University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

KRZYSZTOF JAKUBCZYK

Source Code Analysis Techniques in Property Verification of Real Java Code

PhD dissertation

Supervisor
dr hab. Aleksy Schubert

Institute of Informatics
University of Warsaw

April 2013

Author's declaration:
aware of legal responsibility I hereby declare that I have written this dissertation myself and all the contents of the dissertation have been obtained by legal means.

April 10, 2013	. . . .
date	*Krzysztof Jakubczyk*

Supervisor's declaration:
the dissertation is ready to be reviewed

April 10, 2013	. . . .
date	*dr hab. Aleksy Schubert*

Abstract

The rapid growth of computer industry requires creating large, highly complicated and sophisticated software. This implies increasing probability for errors, bugs and failures. Various software verification techniques are used to ensure quality of produced programs. Unfortunately, verification using *formal methods* is not very popular, because it is considered not practical and expensive. Therefore, *formal methods* are used to verify only high risk programs, such as control software for nuclear power plants or flight control software in airplanes.

The goal of this thesis is to design a static analysis technique that uses *formal methods* and can be applied to real, large computer software created in Java language. Three topics were raised.

First, the thesis focuses on the *abstract interpretation* framework, which is a theory of sound approximation of program semantics. In particular, we are interested in *numerical abstract domains*. We propose a new approach on the abstract domain of *boxes*, which is a *disjunctive refinement* of the domain of *intervals*, and we introduce *thresholds* in the construction of the widening operator for the domain. We present a construction of domain elements based on the *sweeping line* technique, implementation of domain operators, transfer function and widening operator. We introduce two versions of the widening operator: a generic one, and the second one with a theorem about one-step precision of the operator depending on *thresholds*.

Next, practicality of *formal methods* is investigated. A tool *CodeStatistics* is introduced, that makes it possible to discover particular coding patterns on large Java projects and to generate specifications. An experiment is described, where the tool was successfully used to generate JML loop termination specifications on a set of large and popular Java projects.

Finally, an extension of the pattern discovery technique from the second part by the use of a semantic analysis is presented, in particular by *abstract interpretation*. It is shown that the combination is useful in evaluating abstract interpretation domains on real code. Additionally, it is presented that the new widening operator introduced in the first part of the thesis is more precise in practice than the one known so far.

Key words: static analysis, formal methods, abstract interpretation, widening operator, Java, termination, specification generation

ACM Classification: D.2.4, F.3.1, F.3.2

Streszczenie

Szybki rozwój przemysłu komputerowego wymaga budowy coraz potężniejszych i coraz bardziej skomplikowanych systemów, które potrzebują bardzo zaawansowanego i złożonego oprogramowania. Niesie to ze sobą rosnące prawdopodobieństwo wystąpienia błędów. Różne techniki weryfikacji programów są stosowane do zwiększenia jakości oprogramowania. Niestety, techniki wykorzystujące *metody formalne* nie są zbyt popularne, ponieważ uważa się je za niepraktyczne i drogie. Zastosowanie *metod formalnych* jest ograniczone do weryfikacji systemów zwiększonego ryzyka, takich jak oprogramowanie sterujące elektrownią jądrową czy oprogramowanie kontroli lotu w samolotach.

Podstawowym celem niniejszej rozprawy doktorskiej jest stworzenie techniki analizy statycznej opartej na *metodach formalnych*, którą można stosować na rzeczywistych programach pisanych w języku Java. Poruszono trzy zagadnienia.

Najpierw skoncentrowano się na technikach *abstrakcyjnej interpretacji*, która jest teorią przybliżania semantyki programu. W szczególności rozważane są *numeryczne dziedziny abstrakcyjne*. Zaproponowano nowe spojrzenie na dziedzinę *pudełek*, która jest *dysjunktywnym rozszerzeniem* dziedziny *przedziałów*, oraz zastosowano techniki *punktów progowych* w operatorze rozszerzającym dla tej dziedziny. Przedstawiono konstrukcję elementów dziedziny *pudełek* bazującą na technice *zamiatania* wraz z operacjami kratowymi, funkcją transferu i operatorem rozszerzającym. Wprowadzono dwie wersje operatora rozszerzającego: pierwszą ogólną, drugą z twierdzeniem o precyzji jednego kroku rozszerzania uzależnionej od *punktów progowych*.

Następnie skupiono się na praktyczności metod formalnych. Przedstawiono narzędzie *CodeStatistics*, które służy to wyszukiwania wzorców w kodzie źródłowym Javy i umożliwia generowanie specyfikacji. Zostało ono z sukcesem wykorzystane do wygenerowania warunków terminacji pętli w formacie klauzul *decreases* języka JML na zestawie dużych, popularnych projektów.

Na koniec zaprezentowano rozszerzenie techniki wyszukiwania wzorców będącej przedmiotem rozważań etapu poprzedniego, o techniki analizy semantycznej, w szczególności *abstrakcyjnej interpretacji*. Wykazano, że połączenie takie może zostać praktycznie zastosowane do oceny dziedzin abstrakcyjnej interpretacji na rzeczywistych programach. Ponadto pokazano, że nowy operator rozszerzający z części pierwszej daje wyniki dokładniejsze niż dotychczas znany.

Słowa kluczowe: analiza statyczna, metody formalne, abstrakcyjna interpretacja, operator rozszerzający, Java, terminacja, generowanie specyfikacji

Klasyfikacja tematyczna ACM: D.2.4, F.3.1, F.3.2

Acknowledgements

I wish to thank, first and foremost, my supervisor, dr hab. Aleksy Schubert, for the patience, guidance and advice throughout my time as his student. He gave me the opportunity to start working on the topic, and since then has been taking care of my development. His numerous insightful discussions and comments inspired my work.

I must express my deepest gratitude to my family, especially to my father for the continuous encouragement and involvement, and to my mother for the support throughout the way. They were always there for me.

Last but not least, I would like to thank one special person, who did not want to be mentioned here by name. Your support was essential to me.

During the preparation of the thesis I was supported by Polish government grant N N206 493138 and Information Society Technologies programme of the European Commission FET project IST-2005-015905 MOBIUS.

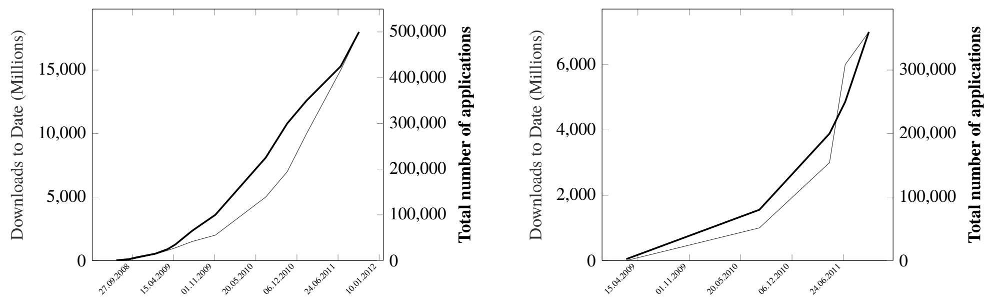
Contents

Chapter 1

Introduction

1.1 Motivation

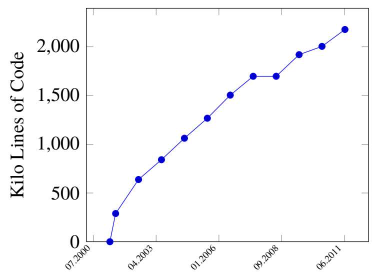
Due to the popularity of computers, computer software became a very large industry. According to market researcher DataMonitor [45], the value of the worldwide software industry in 2009 was 242.4 billion dollars. DataMonitor forecasts that in 2014 the global software market will have a value of 330 billion dollars, which is a 36,1% increase since 2009. According to a report by Market and Markets [78], the mobile application marketplace in 2010 was around 6.8 billion dollars and is estimated to reach 25 billion dollars by 2015. Fig. 1.1 presents growing numbers for the most popular mobile application stores — Apple AppStore and Android Market.



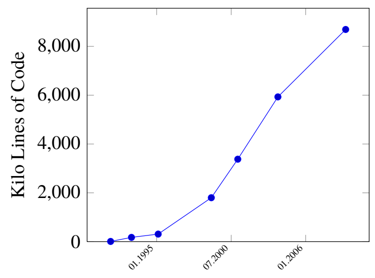
(a) Apple AppStore (b) Android Market

Figure 1.1: Statistics of number of applications and total number of downloads for the most popular smartphone application stores.

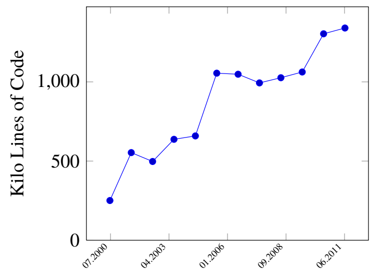
The increasing popularity of computers results in building more powerful and complex computer systems. They require large, highly complicated and sophisticated software. Fig. 1.2 presents increasing size of source code for a few very popular open source projects.



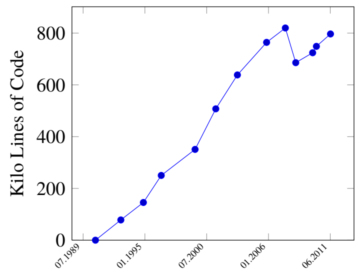
(a) Eclipse programming IDE



(b) Linux Kernel



(c) MySQL database



(d) Python programming language

Figure 1.2: Project growth in time for selected open source projects.

The growth of both complexity and size of programs implies increasing probability for errors, bugs and failures. Unless, of course, the production process is strictly supervised. Usually errors and crashes are harmless but there are some areas, where errors simply cannot happen. Errors in programs have caused a number of spectacular failures:

Mariner 1 [21] A bug in the flight software of a NASA probe Mariner 1 caused the rocket to divert from the desired path on launch. The investigation discovered that the error was in a hand-transcription of a mathematical formula in the program specification.

Therac-25 [76] Therac-25 was a radiation therapy machine. Between June 1985 and January 1987 six people were overexposed during treatments by the machine, which resulted in deaths and serious injuries. As it turned out, the cause of the accidents was the software that operated the machine. It was written by a single person, there was no specification involved, very little documentation was produced during the development. Additionally, there were lacks of quality control and finally hardly any tests were executed.

Ariane 5 [48] On June 4, 1996 a rocket Ariane 5 self-destructed 37 seconds after launch due to a malfunction in the control software. The problem was that the newly designed rocket reused the navigation package code from the previous version — Ariane 4. The new rocket was faster, therefore some values became higher. Just after the start of Ariane 5, a conversion from 64-bit floating-point number to a 16-bit integer generated an overflow. The error was caught but the code executed shut down whole subsystem and rocket veered off the course and exploded. Fortunately, there were no victims but the losses reached approximately 370 million dollars.

As computers and programs become more and more crucial to human civilisation, one should put more effort to ensure high quality of produced software. There are various methods that can be used to serve the purpose.

1.2 Software Verification

Software verification can be considered as any type of analysis, the goal of which is to find errors in programs or check if program does what it was intended to do. In general, by applying verification one wants to assure that the software satisfies the requirements. There are two possible approaches to software verification: dynamic and static.

1.2.1 Dynamic Software Verification

The concept of *dynamic software verification* (or *runtime verification*) covers testing in general — unit tests, integration tests, system tests, functional tests, acceptance tests, stress tests etc. Dynamic software verification is the most popular in practice, mostly because the actual product is tested thus it is easier to understand by non-technical people. The main problem of such approach is that it is usually not feasible to cover all possible executions, therefore one cannot be sure if the software that passes all the tests does not contain any errors and meets all the requirements.

The most popular method to ensure the quality of software in practice is *unit testing* [11]. The idea is to create a set of tests for each unit of the source code. The goal is to show that every individual unit works as expected, where the expected behaviour is expressed by placing assertions in the unit test scenario. *Unit testing* is often used internally in companies to assure that the produced code works properly. There is also a practice called *continuous integration*, which relies on unit tests. The idea is that every change in the source code that is applied to a source code repository triggers an automatic build process of the changed application and execution of a number of unit tests for that application. This way, one can assure that every build that is created presents some quality. The *continuous integration* methodology plays very nicely with *regression tests*: for example, when an error in application is found, a new unit test is written to cover the scenario, so that every future build will be tested against that particular error.

Unit tests are also often used by the other side of a contract — clients. Usually, when a piece of software is ordered, a series of tests is prepared — they are so called *acceptance tests*. Unit tests may be a part of them. Acceptance tests may be prepared by either client, the producer or by both sides. Passing the tests results in the acceptance of the product and the final payment for the work.

In practice unit tests work quite well. The main problem is that it is hard to reach satisfactory code coverage. Usually, it is not feasible to create scenarios for all possible executions of a program.

1.2.2 Static Software Analysis

The *static analysis* is the analysis of program code, which may be either the source code form, bytecode or even machine code, without actually executing the program itself. The notion of static software verification is a very wide subject, it covers:

Checking code conventions — whether the source code meets some guidelines specific for the programming language used. These conventions cover quite broad spectrum: from organisation of files, naming conventions to indentation, comments or white space. Code conventions improve readability of the source code, which allows engineers to understand it much quicker. This is important for two main reasons: the first one is that about 80% of lifetime cost of software goes to maintenance [82]. The second one is that hardly any software is maintained by the same author for its whole lifetime.

Detection of bad practices — searching for so-called *anti-patterns*, which are patterns that are very common but known to be faulty, ineffective or counterproductive [19]. When such *anti-pattern* is detected, by means of refactorisation, one may fix the code and introduce design patterns [19, 55], which are a commonly known to be reliable. Additionally, since design patterns are popular and people understand them, the code is more comprehensive.

Calculation of software metrics — measure some property of software. There are some metrics that help to find fragments of code that are very complex, e.g. McCabe's cyclomatic complexity metrics [80] measures the number of linearly independent paths through the source code of a program (or method). Other, such as *Depth of Inheritance* or *Number of Children*, may suggest bad design of class structure.

Formal verification — this is the analysis that uses mathematical *formal methods* in order to prove some property of the analysed program. Based on some mathematical theory one can conclude that the program, when executed, will not contain a specific type of error, e.g. integer overflow error or array out of bounds error.

In the thesis we focus on the last item — *formal verification*. In the rest of the thesis by *verification* we mean static verification using formal methods. One of popular formal methods is *abstract interpretation*. It may be used to verify software, e.g. it was successfully applied to ensure quality of the Airbus A340 and A380 flight control software [47]. Also it might be applied to automatically infer invariants of a program. A great advantage of abstract interpretation is that the technique is fully automatic. It might be imprecise but does not require any attention from the user.

Commercial applications often do not make use of specification and verification techniques, because people believe that such theoretical solutions are not applicable to software created for the real market. Business managers know that employing formal methods would increase quality of produced software, but these methods are considered to be very expensive and impractical. Therefore, program specification and verification methods are used only in critical components or high risk software, e.g. avionics [30] or in nuclear power plants control software [93], places where correctness and reliability is crucial [13]. It is considered to be not profitable to apply these techniques to standard code.

1.3 Goal of the Thesis

The goal of the thesis is to develop a method that could be used to investigate practicality of a particular software verification technique, abstract interpretation, in real Java code. We divide this into three steps:

- we present a numerical abstract interpretation domain with new: generic and highly configurable widening operator,

- we create a tool that makes it possible to discover particular coding patterns on large Java projects and to generate specifications (we show its usefulness through generation of JML loop termination specifications),
- we also apply the tool to evaluate the strength of the developed domain and the proposed widening operator.

1.4 Overview of the Thesis

The dissertation is organised into seven chapters. The current chapter provides the introduction to the thesis including motivation and goals. Chapter 2 provides necessary mathematical definitions, introduction to the *abstract interpretation* and a brief presentation of the Java Modeling Language. Chapter 3 is an overview of numerical abstract domains and existing approaches to path-sensitive analysis.

Chapter 4 is the main theoretical contribution of the thesis — we introduce a new approach to the abstract domain of *boxes* and a new widening operator for the domain. The preliminary results have been the subject of a publication [69]. We present here an extended version and complete proofs.

Chapter 5 is a description of *CodeStatistics* — a tool for Java language, that can be used to find patterns in code and compute statistics of their occurrences. We also present an application of *CodeStatistics* — an experiment that shows that we are able to generate about 80% of termination conditions for numerical `for` loops. The preliminary results have already been published in [54].

Chapter 6 combines two previous chapters. First, we describe an implementation of an abstract interpreter tool for Java — *JavaAI*. The domain of *boxes* with the proposed widening operator is implemented in the analyser. Next, we present how a combination of pattern discovery extended by results of the abstract interpretation analysis can be applied for practical evaluation of abstract domains.

Finally, in Chapter 7 we conclude the thesis.

Companion disk

The companion disk included with the thesis provides the following:

- A VirtualBox[1] virtual machine: Lubuntu 12.04 Linux distribution with full environment configuration to run the software created for the thesis.
- Built version and sources of both *CodeStatistics* and *JavaAI*.
- Output and input files of both experiments.

Further instructions are in the README.txt file on the companion disk. The contents of the disk is also available at `http://www.mimuw.edu.pl/~kjk/phd/`.

[1]For details see: `https://www.virtualbox.org/`

Chapter 2

Preliminaries

In this chapter, we introduce definitions and notations that are used throughout the thesis. In Section 2.1 and Section 2.2 we focus on basic mathematical concepts. Next, in Section 2.3 we define a simple imperative language, which we use later in the abstract interpretation analysis. In Section 2.4 we introduce the *abstract interpretation* framework. Finally, Section 2.5 is a brief introduction to the Java Modeling Language that is used later in Chapter 5.

2.1 Relations, Orders and Lattices

We start with fixing basic notation used throughout the thesis. In programming, language semantics are usually expressed in terms of partial orders.

Definition 2.1.1. A *partially ordered set* (or a *poset*) is an ordered pair $\langle \mathcal{D}, \leq \rangle$, where $\mathcal{D}$ is a set and $\leq\, : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is a binary relation that fulfils the following conditions:

- *reflexivity* — for all $a \in \mathcal{D}$ it holds that $a \leq a$,
- *anti-symmetry* — for all $a, b \in \mathcal{D}$ if $a \leq b$ and $b \leq a$ then $a = b$,
- *transitivity* — for all $a, b, c \in \mathcal{D}$ if $a \leq b$ and $b \leq c$ then $a \leq c$.

Let $\langle \mathcal{D}, \leq \rangle$ be a partially ordered set and $\mathcal{X}$ be any subset of $\mathcal{D}$. For $x_1, x_2 \in \mathcal{D}$ we write $x_1 < x_2$ to denote $x_1 \leq x_2 \wedge x_1 \neq x_2$. An element $a \in \mathcal{D}$ is called an *upper bound of* $\mathcal{X}$ if for all $x \in \mathcal{X}$ it holds that $x \leq a$. Dually, $b \in \mathcal{D}$ is called a *lower* *bound of $\mathcal{X}$* if for all $x \in \mathcal{X}$ it holds that $b \leq x$. We call $a \in \mathcal{X}$ the *greatest element in $\mathcal{X}$* if for any $x \in \mathcal{X}$ it holds that $x \leq a$. The greatest element in the whole set $\mathcal{D}$, if it exists, is denoted by $\top$. Analogically, $a \in \mathcal{X}$ is the *least element in $\mathcal{X}$* if for any $x \in \mathcal{X}$ it holds that $a \leq x$. The least element in whole set $\mathcal{D}$, if it exists, is denoted by $\perp$.

Definition 2.1.2. Let $\langle \mathcal{D}, \leq \rangle$ be a partially ordered set and $\mathcal{X} \subseteq \mathcal{D}$. An element $a \in \mathcal{D}$ is the *least upper bound of $\mathcal{X}$* if it is an upper bound of $\mathcal{X}$ and for all $b \in \mathcal{D}$ if b is an upper bound of $\mathcal{X}$ then $a \leq b$.

Definition 2.1.3. Let $\langle \mathcal{D}, \leq \rangle$ be a partially ordered set and $\mathcal{X} \subseteq \mathcal{D}$. An element $b \in \mathcal{D}$ is the *greatest lower bound of $\mathcal{X}$* if it is a lower bound of $\mathcal{X}$ and for all $a \in \mathcal{D}$ if a is a lower bound of $\mathcal{X}$ then $a \leq b$.

Both the *least upper bound* and the *greatest lower bound* may not exist. If the *least upper bound of $\mathcal{X}$* exists, we denote it as $\bigsqcup \mathcal{X}$. Analogically, if exists, the *greatest lower bound* is denoted by $\bigsqcap \mathcal{X}$.

