive answer: we then know for sure that with these velocities the program works correctly.

- Now it remains to verify what happens with velocities between 0 and 60, and we try this in test # 3, where we decrease the MWA because we need more precision (the results with greater MWA were presented in Section 5.8.2). Some values of `vx` (i.e., ≥ 31.25) ensure that the property is verified, some other values (i.e., ≤ 12.5) ensure that the property is not verified, but the ones in between are uncertain.

- To do a double check about this data, we execute also tests # 4 and # 5, where we keep, respectively, only the low (between 0 and 15) and the high (between 30 and 60) values: in both cases the analysis is fairly quick in confirming the 100% *no* and 100% *yes*.

- So we try with a smaller MWA (3) in test # 6 on the interval [15..30]: about a quarter of the starting values produces *yes* and another quarter produces *no*. The *no* derives from low values (smaller than 18) and we confirm this in test # 7. As for medium-high values, test # 6 shows that, with a velocity greater than 25, the answer is *almost* always *yes*. It is not always yes because, with this range of velocities, the values of other variables become important to verify the property.

- Test # 8, in fact, shows us that velocities within 25 and 30 produce an 82% of *yes*, but a 18% of *maybe* remains.

- Finally, in test # 9 we modify also other two variables with values chosen looking at the results from test # 6 and # 8: in particular, we set the horizontal position (`px`) between 5 and 10, and the vertical position (`py`) between 40 and 50. With such values, the answers are 100% *yes*.

After these tests, the developer of the case study is sure that, with horizontal velocities greater than 30, the program certainly satisfies the property chosen to verify). For values between 25 and 30, other variable values must be changed (`px` and `py`) to make the program work correctly. Making some other tests, we could also explore what happens with values between 18 and 25. However, with values below 18 the program will certainly not satisfy the property.

Table 5.3: Varying the horizontal velocity (vx)

Test	vx interval	MWA	Time (sec)	Answer	Comment
# 1	[-120 .. 0]	45	1	$no = 100\%$	With negative values the answer is always no.
# 2	[60 .. 120]	45	0.2	$yes = 100\%$	With very high positive values the answer is always yes.
# 3	[0 .. 60]	5	77	$yes = 45\%$ $no = 21\%$	Uncertainty. High values ($\geq$ 31.25) imply *yes*, low values ($\leq$ 12.5) imply *no*.
# 4	[0 .. 15]	24	0.5	$no = 100\%$	Double check on low values: answer always no.
# 5	[30 .. 60]	5	30	$yes = 100\%$	Double check on medium-high values: answer always yes.
# 6	[15 .. 30]	3	526	$yes = 27\%$ $no = 25\%$	Uncertainty. Low values ($\leq$ 18) imply *no*, for high values ($\geq$ 25) depends also on other variables.
# 7	[15 .. 18]	5	7	$no = 100\%$	Double check on medium-low values: answer always no.
# 8	[25 .. 30]	3	164	$yes = 82\%$ $maybe = 18\%$	Double check on medium-high values: answer almost always yes. In this case, also values of other variables influence the result.
# 9	[25 .. 30]	5	1	$yes = 100\%$	Modified also py ([40 .. 50]) and px ([5 .. 10]). Answer always yes.

5.8.4 Varying Other Parameters

In Section 5.8.2 we showed how the precision and performance of the analysis change when modifying the MWA parameter. Here, we do similar experiments for other variables of the program: we will change their initial values (one variable at a time) and we will show how this gives us some important clues about the behaviour of the simulation. In this case, in fact, we focus only on the volume of the *yes* set (instead of the precision, i.e. the whole of *yes* and *no*): we want to understand how changing the values of some parameters influences the satisfaction of the property to verify. For each battery of tests, we will show a plot with the proportion of positive answers (value in $[0, 1]$, where 0 means that the property is never satisfied and 1 means that the property is satisfied in every execution of the program because the *yes* volume corresponds to the *total* volume) and a plot with the execution time.

Position to Reach

In Figure 5.7 we report the results obtained by running the analysis with different values for the position that the ball has to reach to exit the screen. When using value 100 (the one from the standard workbench data), we obtain approximately a 45% of success rate, i.e., half of the starting values volume results in the verification of the property. Intuitively, this corresponds to say that roughly half of the balls thrown by a continued execution of program will satisfy the property. However, if we increase the value of this parameter, we see that the success rate decreases rapidly, and with very high values (from 225 onward) the success rate is zero (i.e., no ball will ever be able to exit the screen within the desired time).

Horizontal Velocity, `vx`

In Figure 5.8 we report the results obtained by running the analysis with different starting intervals for the horizontal velocity of the ball, `vx`. This is very similar to what we did in Section 5.8.3 and in fact it confirms our previous findings: intervals covering only negative values never satisfy the property (the ball goes in the wrong direction!), while intervals covering high values (i.e., $[40, 100]$) always satisfy the property.

Vertical Position, `py`

In Figure 5.9 we report the results obtained by running the analysis with different starting intervals for the vertical position of the ball, `py`. We can see that this variable does not influence significantly the verification of the property: the success rate is 45% when $\texttt{py} \in [0, 50]$, while it is slightly less than 60% when the interval is higher ($[50, 100]$ or $[100, 150]$).

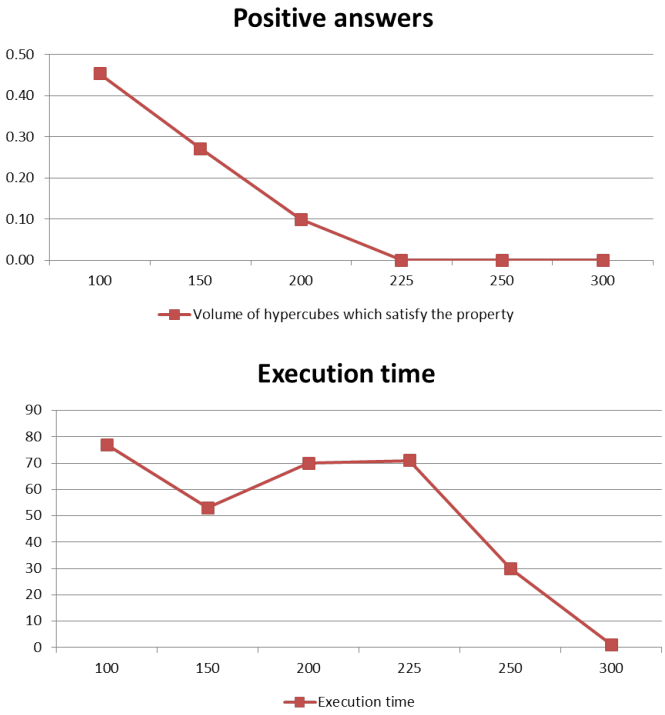


Figure 5.7: Varying the position to reach to exit the screen

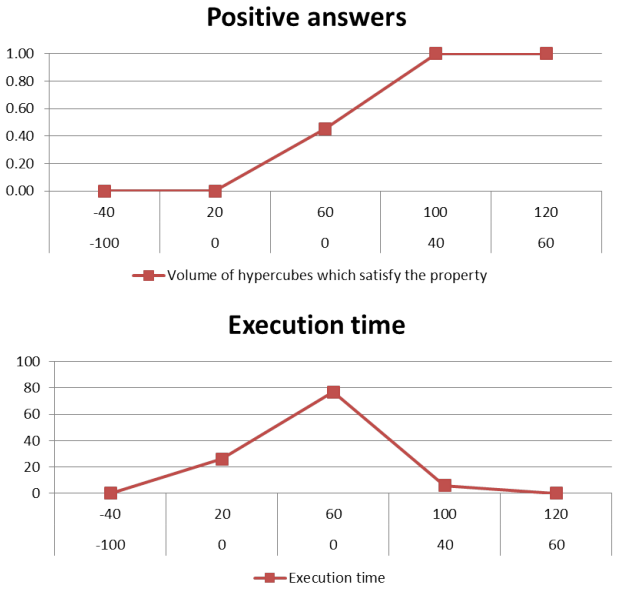


Figure 5.8: Varying the starting horizontal velocity

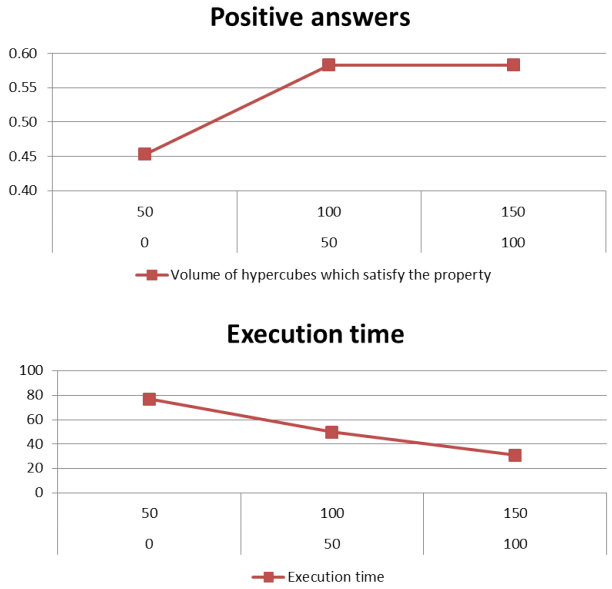


Figure 5.9: Varying the starting vertical position

Vertical Velocity, vy

In Figure 5.10 we report the results obtained by running the analysis with different starting intervals for the vertical velocity of the ball, `vy`. In this case, too, this variable does not seem to hold great significance regarding the property of interest: the success rate does not change much when varying the starting interval of `vy`, and, especially, there is not a recognizable pattern in this change.

Horizontal Position, px

In Figure 5.11 we report the results obtained by running the analysis with different starting intervals for the horizontal position of the ball, `px`. Here the pattern is clear: the higher the starting horizontal position (i.e., the more to the right of the screen is the starting point), the higher the success rate. This is very reasonable and it is the dual of modifying the position to reach: both parameters influence the horizontal distance of the ball from its starting point to its objective.

5.8.5 Discussion

In these scenarios, we ran the analysis by manually changing the initial values of program variables. In Figure 5.12 you can see the GUI of the analysis tool that we implemented and used to obtain the results presented in the previous sections. From such an interface, the user can set up the starting intervals of all four vari-

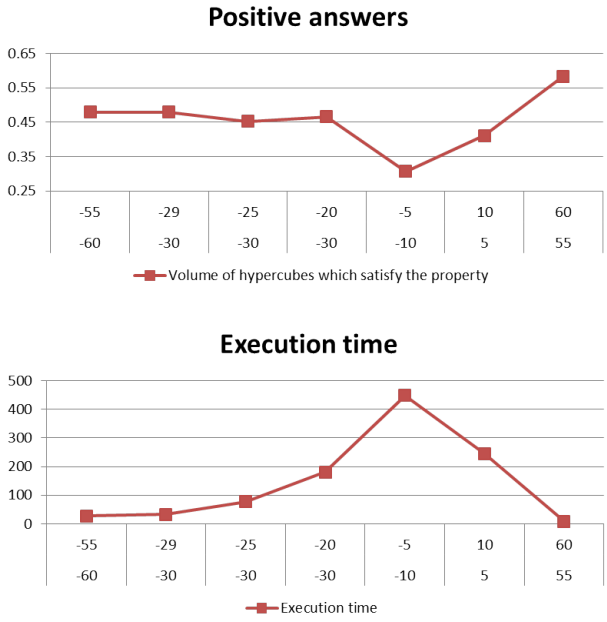


Figure 5.10: Varying the starting vertical velocity

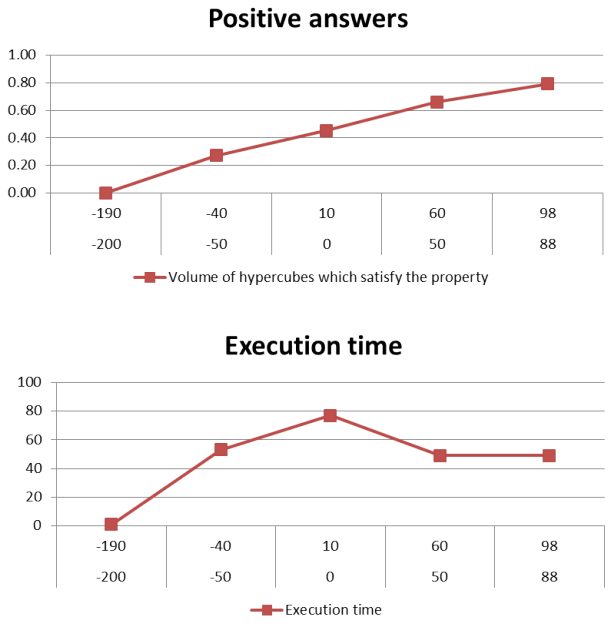


Figure 5.11: Varying the starting horizontal position

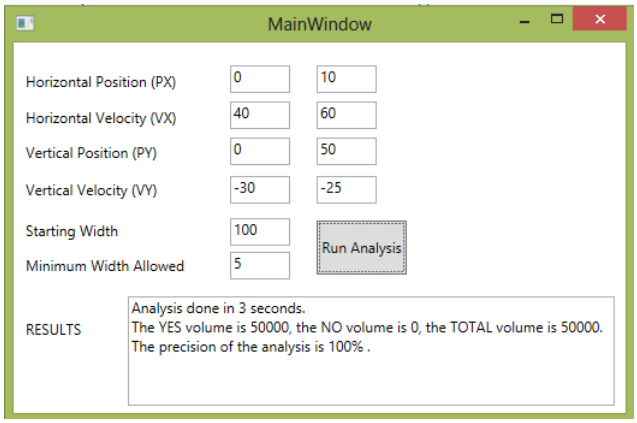


Figure 5.12: Analysis tool

ables (px, vx, py, vy), as well as the starting widths of the hypercubes sides and the minimum width allowed. In the results pane, the tool writes the execution time of the analysis, together with the volumes of the *yes* and *no* sets. It also computes the precision of the analysis in percentage as explained before ($\frac{yes+no}{total}$).

Notice that this process could be automatized. This process can be highly interactive, since the tool could show to the user even partial results while it is automatically improving the precision by adopting narrower intervals on the *maybe* portion as described by Algorithm 5. In this way, the user could iterate the process until it finds suitable initial values.

The execution times obtained so far underline that the analysis is efficient enough to be the basis of practical tools. Moreover, the analysis could be parallelized by running in parallel the computation of the semantics for each initial hypercube: exploiting several cores or even running the analysis in the cloud, we could further improve the efficiency of the overall analysis.

5.8.6 Extending the Case Study from 2D to 3D

The bouncing ball we used as case study featured a two-dimensional motion (horizontal and vertical). However, the most part of modern video games are set in three dimensions. For example, think about sport games: in many of them we can find a ball which bounces (tennis, volleyball, basketball, soccer, etc.). To consider these applications we must extend the 2D bouncing ball to 3D, by adding a third positional variable (pz) together with a velocity in such direction (vz). The code of the 3D motion of the bouncing ball is very similar to the 2D one (see Figure 5.13), because the depth component (z) of the motion behaves like the horizontal one (x). Note that we put dots instead of an interval in the assignment of some variables (py, vx, vz) because we will vary them in the following tests.