We say that a partially ordered set $\mathcal{D}$ satisfies the *ascending chain condition*, if for any sequence of elements from $\mathcal{D}$: $x_1 \leq x_2 \leq \ldots$ there exists an index $n > 0$ such that: $x_n = x_{n+1} = x_{n+2} = \ldots$, i.e. the sequence is eventually constant.

Definition 2.1.4. A *lattice* is a partially ordered set $\langle \mathcal{D}, \leq \rangle$ such that for every pair of elements $a, b \in \mathcal{D}$:

- the set $\{a, b\}$ has the least upper bound *(join)*, denoted as $a \vee b$,
- the set $\{a, b\}$ has the greatest lower bound *(meet)*, denoted as $a \wedge b$.

When one needs to prove that a partially ordered set is a lattice, an equivalent definition may be useful:

Definition 2.1.5. A *lattice* is an algebraic structure $\langle \mathcal{D}, \wedge, \vee \rangle$, where $\wedge, \vee : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ are binary operations that are:

- *commutative* — for all $a, b \in \mathcal{D}$ it holds that $a \vee b = b \vee a$ and $a \wedge b = b \wedge a$,
- *associative* — for all $a, b, c \in \mathcal{D}$ it holds that $a \vee (b \vee c) = (a \vee b) \vee c$ and $a \wedge (b \wedge c) = (a \wedge b) \wedge c$,
- *idempotent* — for all $a \in \mathcal{D}$ it holds that $a \vee a = a$ and $a \wedge a = a$,
- fulfil *absorption laws* — for all $a, b \in \mathcal{D}$ it holds that $a \vee (a \wedge b) = a$ and $a \wedge (a \vee b) = a$.

The equivalence between these two definitions can be seen by an isomorphism, where $x \leq y \iff x \vee y = x \iff x \wedge y = y$ and $\bigsqcup \{x, y\} = x \vee y$, $\bigsqcap \{x, y\} = x \wedge y$. In the thesis we find two variants of a lattice especially interesting. The first one is a *bounded lattice*:

Definition 2.1.6. We say that a lattice $\langle \mathcal{D}, \leq \rangle$ is *bounded* if there exist two elements $\bot, \top \in \mathcal{D}$ such that $\bot = \bigsqcap \mathcal{D}$ and $\top = \bigsqcup \mathcal{D}$.

The second interesting variant is a *complete lattice*:

Definition 2.1.7. We say that a lattice $\langle \mathcal{D}, \leq \rangle$ is *complete* if for every subset $\mathcal{S} \subseteq \mathcal{D}$ both $\bigsqcup \mathcal{S}$ and $\bigsqcap \mathcal{S}$ exist in $\mathcal{D}$.

In some situations we are not interested in the meet operation:

Definition 2.1.8. A *join-semilattice* (or *upper semilattice*) is a partially ordered set that has a least upper bound for any nonempty finite subset.

We write $\langle \mathcal{D}, \leq, \bot, \cup \rangle$ to denote a join-semilattice, where $\bot$ is the least element and $\cup$ is the least upper bound operator.

2.2 Functions and Fixpoints

We proceed with definitions of functions and overview fixpoint theorems that are later required by the abstract interpretation framework.

We say that f is a function from $\mathcal{X}$ to $\mathcal{Y}$ and denote by $f : \mathcal{X} \to \mathcal{Y}$. We write $f(x)$ for the result of f on x. Sometimes a generalisation of a function may be required:

Definition 2.2.1. A *partial function* from $\mathcal{X}$ to $\mathcal{Y}$, denoted by $f : \mathcal{X} \rightharpoonup \mathcal{Y}$, is a function $f : \mathcal{X}' \to \mathcal{Y}$, for some $\mathcal{X}' \subseteq \mathcal{X}$.

Let $f : \mathcal{X} \rightharpoonup \mathcal{Y}$ be a partial function. We say that the set $dom(f) = \{x \in \mathcal{X} \mid \exists y \in \mathcal{Y} : f(x) = y\}$ is a *domain* of f. If f is a function then $dom(f) \overset{\text{def}}{=} \mathcal{X}$. An *image* of f is defined as $im(f) \overset{\text{def}}{=} \{y \in \mathcal{Y} \mid \exists x \in \mathcal{X} : f(x) = y\}$.

Definition 2.2.2. Let $f : \mathcal{X} \to \mathcal{Y}$ be a function and $\langle \mathcal{X}, \leq_{\mathcal{X}} \rangle$, $\langle \mathcal{Y}, \leq_{\mathcal{Y}} \rangle$ be two partially ordered sets. The function f is *monotone* or *order-preserving* if it preserves orderings, that is for all $x_1, x_2 \in \mathcal{X}$ if $x_1 \leq_{\mathcal{X}} x_2$ then $f(x_1) \leq_{\mathcal{Y}} f(x_2)$.

Definition 2.2.3. Let $\langle \mathcal{X}, \leq_{\mathcal{X}} \rangle$ and $\langle \mathcal{Y}, \leq_{\mathcal{Y}} \rangle$ be partially ordered sets. An order-preserving function $f : \mathcal{X} \to \mathcal{Y}$ is *continuous* or *limit-preserving* if for any $\mathcal{A} \subseteq \mathcal{X}$ it holds that:

$$f(\bigsqcup(\mathcal{A})) = \bigsqcup(f(\mathcal{A})),$$

where $f(\mathcal{Z}) \overset{\text{def}}{=} \{f(z) \mid z \in \mathcal{Z}\}$ for $\mathcal{Z} \subseteq \mathcal{X}$.

Let $f : \mathcal{X} \to \mathcal{Y}$ and $g : \mathcal{Y} \to \mathcal{Z}$ be two functions. Then $g \circ f$ denotes a function $h : \mathcal{X} \to \mathcal{Z}$ such that for any $x \in \mathcal{X}$ it holds that $h(x) = g(f(x))$.

The semantics of a program can be expressed in terms of a fixpoint of a semantic function. We introduce definition of a fixpoint and important theorems about existence of fixpoints, and how they can be computed.

Definition 2.2.4. A *fixpoint* of a function $f : \mathcal{X} \to \mathcal{X}$ is an element $x \in \mathcal{X}$ such that $f(x) = x$.

Definition 2.2.5. The *least fixpoint* of a function $f : \mathcal{X} \to \mathcal{X}$, where $\langle \mathcal{X}, \leq \rangle$ is a partially ordered set, is an element $a \in \mathcal{X}$ such that $f(a) = a$ and for all $x \in \mathcal{X}$ if $f(x) = x$ then $a \leq x$.

Definition 2.2.6. The *greatest fixpoint* of a function $f : \mathcal{X} \to \mathcal{X}$, where $\langle \mathcal{X}, \leq \rangle$ is a partially ordered set, is an element $a \in \mathcal{X}$ such that $f(a) = a$ and for all $x \in \mathcal{X}$ if $f(x) = x$ then $x \leq a$.

We denote the *least fixpoint* of a function f by $\mathit{lfp}\left(f\right)$ and the *least fixpoint* of a function f that is greater than x by $\mathit{lfp}_x\left(f\right)$. Dually, by $\mathit{gfp}\left(f\right)$ we denote the *greatest fixpoint* of f and by $\mathit{gfp}_x\left(f\right)$ the *greatest fixpoint* of f that is smaller than the element x.

Now we recall *Tarski's fixpoint theorem* that ensures the existence of a fixpoint in a complete lattice.

Theorem 2.2.7 (Tarski's Fixpoint Theorem). *Let $\langle \mathcal{D}, \leq \rangle$ be any complete lattice and a function $f : \mathcal{D} \to \mathcal{D}$ be monotone. The set $\mathcal{P}$ of all fixpoints of the function f is not empty and $\langle \mathcal{P}, \leq \rangle$ is a complete lattice, in particular:*

$$\mathit{gfp}\left(f\right) = \bigsqcup \mathcal{P} = \bigsqcup \{x \in \mathcal{P} \mid x \leq f(x)\},$$
$$\mathit{lfp}\left(f\right) = \bigsqcap \mathcal{P} = \bigsqcap \{x \in \mathcal{P} \mid f(x) \leq x\}.$$

Proof. See [90]. □

The most important implication of Theorem 2.2.7 is the existence of both the least and the greatest fixed point. Unfortunately, the theorem does not yield a construction that could be used to find a fixpoint.

The following two theorems characterise fixpoints in a constructive fashion. Theorem 2.2.8 uses an ascending Kleene's chain to obtain the least fixpoint of a function f — therefore it presents how the fixpoint can be computed.

Theorem 2.2.8 (Kleene's Fixpoint Theorem). *Let $\langle \mathcal{D}, \leq \rangle$ be a complete partial order and $f : \mathcal{D} \to \mathcal{D}$ be a continuous function. Then the least fixpoint of f is the limit of the following iteration sequence:*

$$\begin{aligned} f^0(x) &= x, \\ f^{n+1}(x) &= f(f^n(x)), \end{aligned}$$

that is:

$$lfp\,(f) = \bigsqcup \{f^n(\bot) \mid n \in \mathbb{N}\}.$$

Proof. See [29, Lecture 12]. □

Next, Theorem 2.2.9 is a constructive version of Tarski's theorem, by Cousot and Cousot.

Theorem 2.2.9 (Constructive Version of Tarski's Lattice Theoretical Fixpoint Theorem). *Let $\langle \mathcal{D}, \leq \rangle$ be a complete lattice and a function $f : \mathcal{D} \to \mathcal{D}$ be monotone. The set $\mathcal{P}$ of all fixpoints of function f is not empty and $\langle \mathcal{P}, \leq \rangle$ is a complete lattice, where:*

- $gfp\,(f) = \bigsqcup \mathcal{P} = \bigsqcup \{f^n(\top) \mid n \in \mathbb{N}\}$,
- $lfp\,(f) = \bigsqcap \mathcal{P} = \bigsqcap \{f^n(\bot) \mid n \in \mathbb{N}\}$,
- *the least upper bound operator is* $\lambda \mathcal{X}.\, \bigsqcup \{f^n(\mathcal{X}) \mid n \in \mathbb{N}\}$,
- *the greatest lower bound operator is* $\lambda \mathcal{X}.\, \bigsqcap \{f^n(\mathcal{X}) \mid n \in \mathbb{N}\}$.

Proof. See [36, Theorem 5.1]. □

2.3 Programming Language *Simple*

In this section we introduce *Simple* — a small programming language, which is used throughout the thesis. The purpose of this language is to instantiate a theory that is presented in the thesis, and show some examples. *Simple* is a plain sequential language that does not contain procedures nor pointers, however it has **if** and **while** loop constructs. There is only one data type that is a selected numeric set, which is either a set of reals $\mathbb{R}$, rationals $\mathbb{Q}$ or integers $\mathbb{Z}$. All instructions in the language have unique labels from the set of labels — *Lab*.

<*var*>	::=	variable from the set of variables *Var*
<*const*>	::=	numeric constants from *Num*
<*atom*>	:: =	<*var*> | <*const*>
<*expr*>	::=	<*atom*> | − <*atom*> | <*atom*> + <*atom*>
	|	<*atom*> − <*atom*> | <*atom*> $*$ <*atom*>
<*test*>	::=	<*atom*> = <*atom*> | <*atom*> $\neq$ <*atom*>
	|	<*atom*> < <*atom*> | <*atom*> $\leq$ <*atom*>
	|	<*test*> **and** <*test*> | <*test*> **or** <*test*>
	|	**not** <*test*>
<*instr*>	::=	<*var*> $\leftarrow$ <*expr*> | **skip**
	|	**if** <*test*> **then** <*block*> **else** <*block*> **fi**
	|	**while** <*test*> **do** <*block*> **od**
<*block*>	::=	<*instr*> **;** <*block*> | <*instr*> **;**
<*program*>	::=	**program** <*block*> **end**

Figure 2.1: The syntax of the *Simple* language described using BNF notation.

2.3.1 Language Syntax

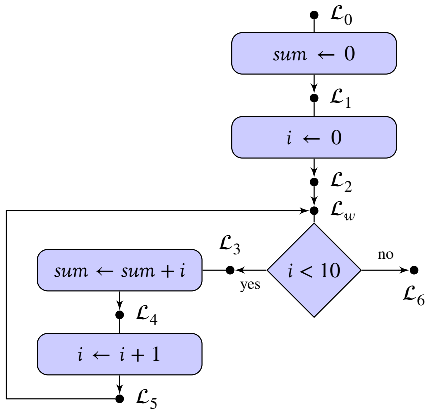
The syntax of the *Simple* language is presented in Fig. 2.1. Numerical expressions can use numeric constants from *Num* and variables from *Var*. Here we do not distinguish variable names from their mathematical symbols. Numerical expressions allowed are sum, difference, multiplication and change of sign (unary minus). *Simple* does not provide boolean variables, however boolean expressions exist. They are used in conditional instruction **if**, and the condition of the **while** loop. These conditions can be constructed by comparisons of numerical expressions extended by negation and binary disjunction, and conjunction operators. A program written in *Simple* is a block of instructions that is a list of instructions separated by a semicolon **;**. The instructions allowed by *Simple* are the empty instruction **skip**, an assignment, a conditional instruction **if-then-else**, and a **while-do** loop.

2.3.2 Instruction Labelling

In the abstract syntax of *Simple* we assume that all instructions are labelled in the following fashion:

```
program
    $\mathcal{L}_0$ : $sum \leftarrow 0$ ;
    $\mathcal{L}_1$ : $i \leftarrow 0$ ;
    $\mathcal{L}_2$ : while $\mathcal{L}_w$ : $i < 10$ do
        $\mathcal{L}_3$ : $sum \leftarrow sum + i$ ;
        $\mathcal{L}_4$ : $i \leftarrow i + 1$;
        $\mathcal{L}_5$
    od;
    $\mathcal{L}_6$
end
```

(a) Source code



(b) Control flow graph

Figure 2.2: An example of a program written in *Simple*: (a) program source code (b) control flow graph of the program with labels (black dots) and transitions between them (arrows).

- every instruction is labelled — there is a label before every instruction in the block and after the last instruction in the block,
- every **while** loop has an additional label, just before the loop test condition (the label is reached when entering the loop for the first time and after every loop iteration, just before executing the test).

An example of a program written in *Simple* language with instruction labelling is presented in Fig. 2.2(a). Every label that exists in a program is unique. We denote labels by letters $\mathcal{L}$ that might have some additional lower or upper index, e.g. $\mathcal{L}'$ or $\mathcal{L}_i$. For every program $\mathcal{P}$ we define the following functions:

- at_P : $\mathit{Stmt} \to \mathit{Lab}$ that returns the unique label that appears in a block just before the instruction in the argument,
- $after_P$: $\mathit{Stmt} \to \mathit{Lab}$ that returns the unique label that appears in a block just after the instruction in the argument,

where *Stmt* is the set of all occurrences of instructions in the program. These functions are easily defined by induction on the structure of the syntax of *Simple* [29, Lecture 5]. We present a few examples of applications of these functions for the program from Fig. 2.2:

- $at_P(sum \leftarrow sum + i) = \mathcal{L}_3$,

- $after_{\mathcal{P}}(sum \leftarrow sum + i) = \mathcal{L}_4$.

Labels that appear in a program $\mathcal{P}$ represent control points of a program.

2.3.3 Language Semantics

Let $\mathbb{I}$ be a selected numerical set, which is either integers $\mathbb{Z}$, rationals $\mathbb{Q}$ or reals $\mathbb{R}$. The concrete environment is the state of variables of the analysed program, i.e. a function:

$$s \in \mathit{State} = \mathit{Var} \to \mathbb{I}$$

that maps every variable to its actual value. In order to get the value of a variable $v \in \mathit{Var}$ in the state $s \in \mathit{State}$ we apply the function s to the variable v and denote such application by $s\,v$. We write $s' = s[u \mapsto c]$ to denote the update of the state s and assign value $c \in \mathbb{I}$ to the variable $u \in \mathit{Var}$. The updated state $s' \in \mathit{State}$ is such that for any $v \in \mathit{Var}$:

$$s'\,v = \begin{cases} s\,v & \text{if } v \neq u \\ u & \text{if } v = u. \end{cases}$$

Numerical constants that appear in the source of the program are mapped to their real values by the following semantic function:

$$\mathcal{N} : \mathit{Num} \to \mathbb{I}$$

that translates strings representing numerical values to real numerical values.

$$\begin{array}{lcl}
\mathcal{E}[\![v]\!]\,s & = & s\,v \\
\mathcal{E}[\![c]\!]\,s & = & \mathcal{N}[\![c]\!] \\
\mathcal{E}[\![-e]\!]\,s & = & -\,\mathcal{E}[\![e]\!]\,s \\
\mathcal{E}[\![e_1 + e_2]\!]\,s & = & \mathcal{E}[\![e_1]\!]\,s + \mathcal{E}[\![e_2]\!]\,s \\
\mathcal{E}[\![e_1 * e_2]\!]\,s & = & \mathcal{E}[\![e_1]\!]\,s * \mathcal{E}[\![e_2]\!]\,s \\
\mathcal{E}[\![e_1 - e_2]\!]\,s & = & \mathcal{E}[\![e_1]\!]\,s - \mathcal{E}[\![e_2]\!]\,s
\end{array}$$

where $v \in \mathit{Var}$, $c \in \mathit{Num}$, $e_1, e_2 \in \mathit{Exp}$ and $s \in \mathit{State}$.

Figure 2.3: Semantics of expressions for *Simple*.

Let Exp be the set of possible arithmetic expressions created with the syntax rules from Fig. 2.1. The meaning of an arithmetic expression is given by the following semantic function:

$$\mathcal{E} : \mathit{Exp} \to (\mathit{State} \to \mathbb{I}).$$

$$
\begin{array}{lcl}
\mathcal{B}\,[\![\textbf{not}\ b]\!]\ s & = & \begin{cases} \mathit{ff} & \text{if } \mathcal{B}\,[\![b]\!]\ s = \mathit{tt} \\ \mathit{tt} & \text{otherwise} \end{cases} \\
\mathcal{B}\,[\![b_1\ \textbf{and}\ b_2]\!]\ s & = & \begin{cases} \mathit{tt} & \text{if } \mathcal{B}\,[\![b_1]\!]\ s = \mathit{tt} \text{ and } \mathcal{B}\,[\![b_2]\!]\ s = \mathit{tt} \\ \mathit{ff} & \text{otherwise} \end{cases} \\
\mathcal{B}\,[\![b_1\ \textbf{or}\ b_2]\!]\ s & = & \begin{cases} \mathit{tt} & \text{if } \mathcal{B}\,[\![b_1]\!]\ s = \mathit{tt} \text{ or } \mathcal{B}\,[\![b_2]\!]\ s = \mathit{tt} \\ \mathit{ff} & \text{otherwise} \end{cases} \\
\mathcal{B}\,[\![e_1 = e_2]\!]\ s & = & \begin{cases} \mathit{tt} & \text{if } \mathcal{E}\,[\![e_1]\!]\ s = \mathcal{E}\,[\![e_2]\!]\ s \\ \mathit{ff} & \text{otherwise} \end{cases} \\
\mathcal{B}\,[\![e_1 < e_2]\!]\ s & = & \begin{cases} \mathit{tt} & \text{if } \mathcal{E}\,[\![e_1]\!]\ s < \mathcal{E}\,[\![e_2]\!]\ s \\ \mathit{ff} & \text{otherwise} \end{cases} \\
\mathcal{B}\,[\![e_1 \leq e_2]\!]\ s & = & \begin{cases} \mathit{tt} & \text{if } \mathcal{E}\,[\![e_1]\!]\ s \leq \mathcal{E}\,[\![e_2]\!]\ s \\ \mathit{ff} & \text{otherwise} \end{cases} \\
\mathcal{B}\,[\![e_1 \neq e_2]\!]\ s & = & \mathcal{B}\,[\![\textbf{not}\ e_1 = e_2]\!]\ s
\end{array}
$$

where $b, b_1, b_2 \in \mathit{BExp}$, $e_1, e_2 \in \mathit{Exp}$ and $s \in \mathit{State}$.

Figure 2.4: Semantics of test expressions for *Simple*.

The function is defined by induction on the syntax of expressions in Fig. 2.3.