Since our approach is generic with respect to the number of program variables,

```
1  px = rand([0.0, 10.0]), py = rand(...), pz = rand([0.0,10.0])
2  vx = rand(...), vy = rand([-30.0, -25.0]), vz = rand(...)
3  dt = 0.05, g = -9.8, k = 0.8
4
5  while (true) do
6     if( py >= 0.0 ) then
7       (px, py, pz) = (px + vx * dt, py + vy * dt, pz + vz * dt)
8       (vx, vy, vz) = (vx, vy + g * dt, vz)
9     else
10      (px, py, pz) = (px + vx * dt, 0.0, pz + vz * dt)
11      (vx, vy, vz) = (vx, -vy, vz) * k
```

Figure 5.13: The bouncing ball case study extended in three dimensions

it is immediate to apply it to this new case study. The hypercubes will now be part of a six-dimensional variable space, because we need to track the values of six variables ($\mathtt{px}, \mathtt{py}, \mathtt{pz}, \mathtt{vx}, \mathtt{vy}, \mathtt{vz}$). Obviously, increasing the variable space increases also the complexity of the analysis. Note that the property to verify changes: the ball can now exit the screen also because it is too far from the observer (i.e., its depth is very high). The purpose of the analysis is then to verify if the ball exceeds a certain position in `x` *or* in `z` within a specific time frame (the same one of the 2D case study, 5 seconds).

In Table 5.4 we report the results of some tests executed on the 3D case study. We experiment by modifying the values of some parameters, while keeping some others fixed. In particular, the starting values of $\mathtt{px}, \mathtt{pz}, \mathtt{vy}$ are always included in the intervals $[0, 10], [0, 10], [-30, -25]$, respectively. For each test, we report the starting values of $\mathtt{py}, \mathtt{vx}, \mathtt{vz}$, the minimum width allowed (*MWA*), the *position to reach* (both in `x` and `z`) to exit the screen (for the verification of the property), the *execution time* in seconds, the *yes+no* and *total* volumes, as well as the derived value of *precision* in percentage (computed as described before, i.e. $\frac{yes+no}{total} \times 100$).

Some general considerations about the tests results follow:

- the variable space to explore is consistently bigger than in the 2D case study, where the total volume of the standard workbench data was 150000. Here, the smallest total volume is 500000 but we consider also much bigger volumes (like the 30 millions of test # 1, even though in such case the performance suffers considerably).

- the precision of the approach is very high. In most tests, we are able to classify 100% of the starting values volume between *yes* e *no* (i.e., between the starting values which certainly make the program satisfy - or not - the property).

- the analysis can be very fast in some cases (i.e., tests # 2,3,4) .

Table 5.4: Results of the analysis of the 3D bouncing ball case study

# test	py	vx	vz	MWA	Position x to reach	Position z to reach	Time (sec.)	*yes+no* volume	*total* volume	Precision
1	[0,50]	[0,60]	[30,50]	5	100	100	2153	30000000	30000000	100.0%
2	[40,50]	[0,10]	[40,50]	5	100	100	0.5	500000	500000	100.0%
3	[40,50]	[50,60]	[40,50]	5	100	100	0.4	500000	500000	100.0%
4	[40,50]	[0,10]	[0,10]	5	100	100	0.5	500000	500000	100.0%
5	[40,50]	[0,30]	[25,55]	5	100	100	270	3562500	4500000	79.2%
6	[10,20]	[0,30]	[25,55]	5	100	100	285	3562500	4500000	79.2%
7	[40,50]	[20,30]	[25,35]	5	100	100	65	187500	500000	37.5%
8	[40,50]	[20,30]	[25,35]	3	100	100	3613	499999.938	500000	100.0%
9	[40,50]	[50,60]	[40,50]	5	200	200	112	499999.969	500000	100.0%

- if the precision of the result is not satisfactory, we can improve it by using a smaller MWA, as clearly demonstrated by tests # 7 and 8. The precision of test # 7 is smaller than 40% (with a good performance). To improve this result, we execute again the test decreasing the MWA to 3. The precision remarkably improves (100%) but, at the same time, the performance is compromised.

5.9 Related Work

As already discussed in Section 5.1, computer games fulfil a very important role in today's software panorama. Such programs would greatly benefit from static analysis approaches which could let the developer verify not only structural but also behavioural properties of the code. Unfortunately, until now there is lack of such tools and we try to fill this gap with our proposed abstract domain $\mathcal{H}$. Parametric Hypercubes is a disjunctive non-relational abstract domain specifically tailored to analyze physics simulations (an integral part of almost every modern game) but which is also parametric on the inner domain (the one used to abstract the single variables) and could thus be exploited in many other applications.

In this section we are going to: (i) explore the similarities between $\mathcal{H}$ and other (general purpose) abstract domains known in the literature; and (ii) compare our approach with existent analysis techniques for hybrid systems (which are quite similar to our definition of physics simulation).

5.9.1 Abstract Domains

A number of different numerical abstract domains have been studied in the literature, and they can be classified with respect to a number of different dimensions: finite versus infinite height, relational versus non-relational, convex versus possibly non-convex, and so on. The computational cost increases when lifting from finite non–relational domains like *Sign* or *Parity*, to infinite non–relational domains like

Intervals, to sophisticated infinite relational domains like *Octagons* [123], *Polyhedra* [62], *Pentagons* [114], and *Stripes* [74], or to donut-like non-convex domains [78]. Moreover, when considering possibly non-convex disjunctive domains, as obtained through the power-set operator [77], the complexity of the analysis is growing (as well as its accuracy) in a full orthogonal (exponential) way. Instead, we designed a specific disjunctive domain relying on Intervals that reduces the practical complexity of the analysis by adopting indexes and offsets.

Noticeable efforts have been put both to reduce the loss of precision due to the upper bound operation, and to accelerate the convergence of the Kleene iterative algorithm. Some ways to reduce the space dimension in polyhedra computations relying on variable elimination and Cartesian factoring are introduced in [91]. Seladji and Bouissou [138] designed refinement tools based on convex analysis to express the convergence of convex sets using support functions, and on numerical analysis to accelerate this convergence applying sequence transformations. On the other hand, Sankaranarayanan et al. [137] faced the issue of reducing the computational cost of the analysis using a power-set domain, by adopting restrictions based on "on the fly elaborations" of the program's control flow graph. Efficiency issues about convergence acceleration by widening in the case of a power-set domain have been studied by Bagnara et al. in [17]. All these domains do not track disjunctive information.

The trace partitioning technique designed by Mauborgne and Rival [118] provides automatic procedures to build suitable partitions of the traces yielding to a refinement that has great impact both on the accuracy and on the efficiency of the analysis. This approach tracks disjunctive information, and it works quite well when the single partitions are carefully designed by an expert user. Unluckily, given the high number of hypercubes tracked by our analysis, this approach is definitely too slow for the scenario we are targeting.

Our spatial representation and width adjustment resembles the hierarchical data-structure of quadtrees in [101]. However, this paper contains only a preliminary discussion of the quadtree domain, and as far as we know it has not been further developed nor applied. Moreover, their domain is targeted to analyze only machine integers (while we deal with real values) and the width is the same in each spatial axis (while we use a different width for each variable and offsets).

[90] introduced the Boxes domain, a refinement of the Interval domain with finite disjunctions: an element of Boxes is a finite union of boxes. Each value of Boxes is a propositional formula over interval constraints and it is represented by the Linear Decision Diagrams data structure (LDDs). Note that the size of an LDD is exponential in the number of variables. We use a fixed width and a fixed partitioning on each hypercube dimension, while they do not employ constraints of this kind. In addition, Boxes uses a specific abstract transformer for each possible operation (for example, distinguishing $x = x + v$, $x = a \times x$, $x = a \times y$ and also making assumptions on the sign of constants) while our definitions are more generic. Finally, Boxes' implementation is based on the specific data structure of LDDs and

cannot be extended to other base domains, while our approach can.

Another similarity comes from the model checking field: our self-adaptive parametrization of the width shares some common concepts with the CounterExample Guided Abstraction Refinement (CEGAR) [39]. CEGAR begins checking with a coarse (imprecise) abstraction of the system and progressively refines it, based on spurious counterexamples seen in prior model checking runs. The process continues until either an abstraction proves the correctness of the system or a valid counterexample is generated.