We deal with boolean test expressions analogously to arithmetic expressions. Let *BExp* be the set of possible test expressions for *Simple*. The meaning of a test expression is given by the following semantic function:

$$\mathcal{B} : \mathit{BExp} \to (\mathit{State} \to \mathit{Bool}),$$

where $\mathit{Bool} = \{\mathit{tt}, \mathit{ff}\}$. The constants tt and ff represent real values of *true* and *false* respectively. The semantics of test expressions in *Simple* is defined by induction on the syntax of test expressions presented in Fig. 2.4.

The state element $s \in \mathit{State}$ records only the valuation of variables, without information about the current program point. We extend the definition by adding information about label $\mathcal{L} \in \mathit{Lab}$ that uniquely determines the current program point. An *execution state* for *Simple* is a pair defined as follows:

$$\langle \mathcal{L}, s \rangle \in \mathit{ExecState} = \mathit{Lab} \times \mathit{State}.$$

An execution of a program in *Simple* is a sequence of transitions between elements of *ExecState* starting from some initial state. These transitions describe a *control* *flow graph* — an example is presented in Fig. 2.2(b). Most of the transitions do not depend on the current state of the program. The ones that depend on the current state are conditional transitions (represented by a diamond-shaped nodes), which appear both in conditional **if-then-else** statements and **while** loops.

We introduce a small-step operational semantics [86] for instructions and programs written in *Simple*. Operational semantics is a transition system that is a pair $\tau = \langle \Sigma, t \rangle$, where:

- Σ is the set of states,
- t is the relation that defines transitions between states from the set Σ: $t \subseteq \Sigma \times \Sigma$ — we write $s \rightsquigarrow_t s'$ if and only if $\langle s, s' \rangle \in t$.

For the language *Simple* it holds that $\Sigma = \mathit{ExecState}$ and the transition relation is deterministic, i.e. for every execution state $s \in \Sigma$ there is at most one state $s' \in \Sigma$ such that $s \rightsquigarrow_t s'$.

A transition rule for the *Simple* language has the following form:

$$\frac{A_1 \quad A_2 \quad \dots \quad A_n}{\langle \mathcal{L}, s \rangle \rightsquigarrow_t \langle \mathcal{L}', s' \rangle}$$

for some $n \geq 1$. The meaning is that the transition below the horizontal line exists if and only if all the conditions above the line, i.e. $A_1, \dots, A_n$, hold.

$$\frac{C = \mathbf{skip} \quad \mathit{at}_P(C) = \mathcal{L} \quad \mathit{after}_P(C) = \mathcal{L}'}{\langle \mathcal{L}, s \rangle \rightsquigarrow_t \langle \mathcal{L}', s \rangle}$$

$$\frac{C = x \leftarrow e \quad \mathit{at}_P(C) = \mathcal{L} \quad \mathit{after}_P(C) = \mathcal{L}' \quad \mathcal{E}[\![\mathrm{e}]\!]\, s = v}{\langle \mathcal{L}, s \rangle \rightsquigarrow_t \langle \mathcal{L}', s[x \mapsto v] \rangle}$$

Figure 2.5: Transitions of **skip** and assignment instructions for *Simple*.

Two basic transitions for the *Simple* language are presented in Fig. 2.5. The transition for **skip** instruction is trivial: it just moves to the label after the instruction without any modification of the input state. The assignment instruction first evaluates the assignment expression and then transfers to a state, in which the modified variable has the computed value.

Next, we deal with transitions for the **if-then-else** instruction — presented in Fig. 2.6. The first two transitions depend on the result of the evaluation of the boolean condition. If the condition evaluates to *true*, there is a transition that leads

$$\frac{C = \textbf{if}\ b\ \textbf{then}\ \mathcal{B}_t\ \textbf{else}\ \mathcal{B}_f\ \textbf{fi} \quad at_P(C) = \mathcal{L} \quad at_P(\mathcal{B}_t) = \mathcal{L}' \quad \mathcal{B}[\![\mathrm{b}]\!]\, s = tt}{\langle \mathcal{L}, s\rangle \rightsquigarrow_t \langle \mathcal{L}', s\rangle}$$

$$\frac{C = \textbf{if}\ b\ \textbf{then}\ \mathcal{B}_t\ \textbf{else}\ \mathcal{B}_f\ \textbf{fi} \quad at_P(C) = \mathcal{L} \quad at_P(\mathcal{B}_f) = \mathcal{L}' \quad \mathcal{B}[\![\mathrm{b}]\!]\, s = ff}{\langle \mathcal{L}, s\rangle \rightsquigarrow_t \langle \mathcal{L}', s\rangle}$$

$$\frac{C = \textbf{if}\ b\ \textbf{then}\ \mathcal{B}_t\ \textbf{else}\ \mathcal{B}_f\ \textbf{fi} \quad after_P(\mathcal{B}_f) = \mathcal{L} \quad after_P(C) = \mathcal{L}'}{\langle \mathcal{L}, s\rangle \rightsquigarrow_t \langle \mathcal{L}', s\rangle}$$

$$\frac{C = \textbf{if}\ b\ \textbf{then}\ \mathcal{B}_t\ \textbf{else}\ \mathcal{B}_f\ \textbf{fi} \quad after_P(\mathcal{B}_t) = \mathcal{L} \quad after_P(C) = \mathcal{L}'}{\langle \mathcal{L}, s\rangle \rightsquigarrow_t \langle \mathcal{L}', s\rangle}$$

Figure 2.6: Transitions for the **if-then-else** instruction.

to the label just before the **true** block. Otherwise, there is a transition that leads to the label just before the **false** block. Both transitions are performed without any modifications to the input state. The next two transitions connect labels after **then** block and **else** block with the label that is after the **if-then-else** instruction.

$$\frac{C = \textbf{while}\ \mathcal{L}_w : b\ \textbf{do}\ \mathcal{B}\ \textbf{od} \quad at_P(\mathcal{B}) = \mathcal{L} \quad \mathcal{B}[\![\mathrm{b}]\!]\, s = tt}{\langle \mathcal{L}_w, s\rangle \rightsquigarrow_t \langle \mathcal{L}, s\rangle}$$

$$\frac{C = \textbf{while}\ \mathcal{L}_w : b\ \textbf{do}\ \mathcal{B}\ \textbf{od} \quad after_P(C) = \mathcal{L} \quad \mathcal{B}[\![\mathrm{b}]\!]\, s = ff}{\langle \mathcal{L}_w, s\rangle \rightsquigarrow_t \langle \mathcal{L}, s\rangle}$$

$$\frac{C = \textbf{while}\ \mathcal{L}_w : b\ \textbf{do}\ \mathcal{B}\ \textbf{od} \quad after_P(\mathcal{B}) = \mathcal{L}}{\langle \mathcal{L}, s\rangle \rightsquigarrow_t \langle \mathcal{L}_w, s\rangle}$$

$$\frac{C = \textbf{while}\ \mathcal{L}_w : b\ \textbf{do}\ \mathcal{B}\ \textbf{od} \quad at_P(C) = \mathcal{L}}{\langle \mathcal{L}, s\rangle \rightsquigarrow_t \langle \mathcal{L}_w, s\rangle}$$

Figure 2.7: Transitions for **while** loop.

Now we deal with transitions for the **while** loop — presented in Fig. 2.7. The first two transitions depend on the evaluation of the loop condition. If the condition evaluates to *true*, there is a transition from the special *loop label*, $\mathcal{L}_w$ to the label before the loop body. Otherwise, there is a transition from the *loop label* to the label just after the loop. The third transition makes the actual loop in the control flow graph: it connects the label after the loop body with the *loop label*. The last transition connects the label before the loop with the *loop label*. All transitions mentioned for the **while** loop do not modify input states.

2.4 Abstract Interpretation Primer

In most cases, the verification of computer software based on concrete program semantics is infeasible. However, the precise semantics usually contains much more information than a verifier needs — the verification process is mainly used to search for specific types of errors such as *index out of bounds error*, *integer overflow* or *null pointer exceptions*. Based on these observations, *abstract interpretation* [28, 32, 39] was created. The idea is to use an abstraction of the real semantics and focus only on interesting properties of a program. The main assumption of the *abstract interpretation* is the soundness of the analysis: if there is no error found in the abstract world, there is no error in the real one. Since the abstraction deals with only a subset of properties, some of the dependencies that exist in the real world are not detected. Therefore, the abstract analysis may yield *false positives* — it may find a possible error that actually does not occur in the program.

Abstract interpretation is a general theory for designing approximate semantics of computer programs. It is used to gather information about programs without executing them in order to provide answers to questions about their run-time behaviour. The essential property of the approximate semantics designed with abstract interpretation is its soundness. The idea is to focus on semantic domains and relationships between them. We deal with two levels of semantics:

- *concrete semantics* — an ordering $\langle \mathcal{D}, \leq \rangle$, it is a domain of concrete or exact properties (elements of $\mathcal{D}$),
- *abstract semantics* — an ordering $\langle \mathcal{D}^\sharp, \leq^\sharp \rangle$, which is an approximation of the concrete one, therefore it deals with abstract or approximate properties.

The most concrete semantics of a program is the *standard semantics*. Usually, it is considered as a set of program states that are reachable from any of the input states, where the transition between states can be derived from a single-step semantics of the program.

2.4.1 Soundness

The crucial part of the definition of the abstraction is the correspondence between the concrete and abstract semantics — it is responsible for the soundness of the abstract semantics. The meaning of abstract properties is defined by a *soundness relation* [34] $\sigma \subseteq \mathcal{D} \times \mathcal{D}^\sharp$ such that $\sigma(x, x^\sharp)$ holds if and only if $x^\sharp \in \mathcal{D}^\sharp$ is a *sound abstraction* of $x \in \mathcal{D}$, i.e. the concrete property x enjoys the abstract property $x^\sharp$.

The goal of the abstract interpretation is to find an abstract property, if any, that is a correct approximation of the concrete element: for $x \in \mathcal{D}$ find $x^\sharp \in \mathcal{D}^\sharp$ such that $\sigma(x, x^\sharp)$. Because concrete and abstract semantics are orderings, both $\leq$ and $\leq^\sharp$ must be compatible with the soundness relation:

- if $x^\sharp$ is a sound abstraction of x, then also $y^\sharp$, such that $x^\sharp \leq^\sharp y^\sharp$, is a sound abstraction of x, however less precise:
$$\forall x \in \mathcal{D} : \forall x^\sharp, y^\sharp \in \mathcal{D}^\sharp : (\sigma(x, x^\sharp) \land x^\sharp \leq^\sharp y^\sharp) \Rightarrow \sigma(x, y^\sharp),$$
- dually, if $x^\sharp$ is a sound abstraction of x, then it is also a sound abstraction of y such that $y \leq x$:
$$\forall x^\sharp \in \mathcal{D}^\sharp : \forall x, y \in \mathcal{D} : (\sigma(x, x^\sharp) \land y \leq x) \Rightarrow \sigma(y, x^\sharp).$$

Various methods of constructing the soundness relation σ exist [34]. A classical, though restrictive approach [32] requires existence of *Galois connection*. Different constructions try to relax the requirement.

Abstraction Based on Galois Connection

A *Galois connection* [32] is a particular correspondence between two partially ordered sets, defined as follows:

Definition 2.4.1. A *Galois connection* between two partially ordered sets $\langle \mathcal{X}, \leq_{\mathcal{X}} \rangle$ and $\langle \mathcal{Y}, \leq_{\mathcal{Y}} \rangle$ is a pair of functions $\langle \alpha, \gamma \rangle$, where $\alpha : \mathcal{X} \to \mathcal{Y}$, $\gamma : \mathcal{Y} \to \mathcal{X}$ such that:
$$\forall x \in \mathcal{X} : \forall y \in \mathcal{Y} : \alpha(x) \leq_{\mathcal{Y}} y \iff x \leq_{\mathcal{X}} \gamma(y),$$
and denoted:
$$\langle \mathcal{X}, \leq_{\mathcal{X}} \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{Y}, \leq_{\mathcal{Y}} \rangle.$$

A *Galois connection* is often used to describe a connection between concrete and abstract domains. Let $\langle \mathcal{D}, \leq \rangle$ be a concrete domain, $\langle \mathcal{D}^\sharp, \leq^\sharp \rangle$ be an abstract domain and there is a *Galois connection* between these domains:
$$\langle \mathcal{D}, \leq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{D}^\sharp, \leq^\sharp \rangle.$$

We say that $d^\sharp \in \mathcal{D}^\sharp$ is a *sound abstraction* (or a *sound approximation*) of $d \in \mathcal{D}$ if $\alpha(d) \leq^\sharp d^\sharp$, or equivalently, if $d \leq \gamma(d^\sharp)$. The function $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ that maps an abstract element $d^\sharp$ to the greatest concrete element that satisfies the property $d^\sharp$ is called a *concretisation function*. The function $\alpha : \mathcal{D} \to \mathcal{D}^\sharp$ that maps a concrete element d to the strongest property (the least abstract element in $\leq^\sharp$) that is satisfied for d is called an *abstraction function*.

Galois connection has some interesting properties [33, Sec. 4.2]. Here we recall some of them:

- the function $\gamma \circ \alpha$ is extensive (or inflationary):

  $$\forall x \in \mathcal{X} : x \preceq_{\mathcal{X}} \gamma(\alpha(x)),$$

  which means that the loss of the information in the abstraction process is sound;

- the function $\gamma \circ \alpha$ is an *upper closure operator*, that is monotone, extensive (see previous point) and idempotent;

- the function $\alpha \circ \gamma$ is reductive:

  $$\forall y \in \mathcal{Y} : \alpha(\gamma(y)) \preceq_{\mathcal{Y}} y,$$

  which means that the concretisation does not introduce any loss of information;

- the function $\alpha \circ \gamma$ is *lower closure operator*, that is monotone, reductive (see previous point) and idempotent;

- in *Galois connection* one function uniquely determines the other, that is if $\langle \mathcal{X}, \preceq_{\mathcal{X}} \rangle \overset{\gamma_1}{\underset{\alpha_1}{\leftrightarrows}} \langle \mathcal{Y}, \preceq_{\mathcal{Y}} \rangle$ and $\langle \mathcal{X}, \preceq_{\mathcal{X}} \rangle \overset{\gamma_2}{\underset{\alpha_2}{\leftrightarrows}} \langle \mathcal{Y}, \preceq_{\mathcal{Y}} \rangle$ then it holds that $\alpha_1 = \alpha_2$ if and only if $\gamma_1 = \gamma_2$;

- a composition of *Galois connections* is a *Galois connection*:

  $$\left(\langle \mathcal{X}, \preceq_{\mathcal{X}} \rangle \overset{\gamma_1}{\underset{\alpha_1}{\leftrightarrows}} \langle \mathcal{Y}, \preceq_{\mathcal{Y}} \rangle \wedge \langle \mathcal{Y}, \preceq_{\mathcal{Y}} \rangle \overset{\gamma_2}{\underset{\alpha_2}{\leftrightarrows}} \langle \mathcal{Z}, \preceq_{\mathcal{Z}} \rangle \right) \Rightarrow \langle \mathcal{X}, \preceq_{\mathcal{X}} \rangle \overset{\gamma_1 \circ \gamma_2}{\underset{\alpha_2 \circ \alpha_1}{\leftrightarrows}} \langle \mathcal{Z}, \preceq_{\mathcal{Z}} \rangle.$$

The last property is a basis for designing analysers by composition of successive approximations.

Abstraction Based on Concretisation Function

The existence of a Galois connection is sometimes a too strong requirement. In fact, any of the two functions from Definition 2.4.1 may not exist, e.g. for the *domain of polyhedra* the best abstraction for a concrete element does not always exist [34, Example 6.3]. The abstraction function often does not exist when the abstract domain is not a complete lattice. The *abstract interpretation* framework can still work in such situation. We assume that there exists a monotone concretisation function $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$. We say that $d^\sharp \in \mathcal{D}^\sharp$ is an *abstraction* of $d \in \mathcal{D}$ if $d \leq \gamma(d^\sharp)$. In the thesis we assume the existence of the concretisation function.

2.4.2 Transformer Abstraction

A *transformer* is an operator on a domain, i.e. for a domain $\langle \mathcal{D}, \leq \rangle$ it is any function $F : \mathcal{D} \to \mathcal{D}$. An *abstract transformer* is a transformer operator for the abstract domain. First, we introduce a definition of a transformer abstraction in the situation when there is a Galois connection between two domains (see [39, Theorem 7.1.0.2]):

Definition 2.4.2 (Sound Transformer Abstraction Using Galois Connection). Let $\langle \mathcal{D}, \leq \rangle$ be the concrete domain, $\langle \mathcal{D}^\sharp, \leq \rangle$ be the abstract domain and $\langle \mathcal{D}, \leq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{D}^\sharp, \leq \rangle$ be a Galois connection between these domains. The abstract transformer operator $F^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ is a *sound abstraction* of the concrete transformer $F : \mathcal{D} \to \mathcal{D}$ if and only if for any $d^\sharp \in \mathcal{D}^\sharp$ it holds that:

$$\alpha \circ F \circ \gamma(d^\sharp) \leq^\sharp F^\sharp(d^\sharp).$$

The idea of a transformer abstraction using *Galois connection* is illustrated in Fig. 2.8(a). The abstract operator $F^\sharp$ is the *best approximation* if $F^\sharp = \alpha \circ F \circ \gamma$.

$$\begin{tikzcd}
d_1^\sharp \arrow[rr, bend left, squiggly, "F^\sharp"] \arrow[d, "\gamma"] & & d_2'^\sharp \arrow[r, phantom, "\leq^\sharp"] & d_2^\sharp \\
d_1 \arrow[rr, squiggly, "F"] & & d_2 \arrow[u, "\alpha"']
\end{tikzcd}$$

(a) Galois connection

$$\begin{tikzcd}
d_1^\sharp \arrow[rr, squiggly, "F^\sharp"] \arrow[d, "\gamma"] & & d_2^\sharp \arrow[d, "\gamma"] \\
d_1 \arrow[r, squiggly, "F"] & d_2 \arrow[r, phantom, "\leq"] & d_2'
\end{tikzcd}$$

(b) Concretisation only

Figure 2.8: Sound transformer abstraction.

An alternative definition of the transformer abstraction using only a concretisation function is as follows:

Definition 2.4.3 (Sound Transformer Abstraction Using Concretisation). Let $\langle \mathcal{D}, \leq \rangle$ be the concrete domain, $\langle \mathcal{D}^\sharp, \leq \rangle$ be the abstract domain and $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ be the concretisation function. The abstract transformer operator $F^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ is called a *sound approximation* of the concrete transformer $F : \mathcal{D} \to \mathcal{D}$ if and only if for any $d^\sharp \in \mathcal{D}^\sharp$ it holds that:

$$F \circ \gamma(d^\sharp) \leq \gamma \circ F^\sharp(d^\sharp)$$

The idea of a transformer abstraction using the *concretisation function* only is illustrated in Fig. 2.8(b). When $\gamma \circ F^\sharp = F \circ \gamma$ the operator is an *exact abstraction* of F. Such situation usually does not appear, since the result of the concrete operator F may not be exactly representable in the abstract domain. When there is a Galois connection both definitions are equivalent (simple proof using properties of Galois connections).

2.4.3 Fixpoint Transfer

In this section we describe dependencies between fixpoints of the concrete and the abstract semantics. Consider a situation when one is able to compute the abstract semantics of a program (the least fixpoint of the semantic function). Then, if assumptions of a *fixpoint transfer theorem* such as Theorem 2.4.4 are fulfilled, the computed result is a sound approximation of the concrete semantics.

Theorem 2.4.4 (Fixpoint Transfer Theorem). *Let* $\langle \mathcal{D}, \leq \rangle$ *and* $\langle \mathcal{D}^\sharp, \leq^\sharp \rangle$ *be both complete partial orders and there is a Galois connection* $\langle \mathcal{D}, \leq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{D}^\sharp, \leq^\sharp \rangle$. *Let* $f : \mathcal{D} \to \mathcal{D}$ *and* $f^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ *be monotone functions such that:*

$$\forall d \in \mathcal{D} : d \leq \mathit{lfp}\left(f\right) \Rightarrow \alpha \circ f(d) = f^\sharp \circ \alpha(d),$$

then

$$\alpha(\mathit{lfp}\left(f\right)) = \mathit{lfp}\left(f^\sharp\right).$$

Proof. See [31, Theorem 2]. □

There are various variants of the *fixpoint transfer theorem* [33, 41], but the main problem is that the computation of a fixpoint of the abstract semantics may be infeasible. If the programming language chosen for the analysis contains loops, the computation of the fixpoint is not trivial. If the abstract domain does not contain any infinite chain of elements that is strictly increasing and the semantic function is monotone, one could simply apply Theorem 2.2.9 and compute Kleene's iteration sequence. Otherwise, if the abstract domain contains strictly increasing infinite chains, the termination of such iteration usually is not guaranteed. In such situations a *fixpoint transfer theorem* is not sufficient. Therefore, *abstract interpretation* introduces a procedure to compute an over-approximation of a fixpoint — it is presented in the next section.

2.4.4 Widening Operator

Since in most cases the computation of the fixpoint with the Kleene iteration is not feasible, a *widening operator* was introduced [38]. It is a form of convergence accelerator for Kleene iterations. The *widening operator* is defined as follows:

Definition 2.4.5 (Widening operator). Let $\langle \mathcal{D}, \leq \rangle$ be a partially ordered set. A binary operator $\triangledown : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is a *widening operator* if and only if the following properties hold:

- *over-approximation* — for all $d_1, d_2 \in \mathcal{D}$ it holds that:
$$d_1 \leq d_1 \triangledown d_2 \text{ and } d_2 \leq d_1 \triangledown d_2,$$
- *stabilisation* — for every increasing chain in $\mathcal{D}$: $d_0 \leq d_1 \leq d_2 \leq \ldots$ the increasing chain defined by:
$$y_0 = d_0,$$
$$y_{n+1} = y_n \triangledown d_{n+1} \text{ for } n \geq 0,$$
is not strictly increasing, i.e. there exists $i \in \mathbb{N}$ such that $y_i = y_{i+1}$.

A *widening operator* is used to compute an over-approximation of a fixpoint — we may receive something more, but the result is sound and computable. Theorem 2.4.6 states that the abstract iteration result is a sound approximation of the concrete fixpoint.

Theorem 2.4.6 (Abstract Iteration Sequence with Widening). *Let $\langle \mathcal{D}, \leq \rangle$ be a complete lattice, function $f : \mathcal{D} \to \mathcal{D}$ be monotone, $\langle \mathcal{D}^\sharp, \leq^\sharp \rangle$ be a complete partial order, and a concretisation function $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ be monotone. Let a function $f^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ be an abstraction of f, i.e. $f \circ \gamma \leq \gamma \circ f^\sharp$. Let $x \in \mathcal{D}$, $x^\sharp \in \mathcal{D}^\sharp$ be such that $x \leq \gamma(x^\sharp)$. Then in a chain defined by:*

$$y_0^\sharp = x^\sharp,$$
$$y_{n+1}^\sharp = y_n^\sharp \triangledown f(y_n^\sharp) \, \textit{for } n \geq 0,$$

there exists $n \geq 0$ *such that* $f^\sharp({y_n}^\sharp) \leq^\sharp {y_n}^\sharp$ *and:*

$$lfp_x(f) \leq \gamma({y_n}^\sharp).$$

Proof. See [84, Theorem 2.2.7]. □

Usually, in order to compute the abstraction, the iteration starts from some abstract element $x^\sharp \in \mathcal{D}^\sharp$ such that $\gamma(x^\sharp) = \bot$, where $\bot$ is the least element in $\mathcal{D}$. Note that the abstract transformer $f^\sharp$ does not need to be monotone.

In some cases one may require a little different definition of the widening operator, where it is a partial operator defined when the second argument is greater or equal to the first one [10, 42]:

Definition 2.4.7 (Widening operator variant). Let $\langle \mathcal{D}, \leq, \bot, \cup \rangle$ be an upper semilattice. A *widening operator* is a partial operator $\triangledown : \mathcal{D} \times \mathcal{D} \rightharpoonup \mathcal{D}$ if and only if the following properties hold:

- *over-approximation* — for every $d_1, d_2 \in \mathcal{D}$, $d_1 \leq d_2$ implies that $d_1 \triangledown d_2$ is defined and $d_2 \leq d_1 \triangledown d_2$,
- *stabilisation* — for every increasing chain $d_0 \leq d_1 \leq d_2 \leq \dots$ the increasing chain defined by:

$$\begin{aligned} y_0 &= d_0, \\ y_{n+1} &= y_n \triangledown (y_n \cup d_{n+1}) \text{ for } n \geq 0, \end{aligned}$$

  is not strictly increasing, i.e. there exists $i \in \mathbb{N}$ such that $y_i = y_{i+1}$.

In the definition of the new widening operator introduced in the thesis we use the variant of the widening operator presented in Definition 2.4.7.

2.4.5 Collecting Semantics

While *standard semantics* describes behaviour of programs during their executions, the *collecting semantics* focuses on some class of properties of these executions. It collects information about program executions and defines the strongest static property of interest (it is also called *static semantics*). The choice of the *collecting semantics* depends on the properties one would like to analyse. In this section we present a few most popular ones.

Let $\tau = \langle \Sigma, t \rangle$ be a transition system for a program P, where Σ is the set of states and $t \in \mathcal{P}(\Sigma \times \Sigma)$ is the transition relation between states from Σ. The notation is the same as for the language *Simple* from Section 2.3.

Partial Trace Semantics

The basic version of the collecting semantics is a *partial trace semantics*, which collects information about all possible program executions. A finite trace $\sigma \in \Sigma^*$ of a program is a finite sequence of program states, which have transitions between them, that is $\sigma = \langle s_0, s_1, \dots, s_n \rangle$ for some $n \geq 0$, where for all $i \in \{0, \dots, n\}$ it holds that $s_i \in \Sigma$ and the sequence is created by transitions from t:

$$\forall i \in \{0, \dots, n-1\} \, : \, s_i \rightsquigarrow_t s_{i+1}.$$

We use a shorter notation to denote sequences from Σ: we write $s_0 s_1 \dots s_n$ for a sequence $\sigma = \langle s_0, s_1, \dots, s_n \rangle$ and $\sigma s'$ for a sequence $\langle s_0, s_1, \dots s_n, s' \rangle$.

Let Σ_t^n denote all the traces of length exactly $n \geq 0$ of a program P and a transition system τ. Then we have:

$$\begin{aligned} \Sigma_t^0 &= \varnothing, \\ \Sigma_t^1 &= \{\langle s \rangle \mid s \in \Sigma\}, \\ \Sigma_t^{n+1} &= \{\sigma s s' \mid \sigma s \in \Sigma_t^n \land s \rightsquigarrow_t s'\} \text{ for } n > 1. \end{aligned}$$

A *partial trace semantics*, denoted Σ_t^*, is a set of all the traces of finite length and is defined as:

$$\Sigma_t^* = \bigcup_{n \geq 0} \Sigma_t^n.$$

Theorem 2.4.8 states that the semantics can be defined as a fixpoint.

Theorem 2.4.8. *The* collecting semantics of partial traces *can be expressed in terms of a fixpoint, that is:*

$$\Sigma_t^* = lfp\left(\mathcal{F}_t^*\right) = \bigcup_{n \geq 0} \mathcal{F}_t^{*n}(\varnothing),$$

where function $\mathcal{F}_t^* : \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ is defined as follows:

$$\mathcal{F}_t^*(\mathcal{X}) = \{\langle s \rangle \mid s \in \Sigma\} \cup \{\sigma s s' \mid \sigma s \in \mathcal{X} \land s \rightsquigarrow_t s'\}.$$

Proof. Can be found in [35, Sec. 4]. □

Transitive Closure Semantics

The *partial traces semantics* is very strong, since it handles all possible execution traces of a program, but one usually does not require to reason about the whole history of an execution. The *transitive closure semantics* abandons most of the information about a trace, it focuses only on the first and the last state in every trace instead. It forgets intermediate states, therefore is as abstraction of the *partial traces semantics*. The fact is stated by the abstraction function $\alpha^* : \mathscr{P}(\Sigma^*) \to \mathscr{P}(\Sigma \times \Sigma)$:

$$\alpha^*(\mathcal{X}) = \{\alpha^{\to}(\sigma) \mid \sigma \in \mathcal{X}\},$$

where $\alpha^{\to}(s_0 s_1 \dots s_k) = \langle s_0, s_k \rangle$ for $k \geq 0$. The *transitive closure semantics* is a set of pairs: $t^* \in \mathscr{P}(\Sigma \times \Sigma)$ and is defined as:

$$t^* = \alpha^*(\Sigma_t^*).$$

Note that t^* is the reflexive transitive closure of the t relation (hence the name). Since we already have the abstraction function α^*, we introduce a concretisation function $\gamma^* : \mathscr{P}(\Sigma \times \Sigma) \to \mathscr{P}(\Sigma^*)$, which for a set of pairs of states returns all possible finite traces that start at the first element of pair and finish at the second element. Formally, the function γ^* is defined as follows:

$$\gamma^*(\mathcal{Y}) = \{\sigma \mid \alpha^{\to}(\sigma) \in \mathcal{Y}\}.$$

Note that for a set of partial traces $\mathcal{X}$, it holds that $\mathcal{X} \subseteq \gamma^*(\alpha^*(\mathcal{X})))$. Additionally, the pair of functions $\langle \alpha^*, \gamma^* \rangle$ forms a Galois connection:

Theorem 2.4.9. *Functions α^* and γ^* form a Galois connection.*

Proof. Can be found in [35, Sec. 7]. □

As it happened with *partial trace semantics*, the *transitive closure semantics* can be expressed in a fixpoint form [35, Sec. 8]. The fact is a consequence of a fixpoint transfer theorem.

Reachability Semantics

Let $\Sigma_i \in \Sigma$ be a set of initial states for the transition system $\tau = \langle \Sigma, t \rangle$. The *reachability semantics* $\mathcal{R}_t \in \Sigma$ is a set of states that are reachable from any initial state $s \in \Sigma_i$ and is defined as:

$$\mathcal{R}_t = \{s' \mid \exists s \in \Sigma_i : \langle s, s' \rangle \in t^*\}.$$

This is an abstraction of the transitive closure semantics (therefore also partial trace semantics). The fact is characterised by the following pair of functions: an abstraction $\alpha^\circ : \mathscr{P}(\Sigma \times \Sigma) \to \mathscr{P}(\Sigma)$ and a concretisation $\gamma^\circ : \mathscr{P}(\Sigma) \to \mathscr{P}(\Sigma \times \Sigma)$ that are defined as follows:

$$\alpha^\circ(\mathcal{Y}) = \{s' \mid \exists s \in \Sigma_i : \langle s, s' \rangle \in \mathcal{Y}\},$$
$$\gamma^\circ(\mathcal{Z}) = \{\langle s, s' \rangle \mid s \in \Sigma_i => s' \in \mathcal{Z}\}.$$

The definition of α° is quite natural — the result is created from elements that appear as second items of pairs in the argument and are reachable from any of the initial states. The function γ° returns all pairs of states between any of the initial states and any element in the argument.

As it happened in case of the transitive closure semantics, this time the pair of abstraction and concretisation functions for the *reachability semantics* form a Galois connection:

Theorem 2.4.10. *Functions α° and γ° form a Galois connection.*

Proof. Can be found in [35, Sec. 9]. □

The *reachability semantics* also can be expressed in terms of a fixpoint [35, Sec. 10].

Hierarchy of Collecting Semantics

The three semantics that were presented form a hierarchy ordered by an abstraction order:

$$\langle \Sigma_t^*, \subseteq \rangle \underset{\alpha^*}{\overset{\gamma^*}{\leftrightarrows}} \langle t^*, \subseteq \rangle \underset{\alpha^{\circ}}{\overset{\gamma^{\circ}}{\leftrightarrows}} \langle \mathcal{R}_t, \subseteq \rangle,$$

where the one on the left is the most abstract and the one on the right is the least abstract. Since the composition of Galois connections is a Galois connection, we obtain:

$$\langle \Sigma_t^*, \subseteq \rangle \underset{\alpha^{\circ}\circ\alpha^*}{\overset{\gamma^*\circ\gamma^{\circ}}{\leftrightarrows}} \langle \mathcal{R}_t, \subseteq \rangle.$$

The Choice of the Collecting Semantics

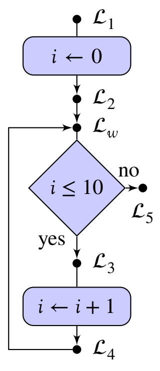
States in the *Simple* language are pairs $\langle \mathcal{L}, s \rangle \in \mathit{Lab} \times \mathit{State}$. The *reachability semantics* for *Simple*, instead of a subset of $\mathit{Lab} \times \mathit{State}$, can be defined as a function $f_{\mathcal{R}_t} : \mathit{Lab} \to \mathcal{P}(\mathit{State})$ as follows:

$$f_{\mathcal{R}_t}(\mathcal{L}) = \{ s \mid \langle \mathcal{L}, s \rangle \in \mathcal{R}_t \}.$$

The function $f_{\mathcal{R}_t}$ simply assigns, to every control point of a program, a set of states (valuations of variables) that appear at that point during the execution. Both definitions are equivalent and the function $f_{\mathcal{R}_t}$ is the classical variant of the definition for the concrete semantics [32]. For the rest of the thesis we choose this definition of the *reachability semantics* to be our collecting (concrete) semantics.

```
program
    $\mathcal{L}_1$ : $i \leftarrow 0$ ;
    $\mathcal{L}_2$ : while $\mathcal{L}_w$ : $i \leq 10$ do
        $\mathcal{L}_3$ : $i \leftarrow i + 1$;
        $\mathcal{L}_4$
    od;
    $\mathcal{L}_5$
end
```

(a) Source code



(b) Control flow graph

$$\begin{aligned} d_1 &= \bot \\ d_2 &= [0,0] \\ d_w &= d_2 \cup d_4 \\ d_3 &= d_2 \cap [-\infty, 10] \\ d_4 &= d_3 + [1,1] \\ d_5 &= d_w \cap [11, +\infty] \end{aligned}$$

(c) Equation system

Figure 2.9: Semantic equations for an example program written in *Simple*: (a) program source code, (b) control flow graph, (c) equation system for the program.

2.4.6 Abstract Interpreter

A computer program can be represented by a control flow graph $\mathcal{G} = (V, E)$. The set of vertices is a finite set of program control points (labels): $V = \mathit{Lab} = \{\mathcal{L}_1, \dots, \mathcal{L}_n\}$. The set of edges $E \subseteq \mathit{Lab} \times \mathit{Lab}$ is the set of connections between control points. We assume that the concrete domain is a pointwise lifting $\mathit{Lab} \to \mathcal{D}$, where $\langle \mathcal{D}, \subseteq \rangle$ is a complete lattice of concrete properties. Every edge in the graph $\mathcal{G}$ represents a transition in the program. The semantics of the program can be easily transformed to a set of equations [32]:

$$\begin{cases} d_1 &= F_1(d_1, d_2, \dots, d_n) \\ \dots \\ d_n &= F_n(d_1, d_2, \dots, d_n) \end{cases}$$

where for every $i \in \{1, \dots, n\}$ it holds that $d_i \in \mathcal{D}$ and $F_i : \mathcal{D}^n \to \mathcal{D}$ is monotone. The equation at index i corresponds to the label $\mathcal{L}_i$. The function F_i computes the property that holds at the point $\mathcal{L}_i$ of the program after one program step executed from any point leading to $\mathcal{L}_i$. It describes how the property at label $\mathcal{L}_i$ depends on its predecessors in graph $\mathcal{G}$. It holds that the function F_i depends on the parameter d_j if there is an edge between $\mathcal{L}_j$ and $\mathcal{L}_i$ in the graph $\mathcal{G}$, i.e. $\langle \mathcal{L}_j, \mathcal{L}_i \rangle \in E$. An example of such equation system for a one-dimensional version of the domain of intervals is presented in Fig. 2.9(c).

The semantics of the program is the least solution of the equation system. An algorithm for solving the equation system is called *chaotic iteration strategy*. An example of such algorithm is to iterate a parallel execution of all equations, analogically to Kleene's iteration sequence. But as expected, such solution is not guaranteed to terminate.

The technique called *chaotic iteration strategy with widening* extends the solution from Theorem 2.4.6 to a system of semantic equations. Let the abstract domain be also a pointwise lifting $\mathit{Lab} \to D^\sharp$. Then for every function F_i there is an abstract version $F_i^\sharp : D^{\sharp n} \to D^\sharp$ that is a sound abstraction of F_i. The goal of an *abstract interpreter* is to solve the equation system in the abstract world. The idea [32, Sec. 9.1.3] is to choose a subset of program points $W \subseteq \mathit{Lab}$ and for every label $\mathcal{L}_i \in W$ replace the i-th equation by:

$$d_i{}^\sharp = d_i{}^\sharp \triangledown F_i{}^\sharp(d_1{}^\sharp, \ldots, d_n{}^\sharp).$$

When the set W is chosen in a way that every cycle in the equation dependency graph contains at least one element from W [32], the computation is guaranteed to terminate and stabilise on a sound approximation of the solution of the concrete equation.

Theorem 2.4.11 (Chaotic Iteration Sequence with Widening). *The* chaotic iteration sequence with widening *stabilises in finite number of steps on a sound approximation of the least solution of the equation system.*

Proof. See [84, Theorem. 2.2.9]. □

The set of widening points W may be chosen in various ways [16]. For a program written in the *Simple* language, we choose as the set W the set of all labels that appear just before the loop test (special labels that were added to **while** loops, see Section 2.3.2).

2.5 Java Modeling Language

The current section is an introduction to the *Java Modeling Language* (JML) that is a specification language for Java that we use in the thesis. The structure of the majority of specifications languages, such as JML [74] for Java or Microsoft Code Contracts for the .NET platform [4], follows the structure of programming languages. The Java Modeling Language is a *behavioural interface specification language* for Java programs [73]. In such specification technique one specifies the interface of a method or abstract data type and the behavior of this method. With JML the interface specification does not require any additional format, JML uses regular Java method descriptors for this purpose. JML specification for a specific class or interface can be placed either inside the source code file or in some external specification file. In the second case one has to duplicate fragments of Java code such as class or method descriptors. An example of an external JML annotation of the `java.util.Dictionary` abstract class from the standard Java library is presented in Fig. 2.10.

```
package java.util;
public abstract class Dictionary<K,V> {
  //@ pure
  public Dictionary();

  //@ modifies \nothing;
  public abstract int size();

  //@ modifies \nothing;
  public abstract boolean isEmpty();

  //@ modifies \nothing;
  public abstract /*@non_null*/ Enumeration<K> keys();

  //@ modifies \nothing;
  public abstract /*@non_null*/ Enumeration<V> elements();

  //@ modifies \nothing;
  public abstract /*@nullable*/ V get(/*@non_null*/ Object key);

  public abstract /*@nullable*/ V put(/*@non_null*/ K key,
                                      /*@non_null*/ V value);

  public abstract /*@nullable*/ V remove(/*@non_null*/ Object key);
}
```

Figure 2.10: An external JML specification of the `java.util.Dictionary` abstract class (taken from JML *Specs* project).

The behaviour of a method is described by JML in the *design-by-contract* fashion, introduced by B. Meyer for Eiffel [81]. The behaviour description is placed within JML annotations, not to be confused with Java annotations. JML annotations are Java comments that have the at-sign (`@`) as their first character following the usual comment beginning. This way, they do not disturb standard Java compilers that treat JML annotations as regular Java comments and ignore them during the compilation process. There exists, however, a compiler JMLRac [24] (JML Runtime Assertion Checker) that is able to compile the specifications along with the regular program code, so that assertions present in specifications are verified during program run-time. There is a *Type Annotations Specification* concept (also known as *JSR 308*[1]) to extend regular Java annotations and use them for the spec-

[1]For details see `http://types.cs.washington.edu/jsr308/`

ification purpose. The idea is to allow Java annotations to be placed in any use of a type instead of declarations only, as it is currently allowed in Java 7. The idea is very limited compared to JML but still may be very useful, especially because there are plans to include type annotations to the next official OpenJDK 8.

JML includes annotations that make it possible to describe invariant properties that are maintained by objects, method specifications (pre- and post-conditions), and some lower level properties of the code (e.g. loop invariants or assertions). Let us focus on the example specification presented in Fig. 2.10. The constructor of the `java.util.Dictionary` class is said to be `pure`, which means that method has no side effects when executed [75, Sec. 6.2.5 and 7.1.1.3] (i.e. the state before the execution is equal to the one after the execution). In case of constructors, the attribute *pure* is equivalent to the clauses:

```
diverges false;
assignable this.*;
```

where `diverges false` means the constructor must return to the caller (either throw an exception or return normally [75, Sec. 9.9.7]), and `assignable this.*` clause means that only assignments to fields of the object being initialised are allowed during the execution of the constructor [75, Sec. 9.9.9]. The `modifies \nothing` specification for most of the methods of the class means that these methods do not modify any location — field of an object or a local variable. In fact, this is an equivalent to `pure`, since if `diverges` is not specified it defaults to `false`. We can see that only `put` and `remove` methods of a `Dictionary` can modify program state. The `non_null` modifier means that the result of a method (e.g. `keys`) or an argument (e.g. `key` argument of the method `get`) cannot have `null` value. Similarly, `nullable` modifier means that the result of a method or an argument may have `null` value.