5.9.2 Hybrid Systems

If on the one hand Parametric Hypercubes have been tailored to Computer Games Software applications, on the other hand some of their features may also be applied to other contexts. In particular, our definition of Computer Games Software applications (i.e., an infinite reactive loop, a complex state space with many real-valued variables, and strong dependencies among variables) exactly matches that of real-time synchronous control-command software (found in many industries such as aerospace and automotive industries). Hybrid systems [3] and hybrid automata have been widely applied to verify this software. The formal analysis of large scale hybrid systems is known to be a very difficult process [13]. In general, existing approaches suffer from performance issues or limitations on the property to prove, on the shape of the program, etc. For instance, [25] deals a simpler example than ours (a bouncing ball with only vertical motion) and in their benchmarks the variable space is quite limited: the velocity is a fixed constant, and the starting position varies only between 10 and 10.1. Instead, our Hypercubes can deal with velocities and positions bound inside any intervals of values. Also in [19] the variable space is more restricted than in our approach: the benchmark *Heater* works on the variable space [0..5], the *Navigation* one on $[3.5..3.6] \times [3.5..3.6]$ and the *Two-tanks* on $[5..5] \times [6..6]$; In addition, this analysis returns an abstraction of the final state of the program, while we also give information about which starting values are responsible for the property verification and which not. [92] presents an application of the Abstract Interpretation by means of convex polyhedra to hybrid systems. This work is focused on a particular class of hybrid systems (*linear* ones), and it is able to represent only convex regions of the space, since it employs the convex hull approximation of a set of values. [12] presents algorithms and tools for reachability analysis of hybrid systems by relying on predicate abstraction and polyhedra. However, this solution suffers from the exponential growth of abstract states and relies on expensive abstract domains. Finally, [134] concerns safety verification of non-linear hybrid systems, starting from a classical method that uses interval arithmetic to check whether trajectories can move over the boundaries in a rectangular grid. This approach is similar to ours in the data representation (boxes). However, they do not employ any concept of offset,

[3] about which we already talked in Section 4.1

their space partitioning is not fixed and the examples they experimented with cover a very limited variable space.

Concluding this survey, we can then affirm that the Parametric Hypercubes proposal presented in this chapter can be seen as a selection and a combination of most of the techniques cited in Section 5.9.1, tailored to get a solution that properly suits the features of Computer Games Software applications (and physics simulations in particular), where we need to track precisely a lot of disjunctive information. However, the parametric nature of the domain (in the abstraction of single variables) and some general domain features (like width parameter tuning, and interval offsets) make our domain suitable to be used also in more general applications than only physics simulations inside games software (hybrid systems, for instance).

5.10 Other Applications

In this section we are going to show how our domain $\mathcal{H}$ can be useful in the analysis of generic programs (changing the abstract domain on which it is parametrized). In particular, given the disjunctive non-relational nature of $\mathcal{H}$, the best applicative scenarios are those where the variables of the program are inter-related in some way without, however, having any *explicit* dependency.

For example, consider the code reported in Figure 5.14. This program manipulates the value of two integer variables (`x` and `y`) inside a while loop which goes on until a counter `i` reaches the constant value N starting from zero (and increasing by one at each iteration). Note that, inside the loop, the value of `x` is assigned anew at each iteration (line 5), receiving its value as input from outside the program (i.e., from the user or from a sensor in a physical environment). At the end of the loop, we find a `if` − `then` − `else` statement which executes two different sub-programs ($\mathtt{P}_1$ or $\mathtt{P}_2$) depending on the sign of the product of the integer variables $\mathtt{x}, \mathtt{y}$.

If we analyze this program using the *Sign* domain to abstract $\mathtt{x}, \mathtt{y}$, we obtain (at line 13, that is after the loop but before testing the branching condition) that both variables have value $\top$, i.e. no information at all. The same happens using more precise domain (*Intervals*, *Polyhedra*, etc.). This happens because in this case study there are no explicit dependencies between the values of $\mathtt{x}, \mathtt{y}$. However, by looking carefully at the code, we can figure out that the signs of `x` and `y` are always the opposite of the other. In fact:

- at line 8, `y` gets a strictly negative value, because `x` has surely a strictly positive value (since we enter that branch only if $\mathtt{x} \geq 0$ and then we execute $\mathtt{x} = \mathtt{x} + 1$, thus certainly obtaining a value greater than zero);

- at line 10, instead, `y` is strictly positive and in that branch we are sure that `x` is strictly negative.

```
1  x = 1;
2  y = -1:
3  i = 0;
4  while (i<N){
5      x = fromInput();
6      if (x >= 0)
7          x = x + 1;
8          y = -x;
9      else
10         y = (y * y) + 1;
11     i++;
12 }
13 if (x * y) > 0 {
14     P_1;
15 } else {
16     P_2;
17 }
```

Figure 5.14: A generic case study with implicit dependencies between variables

For this reason, we know that the branching condition at line 13 will always be false: thus, the sub-program $\texttt{P}_1$ will never be executed, while the sub-program $\texttt{P}_2$ will always be executed. It is difficult, however, to show this property by using existing numerical domains.

Our domain $\mathcal{H}$ is able to understand this program invariance and we are now going to see how. First of all, consider the *Sign* domain where the lattice is made by the values: $\bot, -, 0, +, \top$. We use this abstract domain to abstract the single variables of the program. We exclude the variable `i` from the analysis, since it is used only to determine the number of iterations executed the loop and we consider this number to be unknown at compile time (the analysis converges anyway). The variable space is made by two dimensions and the hypercubes are then pairs of values from the *Sign* domain. Given the hypercube $h = (s_1, s_2)$, the sign s_1 is the abstraction of `x`, while s_2 is the abstraction of `y`. The analysis using $\mathcal{H}$ the proceeds as follows:

Initialization (lines 1-3)

Since `x` has a strictly positive value and `y` a strictly negative one, the state of the program after the initialization of the variables is the singleton set:

$$H_0 = \{(+, -)\}$$

First iteration of the loop (lines 4-12)

- assignment of `x` (line 5): `x` gets an unknown value, so we have to build a new hypercube for each possible abstraction of such variable. The value of `y` remains unaltered, so each new hypercube will still have value $-$ in its second component. The resulting set of hypercubes is: $H_1 = \{(-,-),(0,-),(+,-)\}$.

- branching condition (line 6): the hypercubes set is partitioned based on the branching condition. The subset of hypercubes satisfying the condition is $H_2 = \{(0,-),(+,-)\}$, while the subset of hypercubes not satisfying the condition is $H_3 = \{(-,-)\}$.

- `then` branch (lines 7-8): we execute this branch on the subset H_2. The first statement increases the value of `x` by a positive quantity. Given the classical abstract semantics of *Sign*, we know that this operation results in a positive value, when applied both to a positive or to a zero value. Then, the hypercube $(0,-)$ transforms itself into $(+,-)$ and the hypercube $(+,-)$ remains unchanged. Since the two resulting hypercubes are the same one, the result is a singleton set containing only the hypercube $(+,-)$. The second statement of the branch assigns to `y` the opposite sign of `x`: the hypercube $(+,-)$ becomes then $(+,-)$, again (the opposite of plus is minus, so the sign of `y` does not change). The final result of this branch is the hypercubes set $H_4 = \{(+,-)\}$.

- `else` branch (line 10): we execute this branch on the subset H_3. The statement multiplies `y` by itself, and then increases this value by a positive quantity. Since in H_2 there is only the hypercube $(-,-)$, we know for sure that `y` has a strictly negative value. The multiplication of a strictly negative value by another strictly negative value returns a strictly positive value. Adding a positive quantity to a positive value returns, again, a strictly positive value. Then, the hypercube $(-,-)$ has transformed itself into $(-,+)$. The final result of this branch is the hypercubes set $H_5 = \{(-,+)\}$.

- line 11: we have to merge the result of the two branches by making the union of the two hypercubes sets. We obtain, as a result of the first iteration of the loop, the set $H_6 = H_4 \cup H_5 = \{(+,-),(-,+)\}$.