JML already has a very rich tool support [20]. One of the projects in the JML infrastructure is *Specs*. It provides a specification of the standard Java library. Unfortunately, the project is not complete, specification is present only for some of the classes and interfaces. As we have already described in Section 1.2, there are two approaches for software verification. The *dynamic verification* is performed in JML by *run-time assertion checking* (RAC) tools, for example already mentioned JMLRac [24]. Two tools are especially popular for verification in the JML infrastructure: ESC/Java2 [26] and KeY [2].

Loop Variant Functions

Apart from pre-, postconditions and class invariants JML provides some lower level constructions such as loop invariants or loop variant functions. They are specifications that are often used to assist probing of function invariants. In JML, loop invariants and loop variant functions are placed in JML annotation just before the loop. An example is presented in Fig. 2.11. The `maintaining` keyword is followed by the loop invariant expression. In the example, there are two loop invariants present. The first one describes the range of values for the integer variable *i* and the second one relates the variable with a partial sum variable *sum*. A JML loop variant function is preceded by the `decreases` or `decreasing` keyword.

```
package org.jmlspecs.samples.jmlrefman;
public abstract class SumArrayLoop {
  ...
  public static long sumArray(int [] a) {
    long sum = 0; int i = a.length;

    /*@ maintaining -1 <= i && i <= a.length;
      @ maintaining sum ==
      @   (\sum int j; i <= j && 0 <= j && j < a.length; (\bigint)a[j]);
      @ decreasing i; @*/
    while (--i >= 0) {
      sum += a[i];
    }
   ...
    return sum;
  }
}
```

Figure 2.11: An example of a loop invariant and a loop variant function taken from JML reference manual.

In the thesis we focus on *loop variant functions*, especially on the *decreases* formula [75, Sec. 13.2.2]. The *decreases* formula specifies an expression that is of Java integer type (either `long` or `int`) that in every iteration:

- must be greater than or equal to 0,
- must decrease at least by one.

Semantics of the `decreases` formula

Here we present the semantics of the *decreases* formula in order to acquaint the reader with JML. The semantics is defined in terms of assertions [75, Sec. 13.2.2]. In order to obtain the semantics of a loop with the *decreases* formula, the loop is syntactically transformed to an equivalent one that contains `assert` statements. Here we present a few examples of such transformation for a `while` loop.

First, consider a simple loop that does not contain `continue` statement in its body, presented in Fig. 2.12(a). The *decreases* formula has to be checked in every loop iteration, therefore the translated version of the loop must check value of the expression `E` before and after an execution of the block `S`. The translated version

```
//@ decreases E;
while (B) {
 S
}
```

(a) Original loop

```
while (true) {
  long fv = E; // fv is fresh, unused variable
  if (!(B)) {
    break;
  }
  S
  //@ assert 0 <= fv;
  //@ assert E < fv;
}
```

(b) Loop with *decreases* translated to assertions

Figure 2.12: Translation of `while` loop without `continue` in its body.

```
//@ decreases E;
while (B) {
  S1
  if (C) {
    S2
    continue;
  }
  S3
}
```

(a) Original loop

```
while (true) {
  long fv = E; // fv is fresh, unused variable
  if (!(B)) {
    break;
  }
  S1
  if (C) {
    S2
    //@ assert 0 <= fv;
    //@ assert E < fv;
    continue;
  }
  S3
  //@ assert 0 <= fv;
  //@ assert E < fv;
}
```

(b) Loop with *decreases* translated to assertions

Figure 2.13: Translation of `while` with `continue` statement in its body.

of the loop is presented in Fig. 2.12(b). For the purpose of the assertion check we have introduced a fresh variable `fv` that was not present in the code before. We can see that the translation in the considered case is quite simple.

The situation is more complicated when the `while` loop contains the `continue` statement in its body. As the `continue` statement moves the execution of the loop straight to the next iteration, the loop variant has to be additionally checked before every use of the `continue` statement. A `while` loop with a single `continue` statement and its translated version are presented in Fig. 2.13. As before, we introduce a new variable `fv` that was not present in the code. The only difference is that assertions on the new variable are put into all the positions in the code, where we are just going to jump to the condition of the loop, thus also before all `continue` statement.

In the thesis we are more interested in `for` loops. The semantics of *decreases* formula for a `for` loop is quite similar. The idea is perform two translations: first, we translate a `for` loop into a `while` loop, and then, perform translations as described in the current section. The translation of a `for` loop into an equivalent `while` loop is as follows:

```
for (I; B; U) {
    S;
}
```

$\rightsquigarrow$

```
I;
while (B) {
  S;
  U;
}
```

where `I` are loop initializer expressions, `B` is the loop condition, `U` are update expressions, and `S` is the loop body.

Chapter 3

Numerical Abstract Domains

3.1 Introduction

In the thesis we deal with numerical abstract domains. These domains focus on values of numerical variables that appear in programs. The goal of static analysis based on numerical abstract domains is to collect the set of possible values of all numerical variables for every program control point. The set of possible values for a single control point is a set of vectors, which store valuation of all the variables. An abstraction is used to describe the set of possible values for variables in a compact and effective way. We can divide numerical abstract domains by their construction:
Non-relational domains are numerical abstract domains, in which no relationship between numerical variables is described, that is each variable is abstracted independently. The construction of non-relational domains can be generalised so that one has to define one-variable version of the domain and then the full domain is a pointwise lifting of this one-variable version. The only thing that has to be created for all variables is the transfer function. The most popular non-relational abstract domains are:

- *domain of signs* [32], where every numerical variable is abstracted by a possible sign of its value, an example is presented in Fig. 3.1 (b),
- *domain of intervals* [32] abstracts every variable by an interval that contains its possible values (this corresponds to a set of constraints $a \leq x \leq b$, where x is program variable and a, b are numerical constants), an example is presented in Fig. 3.1 (c),

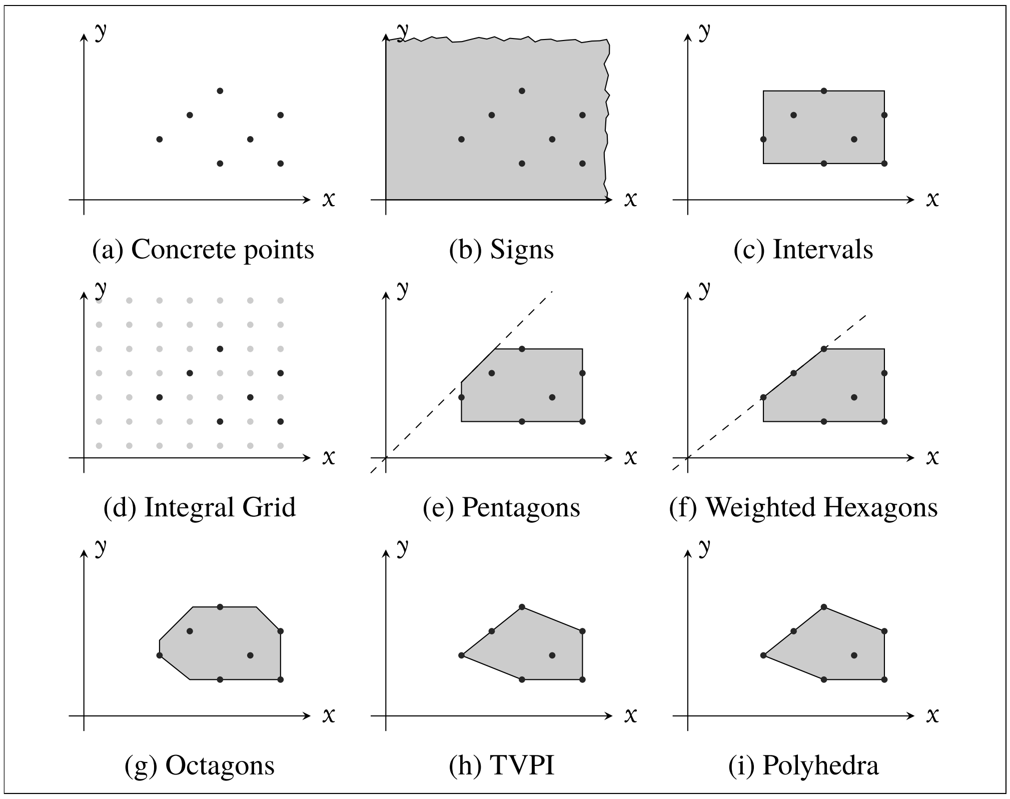


Figure 3.1: Comparison of abstract domains for 2-dimensional case.

- *domain of non-relational integral grids* [58, 59] that can be represented by a conjunctions of linear congruence relations of the form $\sum_i a_i * x_i = c$ mod m, an example is presented in Fig. 3.1 (d).

Relational domains are numerical abstract domains that make it possible to express also relationships between variables. The most popular relational domains are:

- *domain of pentagons* [77] is an extension of the domain of intervals by adding linear constraints of the form $x_1 < x_2$, where x_1, x_2 are program variables, an example is presented in Fig. 3.1 (e),

- *domain of weighted hexagons* [53] handles linear inequalities of the form $x_1 \leq a * x_2$, where x_1, x_2 are program variables and a is non-negative constant, an example is presented in Fig. 3.1 (f),

- *domain of octagons* [83] handles inequalities of the form $\pm x_1 \pm x_2 \leq a$, where a is a numerical constant and x_1, x_2 are program variables, an example is presented Fig. 3.1 (g),

- *domain of polyhedra* [40] handles linear inequalities among program variables, that is $a_1 * x_1 + a_2 * x_2 + \dots a_n * x_n \leq a_0$, where $n > 0$, $x_1, \dots, x_n$ are program variables and $a_0, a_1, \dots, a_n$ are numerical constants, an example is presented in Fig. 3.1 (i),

- *two variables per inequality*(TVPI) abstract domain [89] is a simplification of the domain of polyhedra, where linear constraints are limited to two variables, an example is presented in Fig. 3.1 (h). Note that the two-dimensional example is exactly the same as the one for polyhedra but in general the domain is less precise than the domain of polyhedra.

3.2 Construction of a Numerical Abstract Domain

In this section we focus on the analysis of numerical variables using the abstract interpretation framework. We state the requirements on the construction of a numerical abstract domain that we use in the thesis. The concrete domain for the *Simple* language is a tuple:

$$\langle \mathscr{P}(\mathit{State}), \subseteq, \emptyset, \mathit{State}, \cup, \cap \rangle,$$

where $\subseteq$, $\cap$ and $\cup$ are standard set operators. We recall that $\mathit{State} = \mathit{Var} \to \mathbb{I}$. Note that $\langle \mathscr{P}(\mathit{State}), \subseteq \rangle$ is a complete lattice, where the least element is the empty set $\emptyset$ and the greatest element is State, i.e. the set of all possible functions from Var to $\mathbb{I}$.

For the purpose of the thesis we assume that the *numeric abstract domain* is a bounded lattice. Let:

$$\langle \mathcal{D}^\sharp, \leq^\sharp, \bot^\sharp, \top^\sharp, \cup^\sharp, \cap^\sharp \rangle$$

be the abstract domain, where $\langle \mathcal{D}^\sharp, \leq^\sharp \rangle$ is a lattice, such that $\bot^\sharp$ is the least (bottom) element of the lattice, $\top^\sharp$ is the greatest (top) element of the lattice, $\cup^\sharp$ is the greatest upper bound operator (join) and $\cap^\sharp$ is the least upper bound operator. Additionally, we assume there is a concretisation function $\gamma : \mathcal{D}^\sharp \to \mathscr{P}(\mathit{State})$. An element of the abstract domain is a sound approximation of a set of possible valuations of variables at some control point (label). The full abstract semantics is a bounded lattice obtained by a simple lifting $\mathit{Lab} \to \mathcal{D}^\sharp$.

We set the following requirements on the construction of the *numerical abstract domain*:

- Elements of $\mathcal{D}^\sharp$ are computer-representable.

- The least abstract element is an abstraction of least concrete element, i.e. $\gamma(\bot^\sharp) = \varnothing$.
- The greatest abstract element is an abstraction of the greatest concrete element, i.e. $\gamma(\top^\sharp) = \mathit{Var} \to \mathbb{I}$.
- The abstract join $\cup^\sharp$ and meet $\cap^\sharp$ operators are sound approximations of concrete counterparts, that is for any $d_1^\sharp, d_2^\sharp \in D^\sharp$ and $S_1, S_2 \in \mathscr{P}(\mathbb{I})$ such that $S_1 \subseteq \gamma(d_1^\sharp) \land S_2 \subseteq \gamma(d_2^\sharp)$, it holds that:

$$S_1 \cup S_2 \subseteq \gamma(d_1^\sharp \cup^\sharp d_2^\sharp),$$
$$S_1 \cap S_2 \subseteq \gamma(d_1^\sharp \cap^\sharp d_2^\sharp).$$

- There is a widening operator $\triangledown^\sharp$ for the domain.
- There is a sound abstraction of the semantic function.

We implement the last item with two functions described in what follows.

Abstract Evaluation of Boolean Expressions
We require existence of a sound abstraction of the evaluation of boolean expressions: a function $\mathit{test} : \mathit{BExp} \times \mathit{Bool} \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ such that for every test expression $b \in \mathit{BExp}$, abstract state $d^\sharp \in \mathcal{D}^\sharp$, possible test result $bb \in \mathit{Bool}$, and a concrete state $s \in \mathit{State}$ it holds that:

$$(s \in \gamma(d^\sharp) \land \mathcal{B}\,[\![\mathrm{b}]\!]\; s = bb) \implies s \in \gamma(\mathit{test}(b, bb, d^\sharp)).$$

Additionally, we would like the *test* operation to narrow down the input set of states, therefore we add a condition that $\gamma(\mathit{test}(b, bb, d^\sharp)) \subseteq \gamma(d^\sharp)$. The definition is consistent with the soundness definition of transformer abstraction in Definition 2.4.3. The *test* function is illustrated in Fig. 3.2.

In order to simplify the definition of the *test* function we present generic rules to deal with non-atomic tests. When these rules are applied one is only required to define the *test* function for atomic tests. The first generic step is to apply syntactic transformation of the expressions. We apply De Morgan's laws to move the **not** operator into operands the **and** and **or** expression:

$$\begin{aligned} \textbf{not}\ (a\ \textbf{and}\ b) &\rightsquigarrow (\textbf{not}\ a)\ \textbf{or}\ (\textbf{not}\ b), \\ \textbf{not}\ (a\ \textbf{or}\ b) &\rightsquigarrow (\textbf{not}\ a)\ \textbf{and}\ (\textbf{not}\ b), \\ \textbf{not}\ (\textbf{not}\ a) &\rightsquigarrow a. \end{aligned}$$

```
ℒ : if b then
    ℒ_T
Else
    ℒ_F
fi
ℒ'
```

(a) Source code fragment

$$\begin{tikzcd}
d_1^\sharp \arrow[r, rightsquigarrow, "\sharp", "t"'] \arrow[dd, "\gamma"] & d_2^\sharp = test(b, tt, d_1^\sharp) \arrow[d, "\gamma"] \\
& S_2' \arrow[d, phantom, "\subseteq" sloped] \\
S_1 \arrow[r, rightsquigarrow, "t"'] & S_2 = \{s \mid s \in S_1 \wedge \mathcal{B}[\![\mathrm{b}]\!]\; s = tt\}
\end{tikzcd}$$

(b) Transfer functions

Figure 3.2: Abstract simulation of a concrete computation of a conditional instruction when the test condition is fulfilled.

The next step is to reverse expression comparisons when they are preceded by **not**:

$$\begin{aligned}
&\textbf{not}\ (e_1 \leq e_2) \rightsquigarrow e_2 < e_1,\\
&\textbf{not}\ (e_1 < e_2) \rightsquigarrow e_2 \leq e_1,\\
&\textbf{not}\ (e_1 = e_2) \rightsquigarrow e_2 \neq e_1,\\
&\textbf{not}\ (e_1 \neq e_2) \rightsquigarrow e_2 = e_1.
\end{aligned}$$

The last step is to compute the transfer function for non-atomic boolean expressions inductive application of the following rules:

$$\begin{aligned}
test(a\ \textbf{and}\ b, bb, d^\sharp) &= test(a, bb, d^\sharp) \sqcap^\sharp test(b, bb, d^\sharp),\\
test(a\ \textbf{or}\ b, bb, d^\sharp) &= test(a, bb, d^\sharp) \sqcup^\sharp test(b, bb, d^\sharp).
\end{aligned}$$

Abstract Evaluation of Assignment Expressions
We require existence of a sound abstraction of the assignment operation: a function $assign : \mathit{Var} \times \mathit{Exp} \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ such that for any abstract element $d^\sharp \in D^\sharp$, an expression $e \in \mathit{Exp}$, a variable $v \in \mathit{Var}$ and a concrete state $s \in \mathit{State}$ it holds that:

$$(s \in \gamma(d^\sharp) \wedge \mathcal{E}[\![\mathrm{e}]\!]\; s = w) \Longrightarrow s[v \leftarrow w] \in \gamma(assign(v, e, d^\sharp)).$$

The definition is also consistent with the soundness definition of transformer abstraction in Definition 2.4.3. Abstract simulation using *assign* function is illustrated in Fig. 3.3.

3.2.1 Abstract Semantics

Here we describe the abstract semantics of the *Simple* language. We assume that we have an instance of an abstract numerical domain from the previous section.

$\mathcal{L} : v \leftarrow e$
$\mathcal{L}'$

(a) Source code fragment

$$\begin{tikzcd}
d_1{}^\sharp \arrow[r, rightsquigarrow, "\sharp", "t"'] \arrow[dd, "\gamma"] & d_2{}^\sharp = assign(v, e, d_1^\sharp) \arrow[d, "\gamma"] \\
 & S_2' \arrow[d, phantom, "\rotatebox{90}{$\subseteq$}"] \\
S_1 \arrow[r, rightsquigarrow, "t"'] & S_2 = \{s[v \leftarrow \mathcal{E}[\![\mathrm{e}]\!]\, s] \mid s \in S_1\}
\end{tikzcd}$$

(b) Transfer functions

Figure 3.3: Abstract simulation of a concrete computation of the assignment instruction $v \leftarrow e$.

The semantics is a set of functions that describe transitions between labels, thus they are very similar to those of the concrete semantics. The difference is that we use the *test* and the *assign* operations instead of real evaluation of expressions and conditions.

First, in Fig. 3.4(a) we present the abstract semantics of **skip**, which is analogical to the concrete one since there is no modification of the abstract state. The semantics of the abstract assignment, which is presented in Fig. 3.4(b), uses the *assign* operation that comes from the definition of the abstract domain from the previous section. The abstract semantics of a conditional instruction is presented in Fig. 3.4(c). The first two transitions use the *test* operation to evaluate the condition. The last two transitions are analogical to the ones in the concrete semantics. Finally, the semantics of the **while** loop is presented in Fig. 3.4(d). The first two transitions use the *test* operation while the last two remain unchanged.

Since we have transfer functions for the abstract domain (they correspond to transitions presented in Fig. 3.4) and Theorem 2.4.11, the only thing missing from a working abstract interpreter is an numerical abstract domain that matches the assumptions from Section 3.2.

3.3 Examples of Numerical Abstract Domains

In this section we provide a few examples of numerical abstract domains. We focus on non-relational ones, because a domain of this kind is a subject of our analysis in the next chapter.

Since non-relational abstract domains do not handle constraints between variables, every variable is abstracted independently. It is possible to define a one-dimensional version of an abstract domain together with domain operations, widening operator and abstract transfer functions, and then, lift this abstract domain to

$$\frac{\mathcal{C} = \textbf{skip} \quad at_P(\mathcal{C}) = \mathcal{L} \quad after_P(\mathcal{C}) = \mathcal{L}'}{\langle \mathcal{L}, s^\sharp \rangle \rightsquigarrow_t^\sharp \langle \mathcal{L}', s^\sharp \rangle}$$

(a) Abstract transition for **skip**

$$\frac{\mathcal{C} = x \leftarrow e \quad at_P(\mathcal{C}) = \mathcal{L} \quad after_P(\mathcal{C}) = \mathcal{L}'}{\langle \mathcal{L}, s^\sharp \rangle \rightsquigarrow_t^\sharp \langle \mathcal{L}', assign(x, e, s^\sharp) \rangle}$$

(b) Abstract transition for the assignment instruction

$$\frac{\mathcal{C} = \textbf{if } b \textbf{ then } \mathcal{B}_t \textbf{ else } \mathcal{B}_f \textbf{ fi} \quad at_P(\mathcal{C}) = \mathcal{L} \quad at_P(\mathcal{B}_t) = \mathcal{L}'}{\langle \mathcal{L}, d^\sharp \rangle \rightsquigarrow_t^\sharp \langle \mathcal{L}', test(b, tt, d^\sharp) \rangle}$$

$$\frac{\mathcal{C} = \textbf{if } b \textbf{ then } \mathcal{B}_t \textbf{ else } \mathcal{B}_f \textbf{ fi} \quad at_P(\mathcal{C}) = \mathcal{L} \quad at_P(\mathcal{B}_f) = \mathcal{L}'}{\langle \mathcal{L}, d^\sharp \rangle \rightsquigarrow_t^\sharp \langle \mathcal{L}', test(b, ff, d^\sharp) \rangle}$$

$$\frac{\mathcal{C} = \textbf{if } b \textbf{ then } \mathcal{B}_t \textbf{ else } \mathcal{B}_f \textbf{ fi} \quad after_P(\mathcal{B}_f) = \mathcal{L} \quad after_P(\mathcal{C}) = \mathcal{L}'}{\langle \mathcal{L}, d^\sharp \rangle \rightsquigarrow_t^\sharp \langle \mathcal{L}', d^\sharp \rangle}$$

$$\frac{\mathcal{C} = \textbf{if } b \textbf{ then } \mathcal{B}_t \textbf{ else } \mathcal{B}_f \textbf{ fi} \quad after_P(\mathcal{B}_t) = \mathcal{L} \quad after_P(\mathcal{C}) = \mathcal{L}'}{\langle \mathcal{L}, d^\sharp \rangle \rightsquigarrow_t^\sharp \langle \mathcal{L}', d^\sharp \rangle}$$

(c) Abstract transitions for the **if-then-else** instruction

$$\frac{\mathcal{C} = \textbf{while } \mathcal{L}_w : b \textbf{ do } \mathcal{B} \textbf{ od} \quad at_P(\mathcal{B}) = \mathcal{L}}{\langle \mathcal{L}_w, d^\sharp \rangle \rightsquigarrow_t^\sharp \langle \mathcal{L}, test(b, tt, d^\sharp) \rangle}$$

$$\frac{\mathcal{C} = \textbf{while } \mathcal{L}_w : b \textbf{ do } \mathcal{B} \textbf{ od} \quad after_P(\mathcal{C}) = \mathcal{L}}{\langle \mathcal{L}_w, d^\sharp \rangle \rightsquigarrow_t^\sharp \langle \mathcal{L}, test(b, ff, d^\sharp) \rangle}$$

$$\frac{\mathcal{C} = \textbf{while } \mathcal{L}_w : b \textbf{ do } \mathcal{B} \textbf{ od} \quad after_P(\mathcal{B}) = \mathcal{L}}{\langle \mathcal{L}, d^\sharp \rangle \rightsquigarrow_t^\sharp \langle \mathcal{L}_w, d^\sharp \rangle}$$

$$\frac{\mathcal{C} = \textbf{while } \mathcal{L}_w : b \textbf{ do } \mathcal{B} \textbf{ od} \quad at_P(\mathcal{C}) = \mathcal{L}}{\langle \mathcal{L}, d^\sharp \rangle \rightsquigarrow_t^\sharp \langle \mathcal{L}_w, d^\sharp \rangle}$$

(d) Abstract transitions for the **while** instruction

Figure 3.4: Abstract transitions for the *Simple* language.

create domain that handles multiple variables. The one-dimensional domain is called a *non-relational basis*. Let $\langle \mathcal{B}, \leq_{\mathcal{B}} \rangle$ be a one-dimensional version of the abstract domain that abstracts $\langle \mathscr{P}(\mathbb{I}), \subseteq \rangle$. Additionally, we assume that there is a pair of functions $\alpha_{\mathcal{B}} : \mathscr{P}(\mathbb{I}) \to \mathcal{B}$ and $\gamma_{\mathcal{B}} : \mathcal{B} \to \mathscr{P}(\mathbb{I})$ that form a Galois connection:

$$\langle \mathscr{P}(\mathbb{I}), \subseteq \rangle \underset{\alpha_{\mathcal{B}}}{\overset{\gamma_{\mathcal{B}}}{\leftrightarrows}} \langle \mathcal{B}, \leq_{\mathcal{B}} \rangle.$$

The multi-dimensional domain $\mathcal{B}' = \mathit{Var} \to \mathcal{B}$ is obtained by a pointwise lifting of $\mathcal{B}$ to Var [31, Sec. 8]. Additionally, there is a Galois connection between the concrete domain $\mathscr{P}(\mathit{Var} \to \mathbb{I})$ and $\mathcal{B}'$, which is stated by Theorem 3.3.1.

Theorem 3.3.1. *Let $\langle \mathcal{B}, \leq_{\mathcal{B}} \rangle$ be a complete lattice and let the pair of functions $\alpha_{\mathcal{B}} : \mathscr{P}(\mathbb{I}) \to \mathcal{B}$, $\gamma_{\mathcal{B}} : \mathcal{B} \to \mathscr{P}(\mathbb{I})$ form a Galois connection. Then $\langle \mathcal{B}^{\mathit{Var}}, \leq_{\mathit{Var} \to \mathcal{B}} \rangle$ is a complete lattice, where $\leq_{\mathit{Var} \to \mathcal{B}}$ is a pointwise ordering. Additionally, two functions $\gamma : \mathcal{B}^{\mathit{Var}} \to \mathscr{P}(\mathit{Var} \to \mathbb{I})$ and $\alpha : \mathscr{P}(\mathit{Var} \to \mathbb{I}) \to \mathcal{B}^{\mathit{Var}}$ defined as:*

$$\gamma(f) = \{g : \mathit{Var} \to \mathbb{I} \mid \forall_{v \in \mathit{Var}} g(x) \in \gamma_{\mathcal{B}}(f(v))\},$$
$$\alpha(X) = \lambda v : \alpha_{\mathcal{B}}(\{g(v) \mid g \in X\})$$

form a Galois connection:

$$\langle \mathscr{P}(\mathit{Var} \to \mathbb{I}), \subseteq \rangle \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \langle \mathcal{B}^{\mathit{Var}}, \leq_{\mathit{Var} \to \mathcal{B}} \rangle.$$

Proof. See [43, Prop. 3]. □

Theorem 3.3.1 yields a construction of non-relational domains, where one only has to define domain operations for one-dimensional case. The extension to multi-dimensional version is generic. Since in expressions there may exist multiple variables, one still has to define the transfer function for the multi-dimensional domain. We define abstract semantics of expressions:

$$\mathcal{E}^{\sharp} : \mathit{Exp} \to (\mathcal{B}^{\mathit{Var}} \to \mathcal{B})$$

as presented in Fig. 3.5. Note that the semantics uses abstract operators: unary $-^{\sharp}$ and binary $+^{\sharp}$, $-^{\sharp}$, $*^{\sharp}$.

With the abstract semantics of expressions the definition of the *assign* (see Section 3.2) function is straightforward. It is enough to evaluate the right hand side of the assignment and update the abstract value associated with the variable on left hand side, that is:

$$\mathit{assign}(v, e, \bot^{\sharp}) = \bot^{\sharp}, \qquad \mathit{assign}(v, e, d^{\sharp}) = \lambda w \,.\begin{cases} \mathcal{E}^{\sharp}[\![e]\!]\, d^{\sharp} & \text{if } v = w \\ d^{\sharp}(w) & \text{otherwise.} \end{cases}$$

For boolean test expressions it is enough to present definition of the *test* function for atomic tests.

$$\begin{array}{lcl}
\mathcal{E}^\sharp [\![v]\!]\, d^\sharp & = & d^\sharp(v) \\
\mathcal{E}^\sharp [\![c]\!]\, s & = & \mathcal{N}^\sharp [\![c]\!] \\
\mathcal{E}^\sharp [\![-e]\!]\, d^\sharp & = & -^\sharp\, \mathcal{E}^\sharp [\![e]\!]\, d^\sharp \\
\mathcal{E}^\sharp [\![e_1 + e_2]\!]\, d^\sharp & = & \mathcal{E}^\sharp [\![e_1]\!]\, d^\sharp +^\sharp \mathcal{E}^\sharp [\![e_2]\!]\, d^\sharp \\
\mathcal{E}^\sharp [\![e_1 * e_2]\!]\, d^\sharp & = & \mathcal{E}^\sharp [\![e_1]\!]\, d^\sharp *^\sharp \mathcal{E}^\sharp [\![e_2]\!]\, d^\sharp \\
\mathcal{E}^\sharp [\![e_1 - e_2]\!]\, d^\sharp & = & \mathcal{E}^\sharp [\![e_1]\!]\, d^\sharp -^\sharp \mathcal{E}^\sharp [\![e_2]\!]\, d^\sharp
\end{array}$$

where $v \in \mathsf{Var}$, $c \in \mathsf{Num}$, $e_1, e_2 \in \mathsf{Exp}$ and $d^\sharp \in \mathcal{D}^\sharp$.

Figure 3.5: Non-relational abstract semantics of numerical expressions.

3.3.1 Domain of Signs

The first of the numerical abstract domains presented in the thesis is a simple non-rational domain — the domain of signs, Sgn [32, 33]. The idea is to forget particular values of numerical variables, instead we focus only on their signs. For elementary operations, e.g. multiplication, such abstraction does not introduce any loss of precision, i.e. to compute the sign of a product it is sufficient to know only the signs of the operands. Still, other operations may introduce imprecision. For example, it is impossible to know the sign of a sum, where operands have opposite signs — one of them is negative and another one is positive.

$$\begin{tikzcd}
& \pm_{Sgn} & \\
-_{Sgn} \arrow[ur, no head] & 0_{Sgn} \arrow[u, no head] & +_{Sgn} \arrow[ul, no head] \\
& \bot_{Sgn} \arrow[ul, no head] \arrow[u, no head] \arrow[ur, no head] &
\end{tikzcd}$$

Figure 3.6: A lattice for the domain Sgn.

The Sgn domain uses a simple finite lattice to abstract element values. There are several ways to create the lattice [39, Ex. 10.2.0.2]. Our choice is presented in Fig. 3.6.

Concretisation and Abstraction

In this section we present the abstraction and concretisation functions for the domain of signs. Both, the abstraction function $\alpha_{Sgn} : \mathscr{P}(\mathbb{I}) \to Sgn$ and the concretisation function $\gamma_{Sgn} : Sgn \to \mathscr{P}(\mathbb{I})$, are quite straightforward. They are defined as follows:

$$\alpha_{Sgn}(\mathcal{X}) = \begin{cases} \bot_{Sgn} & \text{if } \mathcal{X} = \varnothing \\ 0_{Sgn} & \text{if } \mathcal{X} = \{0\} \\ -_{Sgn} & \text{if } \forall x \in \mathcal{X} : x < 0 \\ +_{Sgn} & \text{if } \forall x \in \mathcal{X} : 0 < x \\ \pm_{Sgn} & \text{otherwise,} \end{cases} \quad \gamma_{Sgn}(s) = \begin{cases} \varnothing & \text{if } s = \bot_{Sgn} \\ \{0\} & \text{if } s = 0_{Sgn} \\ \{x \in \mathbb{I} \mid x < 0\} & \text{if } s = -_{Sgn} \\ \{x \in \mathbb{I} \mid 0 < x\} & \text{if } s = +_{Sgn} \\ \mathbb{I} & \text{if } s = \pm. \end{cases}$$

It is easy to verify that both functions form a Galois connection, that is:

$$\langle Sgn, \leq_{Sgn} \rangle \underset{\alpha_{Sgn}}{\overset{\gamma_{Sgn}}{\leftrightarrows}} \langle \mathscr{P}(\mathbb{I}), \subseteq \rangle.$$

Transfer functions

The final part required by the abstract domain is a definition of the transfer functions. First, we define the transfer function for numeric expressions. Semantics of simple expressions is presented in Fig. 3.7.

$$\mathcal{N}^\sharp[\![c]\!] \overset{\text{def}}{=} \begin{cases} -_{Sgn} & \text{if } c > 0 \\ 0_{Sgn} & \text{if } c = 0 \\ +_{Sgn} & \text{otherwise,} \end{cases} \qquad -^\sharp d^\sharp \overset{\text{def}}{=} \begin{cases} -_{Sgn} & \text{if } d^\sharp = +_{Sgn} \\ +_{Sgn} & \text{if } d^\sharp = -_{Sgn} \\ d^\sharp & \text{otherwise.} \end{cases}$$

(a) Numeric constant (b) Unary minus

Figure 3.7: Abstract semantics of simple expressions.

In case of any other domain operators, if any of the arguments is $\bot_{Sgn}$ then the result is also $\bot_{Sgn}$. The semantics of a multiplication presented in Fig. 3.8 is a precise operation. We always can deduce the exact sign of the result when we know signs of both arguments.

$$d_1^\sharp *^\sharp d_2^\sharp \overset{\text{def}}{=} \begin{cases} +_{Sgn} & \text{if } d_1^\sharp = d_2^\sharp = +_{Sgn} \text{ or } d_1^\sharp = d_2^\sharp = -_{Sgn} \\ 0_{Sgn} & \text{if } d_1^\sharp = 0_{Sgn} \text{ or } d_2^\sharp = 0_{Sgn} \\ -_{Sgn} & \text{if } d_1^\sharp = -_{Sgn} \text{ and } d_2^\sharp = +_{Sgn} \text{ or } d_1^\sharp = +_{Sgn} \text{ and } d_2^\sharp = -_{Sgn} \\ \pm_{Sgn} & \text{otherwise.} \end{cases}$$

Figure 3.8: Abstract semantics of multiplication expression.

The semantics of a sum presented in Fig. 3.9 is not a precise operation. The imprecision is when both arguments have opposite signs. Then we are unable to deduce the sign of the result based only on signs of both arguments, because it depends on the difference of their values. The transfer function for a binary minus operation can be easily created from transfer functions of a unary minus and a binary sum.

$$d_1^\sharp +^\sharp d_2^\sharp \overset{\text{def}}{=} \begin{cases} +_{Sgn} & \text{if } +_{Sgn} \in \{d_1\} \cup \{d_2\} \text{ and } \{d_1\} \cup \{d_2\} \subseteq \{0_{Sgn}, +_{Sgn}\} \\ 0_{Sgn} & \text{if } -_{Sgn} \in \{d_1\} \cup \{d_2\} \text{ and } \{d_1\} \cup \{d_2\} \subseteq \{0_{Sgn}, -_{Sgn}\} \\ \pm_{Sgn} & \text{otherwise.} \end{cases}$$

Figure 3.9: Abstract semantics of sum expression.

An example of the implementation of a *test* function for an atomic comparison is presented in Fig. 3.10. The rest of atomic comparisons are similar, thus we do not present them here.

$$test(v_1 \leq v_2, tt, d^\sharp) \overset{\text{def}}{=} \begin{cases} \bot_{Sgn} & \text{if } v_1 = +_{Sgn} \text{ and } v_2 \in \{-_{Sgn}, 0_{Sgn}\} \\ \bot_{Sgn} & \text{if } v_2 = -_{Sgn} \text{ and } v_1 = 0_{Sgn} \\ d^\sharp & \text{otherwise.} \end{cases}$$

Figure 3.10: Abstract semantics of multiplication expression.

3.3.2 Domain of Intervals

The abstract domain of intervals uses a concept of *interval arithmetic*, which was originally introduced to handle rounding errors. The idea to adapt the technique for the abstract interpretation was introduced by Cousot and Cousot [38]. We present here the *non-relational basis* of the domain, which is lifted to multiple dimensions using standard non-relational domain lifting construction. We present here a classical version that handles only non-strict interval constraints since it is enough to understand the general idea. The extension introduces unnecessary complication level.

Domain structure

The one-dimensional version of the domain of intervals is a tuple:

$$\langle \mathbb{B}_v, \subseteq_v, \bot_v, \top_v, \uplus, \cap \rangle,$$

where:

- $\mathbb{B}_v$ is the set of all possible non-strict intervals on $\mathbb{I}$, that is:
$$\mathbb{B}_v = \{[a,b] \mid a \in \mathbb{I} \cup \{-\infty\} \text{ and } b \in \mathbb{I} \cup \{\infty\} \text{ and } a \leq b\}.$$
- $\bot_v$ is the smallest element of the lattice — the empty interval. We may write $\emptyset$ to denote the element.
- $\top_v$ is the biggest element in the lattice, that is $\top_v = [-\infty, \infty]$. We may write $\mathbb{I}$ to denote the element.
- $\subseteq_v$ is a subset ordering on elements of $\mathbb{B}_v$ such that for any $\mathcal{X} \in \mathbb{B}_v$ it holds that $\bot_v \subseteq_v \mathcal{X}$ and $[a_1,b_1] \subseteq_v [a_2,b_2] \iff a_2 \leq a_1$ and $b_1 \leq b_2$, where $[a_1,b_1],[a_2,b_2] \in \mathbb{B}_v$ and $\leq$ is natural ordering on $\mathbb{I} \cup \{-\infty, \infty\}$.
- $\uplus$ is the join operator defined for any $\mathcal{X}, Y \in \mathbb{B}_v$ as follows:
$$\mathcal{X} \uplus Y \stackrel{\text{def}}{=} \begin{cases} \mathcal{X} & \text{if } Y = \bot_v \\ Y & \text{if } \mathcal{X} = \bot_v \\ [\min(a_1,a_2), \max(b_1,b_2)] & \text{if } \mathcal{X} = [a_1,b_1] \text{ and } Y = [a_2,b_2]. \end{cases}$$
Note that the join operator may over-approximate the exact result, for example when the arguments of join are disjoint intervals.
- $\cap$ is the meet operator defined for any $\mathcal{X}, Y \in \mathbb{B}_v$ as follows:
$$\mathcal{X} \cap Y \stackrel{\text{def}}{=} \begin{cases} [\max(a_1,a_2), \min(b_1,b_2)] & \text{if } \mathcal{X} = [a_1,b_1] \text{ and } Y = [a_2,b_2] \\ & \text{ and } \max(a_1,a_2) \leq \min(b_1,b_2) \\ \bot_v & \text{otherwise.} \end{cases}$$
The meet operator is always precise.

The structure of the lattice for the case of integer interval endings is illustrated in Fig. 3.11. When $\mathbb{I} \in \{\mathbb{Z}, \mathbb{R}\}$ the lattice is complete but for $\mathbb{I} = \mathbb{Q}$ it is not.

The concretisation function for the domain $\gamma_v : \mathbb{B}_v \to \mathscr{P}(\mathbb{I})$ is defined as follows:
$$\gamma_v(\mathcal{X}) \stackrel{\text{def}}{=} \begin{cases} \emptyset & \text{if } \mathcal{X} = \bot_v \\ \{i \in \mathbb{I} \mid a \leq i \leq b\} & \text{otherwise, for } \mathcal{X} = [a,b] \end{cases}$$
and the abstraction function $\alpha_v : \mathscr{P}(\mathbb{I}) \to \mathbb{B}_v$ for $\mathbb{I} = \mathbb{Z}$ is defined as:
$$\alpha_v(I) \stackrel{\text{def}}{=} \begin{cases} \bot_v & \text{if } I = \emptyset \\ [\min(I), \max(I)] & \text{otherwise.} \end{cases}$$

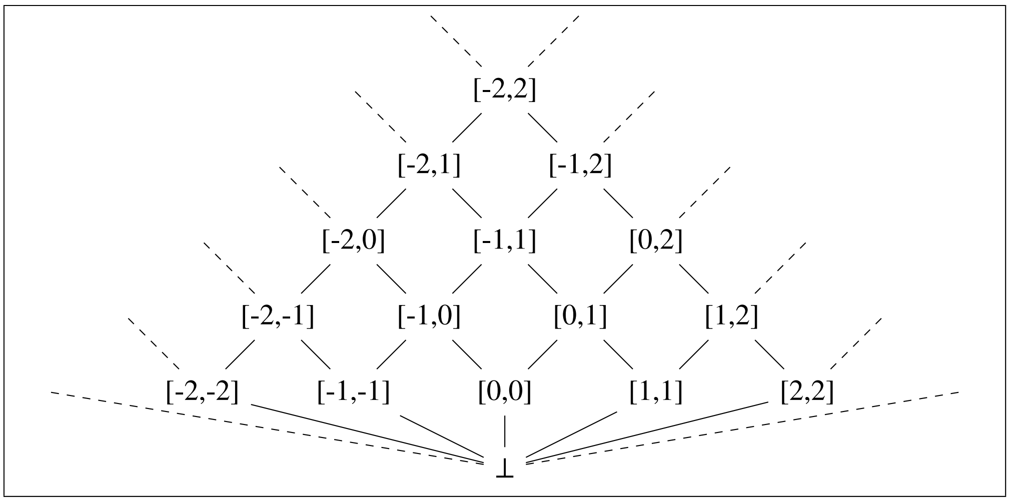


Figure 3.11: A fragment of the lattice for the abstract domain of intervals for $\mathbb{I} = \mathbb{Z}$.