Second iteration of the loop (lines 4-12)

Since $H_6 \neq H_0$, the analysis has not converged yet and we must execute another abstract iteration of the loop, starting this time from the hypercubes set H_6.

- assignment of `x` (line 5): `x` gets an unknown value, while the value of `y` remains unaltered. The resulting set of hypercubes is: $H_7 = \{(-,-),(0,-),(+,-),(-,+),(0,+),(+,+)\}$.

- branching condition (line 6): the hypercubes set is partitioned based on the branching condition. The subset of hypercubes satisfying the condition is $H_8 = \{(0,-),(+,-),(0,+),(+,+)\}$, while the subset of hypercubes not satisfying the condition is $H_9 = \{(-,-),(-,+)\}$.

- `then` branch (lines 7-8): we execute this branch on the subset H_8. The first statement increases the value of `x` by a positive quantity: then, the hypercubes $(0,\cdot)$ transform themselves into $(+,\cdot)$ and the hypercubes $(+,\cdot)$ remain unchanged. The result is the set $\{(+,-),(+,+)\}$. The second statement of the branch assigns to `y` the opposite sign of `x`: the hypercubes $(+,\cdot)$ become all $(+,-)$ (since the opposite of plus is minus). The final result of this branch is the hypercubes set $H_{10} = \{(+,-)\}$.

- `else` branch (line 10): we execute this branch on the subset H_9. The statement multiplies `y` by itself, and then increases this value by a positive quantity. H_9 contains two hypercubes: the multiplication transforms the first hypercube $(-,-)$ into $(-,+)$ (since $-\times- = +$) and the second hypercube $(-,+)$ into itself (since $+\times+ = +$). We have obtained a set containing only one hypercube, $(-,+)$ which remains unaltered after the addition of the positive quantity to `y`. The final result of this branch is the hypercubes set $H_{11} = \{(-,+)\}$.

- line 11: we have to merge the result of the two branches by making the union of the two hypercubes sets. We obtain, as a result of the second iteration of the loop, the set $H_{12} = H_{10} \cup H_{11} = \{(+,-),(-,+)\}$.

Loop convergence

After the second iteration of the loop we have reached convergence, because $H_{12} = H_6$.

Branching condition (line 13)

The abstract state after the loop is $H_{12} = \{(+,-),(-,+)\}$. This hypercubes set is partitioned based on the branching condition. The subset of hypercubes satisfying the condition is $H_{13} = \emptyset$, while the subset of hypercubes not satisfying the condition is $H_{14} = \{(+,-),(-,+)\} = H_{12}$. From this result we get that the sub-program P_1 is never executed (since it is applied to an empty set of hypercubes).

With this example we showed that $\mathcal{H}$ is able to prove interesting properties also on programs which are not necessarily physics simulations. In this particular case, we parametrized $\mathcal{H}$ on the simple *Sign* domain. We obtained a precise and efficient analysis, since: (i) we proved the property of interest; (ii) the number of hypercubes contained in the abstract state is always very low; and (iii) the convergence is reached only after two abstract iterations of the loop.

5.11 Discussion

In this chapter we dealt with the use of Abstract Interpretation techniques to allow for static verification of behavioural properties in the field of Computer Games software. In particular, we focused on physics simulations, a very important part of modern games.

To this purpose, we designed the Parametric Hypercubes domain $\mathcal{H}$, a disjunctive non-relational abstract domain which combines in an original way concepts coming from many well-known abstraction approaches. An abstract state of this domain approximates all the variables of the program together and can be seen as a non-convex volume in the variable space; in particular, it is defined as a set of convex sub-volumes (hypercubes) in such space. The way in which each single variable is abstracted is parametrized on a base abstract domain. We defined the structure of this domain and its abstract semantics on a simple (yet expressive) language. We thoroughly described a representative case study coming from the field of game software, and showed the precision of the approach. The performance of the analysis makes it feasible to apply it in practical settings.

Even though in the presentation of the domain we focused mostly on a specific application, we showed also: (i) the similarities between physics simulations and hybrid systems; (ii) the analysis of a generic program (Section 5.10). Thus, our domain can be interesting for employment in various context other than game software, especially to analyze program with implicit relationships between variables.

6
Conclusions

In this thesis we have contributed to the field of program verification through the design of several novel abstract domains.

Our first contribution is the design of a generic framework for string analysis, composed by five abstract domains which approximate string values in different ways. The freedom offered by our framework is multiple: (i) the user can choose which domain to use for the specific analysis to perform, based on the preferred trade-off between precision and performance; (ii) since the abstracted string operators are the most common, these abstractions can be used to analyze programs written in almost any language which supports strings; (iii) other string operators can be added to the framework by defining their abstract semantics for each of the five domains; (iv) the framework is agnostic with respect to the specific property to prove, since it only defines approximations of lexical variables, and can thus be applied to many contexts. Obviously, so much freedom does not have only positive consequences: the user could make the wrong domain choice and, for each new analysis, she has to implement the code to check if the resulting abstract string satisfies the property to verify. For this reason, an important future work concerns the definition (and implementation) of a suite of property checks for all domains, by choosing a subset of interesting properties based on the most common applications of string analysis. Another future work regards the applicability of our framework in other contexts than string analysis, by generalizing even further the domains. In fact, a string can be seen also as an array of characters: thus, we could generalize our analysis in order to manage arrays of any base type (not only characters), combining it with domains which abstract relevant properties of such base types.

Our second contribution is the improvement of an existing abstract domain for the abstraction of continuous functions (IVSF, presented in [24]) through the novel domain TSF. Comparing the results given by TSF and IVSF in some case studies, we showed that TSF is indeed more precise than IVSF, while maintaining almost the same performance. To evaluate more precisely the efficacy of TSF, we could do a formal complexity analysis on the domain operations. Another ambitious and possibly very interesting extension to our domain would be to change the definition of the upper and lower sides of the trapezoids from straight lines into more complex functions (e.g., polynomials) to further increase the precision of the representation.

Even though the TSF domain was built for a specific application (i.e., to approx-

imate the values of the inputs given by the environment in the context of hybrid systems), its definition opens an interesting research direction, because the literature does not offer a lot of existing approaches to abstract continuous functions. For this reason, there are many other case studies and application fields in which TSF could be useful. For example, we could apply TSF to the approximation of the solutions of Ordinary Differential Equations (as done by IVSF) or we could explore how to use TSF to approximate the values produced by a program (e.g., a simulator of the results given by sensors in embedded systems). We could also try to use our domain to verify the continuity of programs. In addition, we concretely plan to apply TSF domain to the cost analysis of code [11]: to this purpose, we already defined the abstract semantics of the most common arithmetic operations (sum, product, composition, etc.). In this context, we must extend our approach to multivariate functions (TSF represents only univariate ones) in order to allow for more complex cost models. We already started working on the definition of a new domain for bivariate functions (PBF, Parallelepiped Block Functions) and we are close to its completion.

Our third contribution is the design of a novel approach for the efficient analysis of programs with strong relationships between variables, resulting in the definition of the Parametric Hypercubes abstract domain ($\mathcal{H}$). Applied to the analysis of a bouncing ball (a common physical event in a computer game), the domain gave satisfactory results, showing that our approach can be effectively applied to practical settings and infer interesting information. Also, its applications are not limited to physics simulations. In fact, thanks to the modularity of the domain, it is possible to track relationships between variables which do not necessarily represent physical quantities: we proved this through the successful analysis of another case study (not related to game software) where the single variables were abstracted through the *Sign* domain.

Note that our approach offers plenty of venues in order to improve its results, thanks to its flexible and parametric nature. In particular, we could: (i) increase the precision by intersecting our hypercubes with arbitrary bounding volumes which restrict the relationships between variables in a more complex way than the offsets presented in Section 5.5.4; (ii) increase the performance of Algorithm 5 by halving the widths only on some axes, chosen through an analysis of the distribution of hypercubes in the *yes,no,maybe* sets; and (iii) study the derivative with respect to time of the iterations of the main loop in order to define temporal trends to refine the widening operator (thus eliminating the need of doing loop unrolling).

As a final methodological note, observe that our three contributions are very different as far as their topics are concerned. What they strongly have in common, instead, is the approach to their definition, an approach which is induced by the Abstract Interpretation framework. First of all, when building an abstract domain, it must be clear what is the *object to abstract* and which are the *practical contexts*

for the application of such domain. Secondly, the abstraction must be defined: this implies the creation of a lattice of abstract elements, together with all the needed operations on the lattice (most importantly, the partial order and the glb and lub operators, as well as an abstraction and concretization functions, and a widening operator). To apply the abstract domain to the analysis of code, a language syntax supported by the domain needs to be identified (and this obviously depends on the target application) and approximated through the definition of a sound abstract semantics. All the abstract domains presented in this thesis have been built following this procedure.

Note also that, when building a new abstract domain, the contribution does not lie only in the abstract domain itself. In fact, abstract domains should not be seen as isolated objects: they are part of a bigger picture, composed by the Abstract Interpretation framework and all the previously defined abstract domains. Metaphorically, we could see each abstract domain as a single Lego© piece, which can be composed through some standard connectors with all other pieces to build more complex constructions. As showed in Section 2.11, there are many possibilities to combine abstract domains with each other. For example, in [153] the authors performed information leakage analysis by combining two very different domains (a symbolic one and a numeric one) through the reduced product. Another possibility of combination has been exemplified by our third contribution, where the main abstract domain was parametrized with respect to other abstract domains (to abstract the single variables): our target application employed a domain for floating point variables, but we also showed how the domain can be effective using other abstract domains (i.e., the *Sign* one), depending on the specific application to analyze. The future work concerning our novel framework for string analysis presents the same feature: we would like to exploit the overall abstractions for sequences of objects but changing the abstraction of the single objects. This will be done through the creation of a parametric abstract domain.

Bibliography

[1] OWASP. Top ten project. https://www.owasp.org/index.php/Top_10_2013-T10. Accessed: 2013-05-22.

[2] R. Skeel. Roundoff error and the Patriot missile. SIAM News, 25(4):11, 1992. http://www.siam.org/siamnews/general/patriot.htm.

[3] VDC Research (2012-02-01) - Automated Defect Prevention for Embedded Software Quality. http://alm.parasoft.com/embedded-software-vdc-report/. Accessed: 2012-04-10.

[4] Computer based safety systems - technical guidance for assessing software aspects of digital computer based protection systems. http://www.hse.gov.uk/nuclear/operational/tech_asst_guides/tast046.pdf.

[5] Coverity Static Analysis Verification Engine (Coverity SAVE). http://www.coverity.com/products/coverity-save.html. Accessed: 2013-06-05.

[6] FDA (2010-09-08). http://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/GeneralHospitalDevicesandSupplies/InfusionPumps/ucm202511.htm. Accessed: 2010-09-09.

[7] J. L. Lions et al. ARIANE 5, flight 501 failure, report by the inquiry board, 1996. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

[8] SEGA - Coverity Case Study. http://www.coverity.com/wpcme-resources/Coverity_Sega_Case_Study.pdf. Accessed: 2013-05-27.

[9] Static Code Analysis. http://www.altdevblogaday.com/2011/12/24/static-code-analysis/. Accessed: 2013-05-27.

[10] Video Games in the 21st Century: The 2010 ESA Report. http://www.theesa.com/facts/pdfs/videogames21stcentury_2010.pdf. Accessed: 2013-05-30.

[11] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *Proceedings of ESOP '07*, LNCS. Springer-Verlag, 2007.

[12] R. Alur, T. Dang, and F. Ivancic. Reachability analysis of hybrid systems via predicate abstraction. In *Hybrid Systems: Computation and Control, Fifth International Workshop, LNCS 2289*, pages 35–48. Springer-Verlag, 2002.

[13] R. Alur, T.A. Henzinger, G. Lafferriere, and G.J. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(7):971–984, 2000.

[14] G. Amato and F. Scozzari. The abstract domain of parallelotopes. *Electronic Notes Theoretical Computer Science*, 287:17–28, 2012.

[15] J. P. Aubin and A. Cellina. *Differential Inclusions: Set-Valued Maps and Viability Theory*. Springer-Verlag New York, Inc., 1984.

[16] R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2004*, pages 135–148. Springer-Verlag, 2004.

[17] R. Bagnara, P. M. Hill, and E. Zaffanella. Widening operators for powerset domains. *STTT*, 9(3-4):413–414, 2007.

[18] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 196–207. ACM, 2003.

[19] O. Bouissou. Proving the correctness of the implementation of a control-command algorithm. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673 of *Lecture Notes in Computer Science*, pages 102–119. Springer, 2009.

[20] O. Bouissou. From control-command synchronous programs to hybrid automata. In *Analysis and Design of Hybrid Systems (ADHS'12)*, June 2012.

[21] O. Bouissou, E. Conquet, P. Cousot, R. Cousot, K. Ghorbal, D. Lesens, S. Putot, M. Turin, and E. Goubault. Space software validation using abstract interpretation. In *Proceedings of DASIA 2009*, 2009.

[22] O. Bouissou, E. Goubault, S. Putot, K. Tekkal, and F. Vedrine. Hybridfluctuat: a static analyzer of numerical programs within a continious environment. In *In Proc. of the 21st Computer Aided Verification (CAV'09*, pages 620–626. Springer, 2009.

[23] O. Bouissou and M. Martel. GRKLib: a guaranteed runge-kutta library. In *Follow-up of International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*. IEEE Press, 2007.

[24] O. Bouissou and M. Martel. Abstract interpretation of the physical inputs of embedded programs. In *Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA,*

January 7-9, 2008, Proceedings, volume 4905 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 2008.

[25] O. Bouissou, S. Mimram, and A. Chapoutot. Hyson: Set-based simulation of hybrid systems. In *Proceedings of the 23rd IEEE International Symposium on Rapid System Prototyping, RSP 2012, Tampere, Finland*, pages 79–85, 2012.

[26] D. M. Bourg. *Physics for Game Developers*. O'Reilly Media, 2001.

[27] C. Brabrand, A. Møller, and M. I. Schwartzbach. Static validation of dynamically generated HTML. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 221–231, June 2001.

[28] A. Bressan and A. Cortesi. Directionally continuous selections in banach spaces. *Nonlinear Analysis, Theory, Methods and Applications*, 13(8):987–992, 1989.

[29] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *Proceedings of POPL '10*, pages 57–70, 2010.

[30] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. NavidPour. Proving programs robust. In *Proceedings of SIGSOFT FSE '11*, pages 102–112, 2011.

[31] L. Chen, A. Miné, and P. Cousot. A sound floating-point polyhedra abstract domain. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, volume 5356 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2008.

[32] L. Chen, A. Miné, J. Wang, and P. Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673 of *Lecture Notes in Computer Science*, pages 309–325. Springer, 2009.

[33] T. Choi, O. Lee, H. Kim, and K. Doh. A practical string analyzer by the widening approach. In *Proceedings of APLAS '06*, pages 374–388. Springer, 2006.

[34] F.S. Chou, C. M. Wang, and G. D. Cheng. Optimal bounding of curves by continuous piecewise linear functions. *Engineering Optimization*, 21(4):307–317, 1993.

[35] A. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *Proceedings of SAS '03*, pages 1–18. Springer-Verlag, 2003.

[36] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Static analysis for dynamic XML. Technical Report RS-02-24, BRICS, May 2002. Presented at Programming Language Technologies for XML (PLAN-X) 2002.

[37] L.O. Chua and S. M. Kang. Section-wise piecewise-linear functions: Canonical representation, properties, and applications. *Proceedings of the IEEE*, 65(6):915–929, 1977.

[38] R. Clarisó and J. Cortadella. The octahedron abstract domain. In Roberto Giacobazzi, editor, *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2004.