In case of $\mathbb{I} = \mathbb{R}$, instead of min and max, we can use the infimum and supremum respectively. When $\mathbb{I} = \mathbb{Q}$ the lattice is not complete and the abstraction function does not exist.

Widening operator

The domain of intervals is of infinite size and there exist infinite sequences of strictly increasing elements in $\mathbb{B}_v$. For example, the sequence $[0, 1], [0, 2], [0, 3], \ldots$ is strictly increasing. Therefore, a widening operator is required in order to terminate the process of abstract interpretation. The idea behind the widening operator is to go straight to ∞ when the end of the interval in the second argument of widening is greater than the end of the interval in the first argument. The situation is symmetrical for the beginnings of intervals. For any $\mathcal{X}, Y \in \mathbb{B}_v$ the widening operator is defined as follows:

$$\mathcal{X} \triangledown Y \overset{\text{def}}{=} \begin{cases} [a, b] & \text{if } \mathcal{X} = [a_1, b_1] \text{ and } Y = [a_2, b_2] \\ Y & \text{if } \mathcal{X} = \bot_v \\ \mathcal{X} & \text{if } Y = \bot_v, \end{cases}$$

where:

$$a = \begin{cases} a_1 & \text{if } a_1 \leq a_2 \\ -\infty & \text{otherwise,} \end{cases} \qquad b = \begin{cases} b_1 & \text{if } b_2 \leq b_1 \\ \infty & \text{otherwise.} \end{cases}$$

Transfer functions

In order to apply the generic construction of the *assign* function we have do define abstract semantics of expressions for the domain of intervals. In case of any domain operator, if any of the arguments is $\bot_v$ then the result is also $\bot_v$. The semantics for the rest of situations is presented in Fig. 3.12.

$$\begin{aligned}
\mathcal{N}^\sharp[\![c]\!] &\stackrel{\text{def}}{=} [c, c]\\
-^\sharp[a, b] &\stackrel{\text{def}}{=} [-b, -a]\\
[a_1, b_1] +^\sharp [a_2, b_2] &\stackrel{\text{def}}{=} [a_1 + a_2, b_1 + b_2]\\
[a_1, b_1] -^\sharp [a_2, b_2] &\stackrel{\text{def}}{=} [a_1 - b_2, b_1 - a_2]\\
[a_1, b_1] *^\sharp [a_2, b_2] &\stackrel{\text{def}}{=} [\min(a_1 * a_2, a_1 * b_2, b_1 * a_2, b_1 * b_2),\\
&\qquad \max(a_1 * a_2, a_1 * b_2, b_1 * a_2, b_1 * b_2)]
\end{aligned}$$

Figure 3.12: Abstract semantics of expressions for the domain of intervals.

Here we present a few examples of a *test* function for atomic comparisons. When the third argument of the *test* function is bottom element, the result is also bottom, i.e. $test(b, tt, \bot_v) \stackrel{\text{def}}{=} \bot_v$. A simple comparison of a variable and a constant for the case of $\mathbb{I} = \mathbb{Z}$ is as follows:

$$test(v < c, tt, d^\sharp) \stackrel{\text{def}}{=} \lambda w : \begin{cases} d^\sharp(w) & \text{if } w \neq v \\ [a_v, \min(c-1, b_v)] & \text{otherwise for } d^\sharp(v) = [a_v, b_v]. \end{cases}$$

A little more complex comparison of two variables $u, v \in \mathit{Var}$ such that $u \neq v$ is as follows:

$$test(u < v, tt, d^\sharp) \stackrel{\text{def}}{=} \lambda w : \begin{cases} d^\sharp(w) & \text{if } w \notin \{u, v\} \\ [a_v, b_v] \cap [-\infty, b_u] & \text{if } w = v \\ [a_u, b_u] \cap [a_v, +\infty] & \text{if } w = u, \end{cases}$$

where $d^\sharp(u) = [a_u, b_u]$ and $d^\sharp(v) = [a_v, b_v]$.

3.4 Path-sensitive analysis

Numerical domains that were mentioned in the previous section as well as most of the existing ones share two certain features:

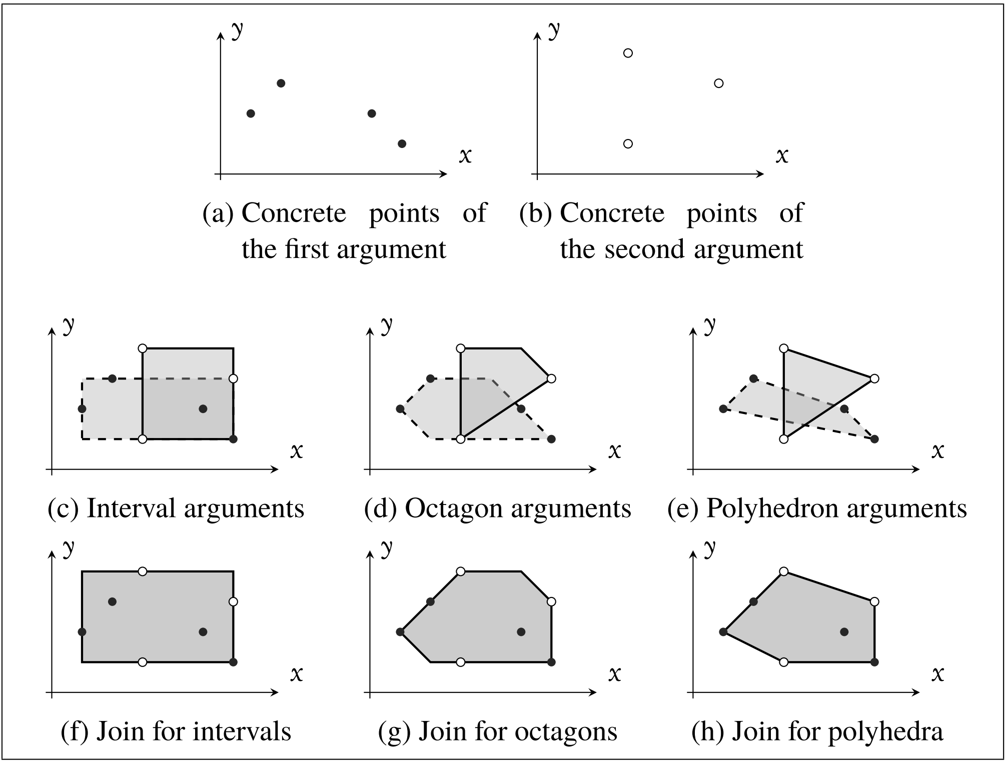


Figure 3.13: Examples of imprecise join for various numerical domains: (a)-(b) concrete arguments, (c)-(e) various representations of arguments, (f)-(h) various results of join.

- meet operation is precise in the domain — the result is an element of the domain,
- join operation is not precise — the result may not be an element of the domain and usually is over-approximated to the smallest element that contains the result (in case of polyhedra such element may not exist).

Examples of over-approximation during the join operation for a few of numerical domains from the previous section are presented in Fig. 3.13.

Let us consider an analysis that uses domains that over-approximate join. Suppose that a program P at some control point $\mathcal{L}$ has a few input edges. When the analysis goes through the point it merges information from all the input edges using the join operation. Since this is an over-approximating procedure, one may loose particular information about each of the input edges (and also about paths that lead to the label $\mathcal{L}$). Therefore, such analysis is called *path-insensitive*. Another disadvantage of the *path-insensitive* analysis is that it produces high rate of false positives. In disjunctive analysis the join operation is precise, therefore information about each of the input paths is kept (as disjuncts) — such analysis is called *path-sensitive*.

An overview of techniques used for the *path-sensitive* analysis is presented in Section 3.4.1. In Section 3.4.2 we describe a *trace partitioning* technique introduced by Mauborgne and Rival [79]. Next, in Section 3.4.3, we present a traditional construction of a *disjunctive refinement* (or *disjunctive completion*) [39] with an overview of methods for creating widening operators.

3.4.1 Disjunctive Refinement

The idea of a *disjunctive completion* [37, 39] was introduced by Cousot and Cousot as one of generic constructions that can be used to enhance abstract domains. Let $\mathcal{D}$ be the base domain that has precise meet but join may loose some information. The idea is to move to a more expressive abstract domain $\mathscr{P}(\mathcal{D})$, where sets of elements from the original domain $\mathcal{D}$ represent disjunctions of these elements. The original definition of *disjunctive completion* allows for any set of disjuncts, even infinite.

The construction by Patric Cousot is not very practical since the number of disjuncts may be infinite. The *finite powerset domain* [8, 10] is a similar construction to the *disjunctive completion* with the difference that the number of disjuncts is finite (but not bounded). The assumption is reasonable since the representation of infinite number of disjuncts may be very hard, or even impossible to store in the memory of a computer.

In the thesis we are interested in a *disjunctive refinement*, which is a general term for dealing with refinements of domains that handle disjunctions. Patric Cousot's *disjunctive completion* is an example of the *disjunctive refinement*. Popular techniques, such as trace partitioning [79] or splitting locations in the control flow graph [88], allow bounded number of disjuncts. The main disadvantage of these techniques is that they usually do not scale well to a large number of disjuncts.

3.4.2 Trace Partitioning

There are various techniques to introduce path-sensitivity and, therefore, improve precision beside the disjunctive refinement construction. The idea of *trace partitioning* was proposed by Handjieva and Tzolovski [63], then expanded and generalised by Rival and Mauborgne [79, 87]. In the described approach the set of all possible finite traces Σ_t^* (see Section 2.4.5) is split into number of groups using some chosen criterion. Instead of handling every possible trace separately one can look at groups of traces that share a specific characteristic. When the abstraction is well-chosen, static analysis using the abstract domain may be feasible.

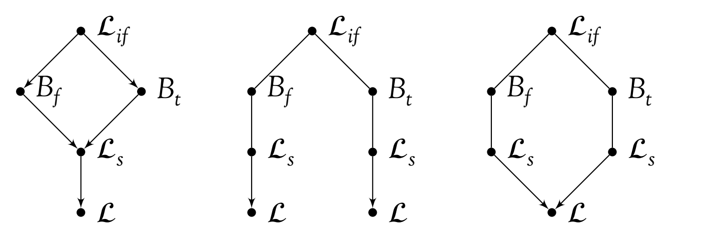
The *trace discriminating reachability semantics* that was introduced by Rival and Mauborgne lies in terms of precision between the *partial trace semantics* and the *reachability semantics*:

- it abstracts the *partial trace semantics* — makes some of the traces indistinguishable (they are put into one group),
- the *reachability semantics* is an abstraction of the *trace discriminating reachability semantics* — any information about traces is abandoned, only reachable states are considered.

The technique is generic and can be applied to any partitioning of the set of traces. Since we would like to improve accuracy of the join operation, considering traces that go through control points of a program, where branching appears, are especially interesting.

```
$\mathcal{L}_{if}$ : **if** $c$ **then**
      $B_t$
**else**
      $B_f$
**fi**
$\mathcal{L}_s$ : $s$
$\mathcal{L}$
```

(a) Source code



(b) Control flow graph (c) Control flow partition (d) Merging control flow partition

Figure 3.14: Example of a program trace partitions.

The first place to improve precision are conditional instructions. As an example, we use the domain of signs Sgn as the base domain and we analyse a fragment of a program presented in Fig. 3.14(a). The program consists of a conditional instruction and exactly one instruction s that appears after it. In Fig. 3.14 (b) a control flow graph of the program is presented. Abstract interpretation algorithm using the base domain of signs, presented in Section 3.3.1, would result in applying the join operation at the label $\mathcal{L}_s$. This may cause loss of precision. Assume that after the block B_f we have information that some variable v has the abstract value of $+_{Sgn}$, while after block B_t it has the value of $-_{Sgn}$. The application of join in the label $\mathcal{L}_s$ would cause the value of v to be $\pm_{Sgn}$. Therefore, we loose information that v cannot be equal to 0. If the instruction s is a division by v, the analysis would say that there is a division by 0 error, which is a false positive. If we use control flow partition, where traces that go though the block B_t are distinguished from the ones that go through the block B_f — presented in Fig. 3.14 (c), we get two instances of the label $\mathcal{L}_s$ (and also $\mathcal{L}$) to analyse (therefore, such analysis is also called *control flow based partition*) but in both cases one may reason that v is not equal to 0.

The second place, where branching is introduced, are loop instructions. One of strategies that can be applied to those is called *loop unrolling*. It is especially useful in cases, where computation of the widening in the initial iterations may result in a very imprecise result. The idea is to delay application of the widening operator — for the first n iterations the widening is not applied (a regular join is computed), and then, for the following iterations widening operator is applied. The result may be then merged at the loop end or every element of partitions can be analysed separately.

In the example from Fig. 3.14 (c) the traces that go through the positive branch and negative branch are distinguished (note two instances of the label $\mathcal{L}$). In order to reduce the cost of the analysis it is possible to merge traces at some points. An example is presented in Fig. 3.14 (d), where the merge of traces is done in the label $\mathcal{L}$. As a result we do not receive false positive alarm in the point $\mathcal{L}_s$ but the analysis is still efficient.

Another method introduced by Rival and Mauborgne is a *value based paritioning*. When one focuses only on the partition based on the control flow, some interesting information about relationships between variables might get lost. To regain the relationship one can introduce partitioning according to the values of the variable at some control point.

The construction by Mauborgne and Rival is generic — it can be applied to any abstract domain. The definition of *trace paritioning abstract domain* created for some base domain includes domain operations and the widening operator that are created using the corresponding operations from the base domain. The crucial and a very difficult part of the construction is to properly choose a correct strategy for the partitioning:

- which control points to choose for the control flow partitioning,
- in which points to apply the value based partition and how to split values — the value based partitioning is domain-specific,
- and finally, which control points to choose for merging.

It seems that there is no universal solution but authors give some ideas for “good” partitions [87]:

- for sequences of conditional statements, partitioning done in the first may improve precision of the following ones when the condition of the second conditionals depends on the content of the branches created in the first one,

- unrolling loops — in some loops the first iteration includes initialization of some variables, sometimes it is worth to distinguish the first iteration, so that this is taken into account.

3.4.3 Powerset Domains

Assume that we have a *base domain*, where the join operation is imprecise. The idea of a *powerset construction* is to create a new domain, in which elements are sets of the base domain elements. This way, join can be easily implemented as a set sum of both arguments. There are two main problems with such constructions: how to efficiently represent domain elements and the implementation of the widening operator. The idea of the powerset construction was introduced by Cousot and Cousot [39]. It is based on a *down-set completion* [33].

In the description we focus on the join operation, therefore the construction uses a join-semilattice instead of a lattice.

Definition 3.4.1 (Non-redundancy). Let $\langle \mathcal{D}, \sqsubseteq_{\mathcal{D}}, \bot_{\mathcal{D}}, \sqcup_{\mathcal{D}} \rangle$ be a join-semilattice. The set $\mathcal{X} \in \mathscr{P}(\mathcal{D})$ is called *non-redundant* with respect to $\sqsubseteq_{\mathcal{D}}$ if and only if $\bot_{\mathcal{D}} \notin \mathcal{X}$ and for all $x_1, x_2 \in \mathcal{X}$ it holds that $x_1 \sqsubseteq_{\mathcal{D}} x_2 \Rightarrow x_1 =_{\mathcal{D}} x_2$.

We denote $\mathscr{P}_{fn}(\mathcal{D}_{\mathcal{D}})$ as the set of all finite subsets of $\mathcal{D}$ and $\mathscr{P}_{fn}(\mathcal{D}, \sqsubseteq_{\mathcal{D}})$ as the set of all finite non-redundant subsets of $\mathcal{D}$ with respect to $\sqsubseteq_{\mathcal{D}}$. There is a reduction function $\Omega_{\mathcal{D}}^{\sqsubseteq_{\mathcal{D}}} : \mathscr{P}_{fn}(\mathcal{D}) \to \mathscr{P}_{fn}(\mathcal{D}, \sqsubseteq_{\mathcal{D}})$, which maps any finite set $\mathcal{X} \in \mathscr{P}_{fn}(\mathcal{D})$ to its non-redundant counterpart. Let $\mathcal{X} \in \mathscr{P}_{fn}(\mathcal{D})$, then:

$$\Omega_{\mathcal{D}}^{\sqsubseteq_{\mathcal{D}}}(\mathcal{X}) = \mathcal{X} \setminus \{x \in \mathcal{X} \mid x = \bot_{\mathcal{D}} \vee \exists x' \in \mathcal{X} : x \sqsubset x'\}.$$

For any non-empty set $\mathcal{X} \in \mathscr{P}_{fn}(\mathcal{D})$ the reduction function $\Omega_{\mathcal{D}}^{\sqsubseteq_{\mathcal{D}}}(\mathcal{X})$ returns a set of maximal elements from $\mathcal{X}$ according to the ordering $\sqsubseteq_{\mathcal{D}}$ — it is sufficient as a representation of $\mathcal{X}$. This kind of reduction is the main idea behind the *down-set completion* construction (here presented for finite subsets only). It may highly reduce the size of the representation of domain elements. As an example, consider a powerset of elements the domain of signs from Section 3.3.1. The sets: $\{\pm_{Sgn}\}$, $\{\pm_{Sgn}, -_{Sgn}\}$ and $\{\pm_{Sgn}, -_{Sgn}, +_{Sgn}, 0_{Sgn}, \bot_{Sgn}\}$ have the same concrete meaning. The *down-set completion* construction removes this obvious redundancy — it is considered as a kind of partial, syntactic reduction. A complete example of a *down-set completion* for the domain of signs is presented in Fig. 3.15. Note that the lattice from Fig. 3.15(b) is another possible variant for the domain of signs.

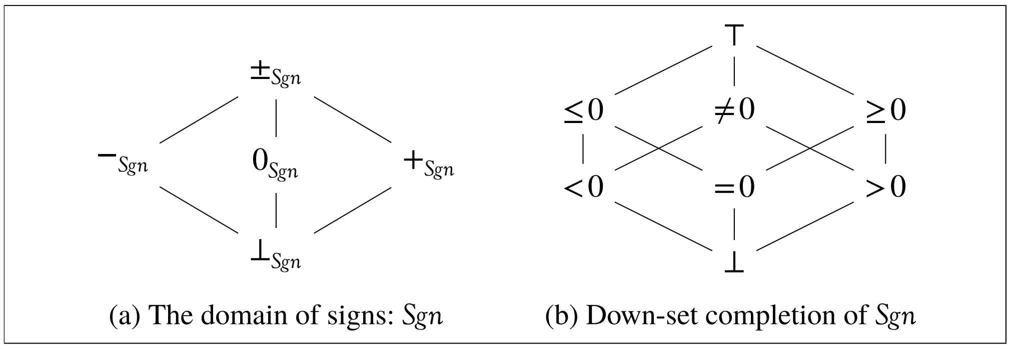


Figure 3.15: A *down-set completion* for the domain of signs Sgn.

Finite Powerset Domain Construction

The *finite powerset domain* [8, 10] is a construction that does not limit the number of disjuncts but assumes that the number is finite. The assumption is reasonable since the representation of infinite number of disjuncts may be very hard or even impossible to store in computer memory.