[39] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV), 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[40] A. Cortesi. Widening operators for abstract interpretation. In *Proceedings of SEFM '08*. IEEE, 2008.

[41] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming: open product and generic pattern construction. *Science of Computer Programming*, 38(1-3):27–71, August 2000.

[42] A. Cortesi, G. Costantini, and P. Ferrara. A survey on product operators in abstract interpretation. In A. Banerjee, O. Danvy, K.-G. Doh, and J. Hatcliff, editors, *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday,* Manhattan, Kansas, USA, 19-20th September 2013, volume 129 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 325–336. Open Publishing Association, 2013.

[43] A. Cortesi and M. Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures*, 37(1):24–42, 2011.

[44] G. Costantini. Abstract domains for static analysis of strings. Master's thesis, Ca' Foscari University of Venice, 2010.

[45] G. Costantini, P. Ferrara, and A. Cortesi. Static analysis of string values. In *Proceedings of 13th International Conference on Formal Engineering Methods, ICFEM 2011*, volume 6991 of *LNCS*, pages 505–521. Springer, 2011.

[46] G. Costantini, P. Ferrara, and A. Cortesi. Linear approximation of continuous systems with trapezoid step functions. In *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings*, volume 7705 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2012.

[47] G. Costantini, P. Ferrara, and A. Cortesi. A suite of abstract domains for static analysis of string values. *to appear in Software: Practice and Experience*, 2013.

[48] G. Costantini, P. Ferrara, G. Maggiore, and A. Cortesi. The domain of parametric hypercubes for static analysis of computer games software. In *Proceedings of 15th International Conference on Formal Engineering Methods, ICFEM 2013 (to appear)*, LNCS. Springer, 2013.

[49] P. Cousot. Mit course 16.399: Abstract interpretation.

[50] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[51] P. Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[52] P. Cousot. Integrating physical systems in the static analysis of embedded control software. In *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 135–138. Springer, 2005.

[53] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[54] P. Cousot and R. Cousot. Constructive versions of tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82:43–57, 1979.

[55] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM.

[56] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2-3):103–179, July 1992.

[57] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[58] 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.

[59] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *FPCA*, pages 170–181, 1995.

[60] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D.Monniaux, and X. Rival. The astreé analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[61] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 105–118, New York, NY, USA, 2011. ACM.

[62] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[63] M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 57–68, New York, NY, USA, 2002. ACM.

[64] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems*, FMICS '09, pages 53–69, Berlin, Heidelberg, 2009. Springer-Verlag.

[65] D. Delmas and J. Souyris. Astrée: from research to industry. In *Proceedings of the 14th international conference on Static Analysis*, SAS'07, pages 437–451, Berlin, Heidelberg, 2007. Springer-Verlag.

[66] Dr. A. Deutsch. Static verification of dynamic properties. Technical report, PolySpace Technologies, November 2003.

[67] K. Doh, H. Kim, and D. Schmidt. Abstract parsing: Static analysis of dynamically generated string output using lr-parsing technology. In *Proceedings of SAS '09*, pages 256–272. Springer-Verlag, 2009.

[68] D. H. Eberly and K. (contributor) Shoemake. *Game Physics*. Morgan Kaufmann, 2003.

[69] D.H. Eberly. *Game Physics*. Interactive 3D technology series. Elsevier Science, 2010.

[70] A. Edalat and A. Lieutier. Domain theory and differential calculus (functions of one variable). *Mathematical. Structures in Comp. Sci.*, 14(6), 2004.

[71] J. Feret. Static analysis of digital filters. In *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2986 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2004.

[72] P. Ferrara. Static type analysis of pattern matching by abstract interpretation. In *Formal Techniques for Distributed Systems, Joint 12th IFIP WG 6.1 International Conference, FMOODS 2010 and 30th IFIP WG 6.1 International Conference, FORTE 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings*, volume 6117 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2010.

[73] P. Ferrara, R. Fuchs, and U. Juhasz. Tval+ : Tvla and value analyses together. In *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, volume 7504 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2012.

[74] P. Ferrara, F. Logozzo, and M. Fähndrich. Safer unsafe code for .net. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 329–346, New York, NY, USA, 2008. ACM.

[75] P. Ferrara and P. Müller. Automatic inference of access permissions. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, volume 7148 of *Lecture Notes in Computer Science*, pages 202–218. Springer, 2012.

[76] G. Filé, R. Giacobazzi, and F. Ranzato. A unifying view of abstract domain design. *ACM Computing Surveys (CSUR)*, 28(2):333–336, June 1996.

[77] G. Filé and F. Ranzato. The powerset operator on abstract interpretations. *Theor. Comput. Sci.*, 222(1-2):77–111, 1999.

[78] K. Ghorbal, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta. Donut domains: Efficient non-convex domains for abstract interpretation. In *Proceedings of VMCAI '12*, pages 235–250. Springer-Verlag, 2012.

[79] R. Giacobazzi and F. Ranzato. Refining and compressing abstract domains. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, ICALP '97, pages 771–781, London, UK, UK, 1997. Springer-Verlag.

[80] R. Giacobazzi and F. Ranzato. Optimal domains for disjunctive abstract interpretation. *Sci. Comput. Program*, 32:1–3, 1998.

[81] R. Giacobazzi and F. Ranzato. The reduced relative power operation on abstract domains. *Theoretical Computer Science*, 216(1-2):159–211, March 1999.

[82] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *J. of the ACM*, 47(2):361–416, 2000.

[83] E. Goubault, M. Martel, and S. Putot. Some future challenges in the validation of control systems. In *European Congress on Embedded Real Time Software (ERTS)*, 2006.

[84] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *In Proceedings of SAS'06, LNCS 4134*, pages 18–34. Springer-Verlag, 2006.

[85] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of ICSE '04*, pages 645–654. IEEE Computer Society, 2004.

[86] P. Granger. Static analysis of arithmetical congruences. *Int. Journal of Computer Mathematics*, 30:165–190, 1989.

[87] P. Granger. Static analysis of linear congruence equalities among variables of a program. In Samson Abramsky and T. S. E. Maibaum, editors, *TAPSOFT 91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Brighton, UK, April 8-12, 1991, Volume 1: Colloquium on Trees in Algebra and Programming (CAAP 91)*, volume 493 of *Lecture Notes in Computer Science*, pages 169–192. Springer, 1991.

[88] P. Granger. Improving the results of static analyses programs by local decreasing iteration. In *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS, pages 68–79, London, UK, UK, 1992. Springer-Verlag.

[89] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330. ACM, 2011.

[90] A. Gurfinkel and S. Chaki. Boxes: A symbolic abstract domain of boxes. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2010.

[91] N. Halbwachs, D. Merchat, and L. Gonnord. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods in System Design*, 29(1):79–95, 2006.

[92] N. Halbwachs, P. Raymond, and Y.-E. Proy. Verification of linear hybrid systems by means of convex approximations. In *Proceedings of SAS '94*, LNCS, pages 223–237. Springer-Verlag, 1994.

[93] R. Halder and A. Cortesi. Obfuscation-based analysis of sql injection attacks. In *Proceedings of the The IEEE symposium on Computers and Communications*, ISCC '10, pages 931–938, Washington, DC, USA, 2010. IEEE Computer Society.

[94] M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings*, volume 1503 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 1998.

[95] T. A. Henzinger and P.-H. Ho. A note on abstract interpretation strategies for hybrid automata. In *Proceedings of Hybrid Systems II*, LNCS. Springer, 1995.

[96] M. Heymann, G. Meyer, and S. Resmerita. Analysis of zeno behaviors in hybrid systems. In *In: Proceedings of the 41st IEEE Conference on Decision and Control, Las Vagas, NV (2002*, pages 2379–2384, 2002.

[97] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[98] P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *Proceedings of VMCAI '11*. Springer Verlag, 2011.

[99] H. Hosoya and B. Pierce. Xduce: A statically typed xml processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, 2003.

[100] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for xml. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, January 2005.

[101] J. M. Howe, A. King, and C. Lawrence-Jones. Quadtrees as an abstract domain. *Electronic Notes in Theoretical Computer Science.*, 267(1):89–100, 2010.

[102] H. Imai and M. Iri. An optimal algorithm for approximating a piecewise linear function. *Journal of information processing*, 9(3):159–162, 1987.

[103] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation: Definitions and proofs. Technical Report CW-107, Computer Science Dept., K.U. Leuven, March 1990.

[104] G. Janssens and M. Bruynooghe. Deriving description of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2-3):205–258, 1992.

[105] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS)*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[106] T. P. Jensen. Disjunctive program analysis for algebraic data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(5):751–803, September 1997.

[107] C. Kahlert and L.O. Chua. A generalized canonical piecewise-linear representation. *IEEE Transactions on Circuits and Systems*, 37(3):373–383, 1990.

[108] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[109] S.-W. Kim and K.-M. Choe. String analysis as an abstract interpretation. In *Proceedings of VMCAI '11*. Springer Verlag, 2011.

[110] C. Kirkegaard and A. Møller. Type checking with XML Schema in Xact. Technical Report RS-05-31, BRICS, September 2005. Presented at Programming Language Technologies for XML (PLAN-X).

[111] C. Kirkegaard and A. Møller. Static analysis for Java Servlets and JSP. In *Proc. 13th International Static Analysis Symposium (SAS)*, volume 4134 of *LNCS*. Springer-Verlag, August 2006. Full version available as BRICS RS-06-10.

[112] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.

[113] B. Livshits. *Improving software security with precise static and runtime analysis*. PhD thesis, Stanford, CA, USA, 2006. AAI3242585.

[114] F. Logozzo and M. Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*, pages 184–188. ACM, 2008.

[115] J. Lygeros, C. Tomlin, and S. Sastry. *Hybrid Systems: Modeling, Analysis and Control*. 2008.

[116] R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *Proceedings of the 11th International Symposium, SAS 2004*, pages 265–279. Springer, 2004.

[117] I. Mastroeni. *Abstract Non-Interference - An Abstract Interpretation-based Approach to Secure Information Flow*. PhD thesis, University of Verona, Verona, Italia, 2005.

[118] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of ESOP '05*, pages 5–20. Springer-Verlag, 2005.

[119] I. Millington. *Game Physics Engine Development*. Morgan Kaufmann, 2007.

[120] A. Miné. A few graph-based relational numerical abstract domains. In *In Static Analysis Symp*, pages 117–132. Springer-Verlag, 2002.

[121] A. Miné, École Normale, and Supérieure Paris. A new numerical abstract domain based on difference-bound matrices. In *Proceedings of the 2nd Symposium on Programs as Data Objects (PADO 2001), volume 2053 of Lecture Notes in Computer Science*, pages 155–172. Springer-Verlag, 2001.

[122] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of WWW '05*, pages 432–441. ACM, 2005.

[123] A. Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19(1):31–100, March 2006.

[124] A. Møller, M. Østerby Olesen, and M. I. Schwartzbach. Static validation of XSL Transformations. *ACM Transactions on Programming Languages and Systems*, 29(4), July 2007.

[125] A. Møller and M. I. Schwartzbach. The design space of type checkers for XML transformation languages. In *Proc. 10th International Conference on Database Theory (ICDT)*, volume 3363 of *LNCS*, pages 17–36. Springer-Verlag, January 2005.

[126] A. Møller and M. I. Schwartzbach. XML graphs in program analysis. *Science of Computer Programming*, 76(6):492–515, June 2011. Earlier version in Proc. ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM) 2007.

[127] A. Møller and M. Schwarz. HTML validation of context-free languages. In *Proc. 14th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 6604 of *LNCS*. Springer-Verlag, March 2011.

[128] D. Monniaux. *Analyse statique : de la théorie à la pratique*. Habilitation to direct research, Université Joseph Fourier, Grenoble, France, jun 2009.

[129] R. E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliff, New Jersey, 1966.

[130] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Internet Technol.*, 5(4):660–704, November 2005.

[131] A. Mycroft. Completeness and predicate-based abstract interpretation. In *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '93, pages 179–185, New York, NY, USA, 1993. ACM Press.

[132] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[133] E. Noether. Idealtheorie in ringbereichen. *Mathematische Annalen*, 1921.

[134] S. Ratschan and Z. She. Safety verification of hybrid systems by constraint propagation based abstraction refinement. In *Proceedings of HSCC 2005*, volume 3414 of *Lecture Notes in Computer Science*, pages 573–589. Springer, 2005.

[135] J. Reed and B. C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *Proceedings of ICFP '10*, pages 157–168, 2010.

[136] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007.

[137] S. Sankaranarayanan, F. Ivančić, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *Proceedings of the 13th international conference on Static Analysis*, SAS'06, pages 3–17. Springer-Verlag, 2006.

[138] Y. Seladji and O. Bouissou. Fixpoint computation in the polyhedra abstract domain using convex and numerical analysis tools. In *Proceedings of VMCAI '08*, pages 149–168. Springer-Verlag, 2013.

[139] A. Simon, A. King, and J. M. Howe. Two variables per linear inequality as an abstract domain. In *Proceedings of the 12th international conference on Logic based program synthesis and transformation*, LOPSTR'02, pages 71–89, Berlin, Heidelberg, 2003. Springer-Verlag.

[140] J. Souyris and D. Delmas. Experimental assessment of astrée on safety-critical avionics software. In *Proceedings of the 26th international conference on Computer Safety, Reliability, and Security*, SAFECOMP'07, pages 479–490, Berlin, Heidelberg, 2007. Springer-Verlag.

[141] F. Spoto. Julia: A generic static analyser for the java bytecode. In *Proceedings of FTfjP'2005*, 2005.

[142] N. Tabuchi, E. Sumii, and A. Yonezawa. Regular expression types for strings in a text processing language. *Electr. Notes Theor. Comput. Sci.*, 75:95–113, 2002.

[143] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[144] P. Thiemann. Grammar-based analysis of string expressions. In *Proceedings of TLDI '05*, pages 59–70. ACM, 2005.

[145] I. Tomek. Two algorithms for piecewise-linear continuous approximation of functions of one variable. *IEEE Trans. Comput.*, 23(4):445–448, 1974.

[146] A.J. van der Schaft and J.M. Schumacher. *An Introduction to Hybrid Dynamical Systems*. Lecture Notes in Control and Information Sciences. Springer, 2000.

[147] P. van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3):179–208, 1995.

[148] A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded c programs. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 231–242, New York, NY, USA, 2004. ACM.

[149] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. Dover Publications, Incorporated, 1994.

[150] F. Yu, T. Bultan, M. Cova, and O. Ibarra. Symbolic string verification: An automata-based approach. In *Proceedings of SPIN '08*, 2008.

[151] F. Yu, T. Bultan, and B. Hardekopf. String abstractions for string verification. In *Model Checking Software - 18th International SPIN Workshop, Snowbird, UT, USA, July 14-15, 2011. Proceedings*, volume 6823 of *Lecture Notes in Computer Science*, pages 20–37. Springer, 2011.

[152] F. Yu, T. Bultan, and O. H. Ibarra. Relational string verification using multi-track automata. In *Implementation and Application of Automata - 15th International Conference, CIAA 2010, Winnipeg, MB, Canada, August 12-15, 2010. Revised Selected Papers*, volume 6482 of *Lecture Notes in Computer Science*, pages 290–299. Springer, 2010.

[153] M. Zanioli and A. Cortesi. Information leakage analysis by abstract interpretation. In *SOFSEM 2011: Theory and Practice of Computer Science - 37th Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 22-28, 2011. Proceedings*, volume 6543 of *Lecture Notes in Computer Science*, pages 545–557. Springer, 2011.

[154] M. Zanioli, P. Ferrara, and A. Cortesi. Sails: static analysis of information leakage with sample. In *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*, pages 1308–1313. ACM, 2012.