Let $\langle C, \sqsubseteq, \perp, \top, \sqcup, \sqcap \rangle$ be the concrete domain, where $\langle C, \sqsubseteq \rangle$ is a complete lattice, $\perp$ is the smallest element in C, $\top$ is the greatest element, $\sqcup$ is the join operator, and $\sqcap$ is the meet operator. We consider the abstract domain $\langle \mathcal{D}^\sharp, \subseteq^\sharp, \perp^\sharp, \cup^\sharp \rangle$ to be a join-semilattice and assume there is a concretisation function $\gamma : \mathcal{D}^\sharp \to C$ that is monotone and injective. For $\mathcal{X} \in \mathscr{P}_{fn}(\mathcal{D}^\sharp)$ we write $\bigcup^\sharp \mathcal{X}$ to denote the greatest upper bound of $\mathcal{X}$.

Definition 3.4.2 (Finite Powerset Domain). Let $\check{\mathcal{D}}^\sharp = \langle \mathcal{D}^\sharp, \subseteq^\sharp, \perp^\sharp, \cup^\sharp \rangle$ be a join-semilattice. A *finite powerset domain over* $\check{\mathcal{D}}^\sharp$ is the join-semilattice:

$$\check{\mathcal{D}}^\sharp_P = \langle \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp), \subseteq^\sharp_P, \perp^\sharp_P, \cup^\sharp_P \rangle,$$

where $\perp^\sharp_P = \varnothing$, the relation $\subseteq^\sharp_P$ is defined as follows:

$$\mathcal{X}_1 \subseteq^\sharp_P \mathcal{X}_2 \iff \forall x_1 \in \mathcal{X}_1 : \exists x_2 \in \mathcal{X}_2 : x_1 \subseteq^\sharp x_2$$

and $\mathcal{X}_1 \cup^\sharp_P \mathcal{X}_2 = \Omega^{\subseteq^\sharp}_{\mathcal{D}^\sharp}(\mathcal{X}_1 \cup \mathcal{X}_2)$.

Note that in the definition of $\cup_P{}^\sharp$ a regular set sum is used: we perform a syntactic normalisation of the sum of both arguments. The concretisation function $\gamma_P : \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp) \to C$ relates the finite powerset domain to the concrete domain by the function:

$$\gamma_P(\mathcal{X}) = \bigcup \{\gamma(x) \mid x \in \mathcal{X}\}.$$

Note that the function γ_P is monotone but not necessarily injective — an example is presented in Fig. 3.16 — every set in $\{\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3, \mathcal{X}_4\}$ is non-redundant with respect to $\subseteq$, they are all different and $\gamma_P(\mathcal{X}_1) = \gamma_P(\mathcal{X}_2) = \gamma_P(\mathcal{X}_3) = \gamma_P(\mathcal{X}_4)$.

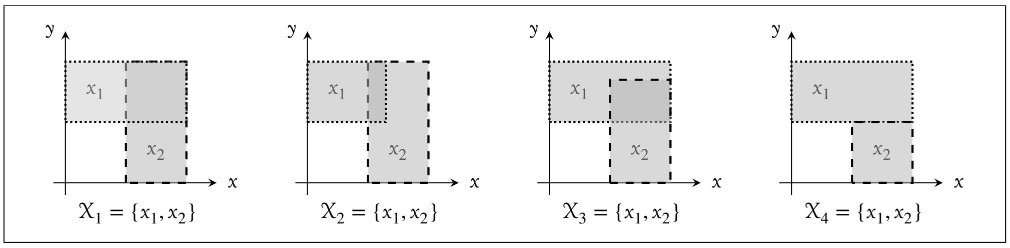

y
x
x_1
x_2
$\mathcal{X}_1 = \{x_1, x_2\}$
$\mathcal{X}_2 = \{x_1, x_2\}$
$\mathcal{X}_3 = \{x_1, x_2\}$
$\mathcal{X}_4 = \{x_1, x_2\}$

Figure 3.16: Many possible representations of the same set of concrete points by the *finite powerset domain* extension for intervals.

As we have already stated, the non-redundancy constraint provides only a partial form of reduction. It is not a full reduction because the concretisation function γ_P is not injective (example in Fig. 3.16). The full form of reduction is required by abstract interpretation since the equality comparison is required to compute the abstract semantics of a given program with either a form of Kleene's chains or with help of a widening operator.

For two elements $\mathcal{X}_1, \mathcal{X}_2 \in \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp)$ we denote $\mathcal{X}_1 \equiv_{\gamma_P} \mathcal{X}_2$ if and only if $\gamma_P(\mathcal{X}_1) = \gamma_P(\mathcal{X}_2)$. The relation $\equiv_{\gamma_P}$ is a congruence on $\mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp)$. It distinguishes abstract elements that represent the same concrete element. The direct implementation of the simple redundancy reduction function $\Omega_{\mathcal{D}^\sharp}^{\subseteq^\sharp}$ performs a quadratic (in size of the set in the argument) number of the base domain inclusion $\subseteq^\sharp$ tests. Bagnara et al. [10] do not present a generic full reduction algorithm (or an operator that checks equality). Such operation is usually implemented domain-specific and even then can be computationally very expensive [9].

The advantage of the idea by Bagnara et al. is that in some cases an abstract semantic function can be simply constructed instead of introducing ad hoc definitions. If the concrete semantic function $F : \mathcal{C} \to \mathcal{C}$ is additive, the abstract semantic function for the finite powerset domain $F_P^\sharp : \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp) \to \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp)$ can be constructed from the abstract semantic function for the base domain $F^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ as follows:

$$F_P^\sharp(\mathcal{X}) = \Omega_{\mathcal{D}^\sharp}^{\subseteq^\sharp}(\{F^\sharp(x) \mid x \in \mathcal{X}\}).$$

A different solution to decrease computational cost of introducing a disjunctive refinement is to exploit certain characteristics of the base domain — in the current thesis we apply the idea to the abstract domain of intervals.

Widening Operator

The main contribution of Bagnara, Hill, and Zaffanella [10] is the introduction of generic techniques for creating widening operators for finite powerset domains. They present a few generic constructions that use widening operators from the base domain in order to create a proper widening for the finite powerset version. These constructions are based on the structure of the powerset domain elements and orderings. They are quite complex. In order to describe their ideas we first introduce some definitions and denotations.

Bagnara et al. use a variant of the widening operator, where the second argument is greater than or equal to the first one (see Definition 2.4.7). The widening operator for the base domain $\nabla : \mathcal{D}^\sharp \times \mathcal{D}^\sharp \rightarrow \mathcal{D}^\sharp$ induces a partial ordering $\preceq_\nabla$ on $\mathcal{D}^\sharp$, which is defined as the reflexive, transitive closure of the relation:

$$\{(d_1, d_2) \in \mathcal{D}^\sharp \times \mathcal{D}^\sharp \mid \exists d \in \mathcal{D}^\sharp : d_1 \subset^\sharp d \wedge d_2 = d_1 \nabla d\}. \tag{3.1}$$

The statement $d_1 \preceq_\nabla d_2$ means that element d_1 precedes d_2 in some *iteration sequence with widening*. We define a partial order relation $\subseteq_{EM}$ on $\mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp)$ as follows:

$$\mathcal{X} \subseteq_{EM} \mathcal{Y} \iff \mathcal{X} = \bot_P^\sharp \vee (\mathcal{X} \subseteq_P^\sharp \mathcal{Y} \wedge \forall y \in \mathcal{Y} : \exists x \in \mathcal{X} : x \subseteq^\sharp y).$$

The statement $\mathcal{X} \subseteq_{EM} \mathcal{Y}$ means that either $\mathcal{X}$ is bottom or every element in $\mathcal{Y}$ is an over-approximation of some element from $\mathcal{X}$ and every element in $\mathcal{X}$ is over-approximated by some element from $\mathcal{Y}$.

Definition 3.4.3 (Extrapolation heuristics). An operator $h_P^\nabla : \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp) \times \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp) \rightarrow \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp)$ is an *extrapolation heuristics* for $\check{\mathcal{D}}$ when for any $\mathcal{X}, \mathcal{Y} \in \mathscr{P}_{fn}(\mathcal{D}, \subseteq^\sharp)$ such that $\mathcal{X} \subset_P \mathcal{Y}$, h_P^∇ is defined and satisfies the following conditions:

$$\mathcal{Y} \subseteq_{EM} h_P^\nabla(\mathcal{X}, \mathcal{Y}), \tag{3.2}$$
$$\forall x \in h_P^\nabla(\mathcal{X}, \mathcal{Y}) \setminus \mathcal{Y} : \exists x' \in \mathcal{X} : x' \prec_\nabla x. \tag{3.3}$$

The condition (3.2) ensures that the result is an over-approximation of $\mathcal{Y}$ — every element in the result is an over-approximation of some element in $\mathcal{Y}$. Thus, the *extrapolation heuristics* cannot introduce any element that is unrelated to $\mathcal{Y}$. The condition (3.3) ensures that every element in the result set, that was not in $\mathcal{Y}$, is obtained by an application of the widening operator to some element of $\mathcal{X}$ (not necessarily once).

The *extrapolation heuristics* itself is not enough for the widening. As an example, we take a one-dimensional domain of intervals. We define a sequence of elements $\mathcal{T}_0, \mathcal{T}_1, \dots$ such that $\mathcal{T}_j = \{[i,i] \mid 0 \leq i \leq j\}$. The element $\mathcal{T}_j$ consists of $j+1$ points. Note that by (3.2) it holds that $h^{\nabla}_{\mathcal{P}}(\mathcal{T}_j, \mathcal{T}_{j+1}) = \mathcal{T}_{j+1}$. Therefore, for any definition of the base widening the sequence of applications of the *extrapolation heuristics* operator is diverging.

The *extrapolation heuristics* is the base for creating generic widening constructions. It already gives some intuition about the constructions of these widening operators: in the following constructions we control the origin of elements that appear during the widening.

Powerset Widenings using Set Cardinality

The first method of creating widening operators is to control the number of disjuncts during the widening sequence. For that purpose a *k-collapsor* unary operator for $\check{\mathcal{D}}^\sharp$ is introduced.

Definition 3.4.4 (k-collapsor). For $k \geq 1$, a unary operator $\Uparrow_k\colon \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp) \to \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp)$ is called a *k-collapsor* for $\check{\mathcal{D}}^\sharp$ if for every $\mathcal{X} \in \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp)$ it holds that if $|\mathcal{X}| \leq k$ then $\Uparrow_k(\mathcal{X}) = \mathcal{X}$ or when $|\mathcal{X}| > k$ then there is $\mathcal{Y} \subseteq \mathcal{X}$ such that $|\mathcal{X}| - k < |\mathcal{Y}|$ and $\Uparrow_k(\mathcal{X}) = (\mathcal{X} \setminus \mathcal{Y}) \cup_{\mathcal{P}} \{\bigcup^\sharp \mathcal{Y}\}$.

The k-collapsor ensures that the result has at most k elements. If the argument $\mathcal{X}$ has more than k elements, the k-collapsor replaces a subset $\mathcal{Y}$ of elements of $\mathcal{X}$ by their join. The number of elements that are left in the set $\mathcal{X} \setminus \mathcal{Y}$ is smaller than k.

The k-collapsor itself also does not ensure the termination [10, Example 3] (an example is quite complex, thus we do not present it here). The problem is that the reduction function $\Omega^{\subseteq^\sharp}_{\mathcal{D}^\sharp}$ somehow interferes with the cardinality control mechanism. A solution is to add some additional constraints on the extrapolation heuristics:

Definition 3.4.5 ($\triangledown$-covered heuristics). The extrapolation heuristics $h^{\nabla}_{\mathcal{P}}$ is called $\triangledown$*-covered* if for all $\mathcal{X}, \mathcal{Y} \in \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp)$ such that $\mathcal{X} \subset_{\mathcal{P}} \mathcal{Y}$ it holds that:

$$\forall x \in \mathcal{X} : \exists z \in h^{\nabla}_{\mathcal{P}}(\mathcal{X}, \mathcal{Y}) : x \preceq_{\triangledown} z.$$

Now every element x in the first argument must be $\triangledown$-*covered* by some element z from the result: the element z is obtained by an application of the widening operator to x (possibly multiple times). The *cardinality-based widening* is a combination of the $\triangledown$-covered heuristics and the k-collapsor:

Theorem 3.4.6 (Cardinality-based widening). *Let $h^{\nabla}_{\mathcal{P}}$ be a $\triangledown$-covered extrapolation heuristics and $\Uparrow_k$ be a k-collapsor for $\check{\mathcal{D}}$ and some $k \geq 1$. Let $\triangledown_{k,\mathcal{P}}$:*

$\mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp) \times \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp) \rightharpoonup \mathscr{P}_{fn}(\mathcal{D}^\sharp)$ *be a partial operator such that for any* $\mathcal{X}, \mathcal{Y} \in \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp)$ *it holds that* $\mathcal{X} \sqsubset_P^\sharp \mathcal{Y}$. *Then:*

$$\mathcal{X} \nabla_{k,P} \mathcal{Y} \stackrel{\text{def}}{=} h_P^\nabla(\mathcal{X}, \Uparrow_k(\mathcal{Y}))$$

is a widening operator for $\check{\mathcal{D}}$. *The operator is called* cardinality-based widening.

Proof. In [10]. □

The precision of the *cardinality-based widening* depends on quite a lot of factors. There is a generic construction of a ∇-covered heuristics that can be obtained for any base-level widening operator [10, Def. 9]. Still, one has to adjust the k-collapsor operator. First of all, it depends on the parameter k, which controls the number of disjuncts. The k-collapsor is also responsible for the choice of a subset of elements to join together.

Powerset Widenings using Connectors

Another method of obtaining a generic widening operator proposed by Bagnara et al. is to ensure that the base-level widening is used. We define a new subclass of extrapolation heuristics:

Definition 3.4.7 (∇-connected heuristics). The extrapolation heuristics h_P^∇ is said to be ∇-connected if for all $\mathcal{X}, \mathcal{Y} \in \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp)$ such that $\mathcal{X} \sqsubset_P^\sharp \mathcal{Y}$ it holds that:

$$\forall z \in h_P^\nabla(\mathcal{X}, \mathcal{Y}) \cap \mathcal{Y} : (\exists x \in \mathcal{X} : x \subset^\sharp z) \Rightarrow (\exists x' \in \mathcal{X} : x' \prec_\nabla z).$$

The ∇-connected heuristics makes sure that every element in the result that *covers* (i.e. is greater than) some element of the first argument $\mathcal{X}$ originates from an application of the widening operator to an element of $\mathcal{X}$. Note it might be a different element of $\mathcal{X}$ than the one that is covered.

We recall the simple example, which exposed that the extrapolation heuristics is not a widening operator. The example still applies for the ∇-connected heuristics. The divergence is caused by elements in the second argument that do not *cover* elements from the first argument. The idea is to replace the second argument by an upper bound of both arguments in the $\subseteq_{EM}$ order (any upper bound):

Definition 3.4.8 (Connector). An operator $\sqcup_{EM} : \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp) \times \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp) \to \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp)$ is called a *connector* for $\check{\mathcal{D}}^\sharp$ if it is an upper bound operator for $\subseteq_{EM}$, that is for any $\mathcal{X}, \mathcal{Y} \in \mathscr{P}_{fn}(\mathcal{D}^\sharp, \subseteq^\sharp)$ it holds that $\mathcal{X} \subseteq_{EM} (\mathcal{X} \sqcup_{EM} \mathcal{Y})$ and $\mathcal{Y} \subseteq_{EM} (\mathcal{X} \sqcup_{EM} \mathcal{Y})$.

A combination of ∇-connected extrapolation heuristics with the connector can be used to create a widening operator, which is stated by Theorem 3.4.9.

Theorem 3.4.9 (Connector-based widening). *Let h_P^{∇} be a ∇-connected extrapolation heuristics and $\sqcup_{EM}$ be a connector for $\check{\mathcal{D}}^{\sharp}$. Let also $\mathcal{X}, \mathcal{Y} \in \wp_{fn}(\mathcal{D}^{\sharp}, \sqsubseteq^{\sharp})$ be such that $\mathcal{X} \sqsubset_P^{\sharp} \mathcal{Y}$. Then a partial operator $\nabla_{EM,P} : \wp_{fn}(\mathcal{D}^{\sharp}, \sqsubseteq^{\sharp}) \times \wp_{fn}(\mathcal{D}^{\sharp}, \sqsubseteq^{\sharp}) \rightharpoonup \wp_{fn}(\mathcal{D}^{\sharp}, \sqsubseteq^{\sharp})$ such that $\mathcal{X} \nabla_{EM,P} \mathcal{Y} \stackrel{\text{def}}{=} h_P^{\nabla}(\mathcal{X}, \mathcal{Y}')$, where:*

$$\mathcal{Y}' = \begin{cases} \mathcal{Y} & \textit{if } \mathcal{X} \sqsubseteq_{EM} \mathcal{Y} \\ \mathcal{X} \sqcup_{EM} \mathcal{Y} & \textit{otherwise}, \end{cases}$$

is a widening operator for $\check{\mathcal{D}}$. The operator is called connector-based widening.

Proof. In [10]. □

Powerset Widenings using Certificates

Bagnara et al. also introduce a *certificate-based widening operator*. The construction is a bit different from the constructions based on connectors and cardinality that we have presented, but it is even more complicated. Authors claim that the certificate-based widening construction is not a good choice for powersets of simpler domains such as intervals, therefore we do not present the technique in detail here. The widening operator is based on a notion of a *convergence certificate*:

Definition 3.4.10 (Finite convergence certificate). A *finite convergence certificate* for an upper bound operator $\sqcup$ on $\check{\mathcal{D}}^{\sharp}$ is a triple $\langle \mathcal{O}, \preceq, \mu \rangle$, where $\langle \mathcal{O}, \preceq \rangle$ is a partially ordered set satisfying the ascending chain condition and $\mu : \mathcal{D} \to \mathcal{O}$ (called a *level mapping function*) is such that:

$$\forall x, x' \in \mathcal{D} : x \sqsubset^{\sharp} x' \Rightarrow \mu(x) \prec \mu(x \sqcup x').$$

The *certificate based widening operator* is created from a *finitely computable certificate* for the base-domain. The crucial part of the construction is to create, from the certificate and the widening operator for the base-domain, a relation on the powerset domain that is both finitely computable and satisfies the ascending chain condition. Bagnara et al. present a generic construction of such relation — $\curvearrowright_P$. Together with any *upper bound operator* on the powerset domain, the relation $\curvearrowright_P$ is used to build a widening operator for the powerset domain.

Conclusion

We have presented a brief overview of three generic constructions of widening operators that can be applied for powerset domains. All the constructions use the widening operator from the base domain. The main drawback of these constructions is their complexity — the blocks required to build each of presented widenings are listed in Table 3.1. The level of complexity and various constraints of all three constructions makes creating a reasonable widening operator very hard. On the other hand, the generality of these constructions enables to create a widening for a powerset domain possibly without deeper understanding of the underlying base domain. Bagnara et al. have presented applications of the method for the powerset of the domain of polyhedra. Their widening operators turned out to be better that known before. We have to note that the domain of polyhedra is quite complex and creation of a widening that takes advantage of its specific features is very hard.

Widening	**Required constructions**
Cardinality-based	∇-covered heuristics, k-collapsor
Connector-based	∇-connected heuristics, connector operator
Certificate-based	convergence certificate, upper bound operator, substraction operator

Table 3.1: Operators required to apply presented widening constructions.

In case of simpler domains, as domain of intervals, the powerset domain is more natural. The simplicity of the base domain gives us possibility to create our own implementation of the powerset version (possibly with a full reduction) along with a specific construction of the widening operator that exploits some of the features of the domain. In the construction presented in the thesis we apply this approach in order to create an implementation of the powerset of the domain of intervals along with the widening operator.

3.4.4 Other techniques

There are also other techniques to introduce path-sensitive analysis. For example, Sankaranarayanan et al. [88] introduce a concept of *elaboration*. The idea is similar to the trace-partitioning technique by Mauborgne and Rival. One creates a modified control flow graph, which is constructed as an extension of the original graph by replicating some of the nodes. As an application of the technique Sankaranarayanan et al. consider a *bounded elaboration*, which corresponds to powerset extensions, where the number of disjuncts is limited by some constant.

Chapter 4

Generic Disjunctive Refinement for the Domain of Intervals

4.1 Introduction

In this chapter, we present an adaptation of the concept of the sweeping line to abstract interpretation. Sweeping line algorithms are very important in the computational geometry. They are used to compute all crossings in a set of line segments (Bentley-Ottmann algorithm [12]) or a construction of the Voronoi diagram (Fortune's algorithm [52]). We demonstrate here how the sweeping line technique can be used to efficiently represent elements and perform operations for a domain that is a disjunctive refinement of the domain of *intervals*. Our format handles both strict and non-strict inequalities. We also give some ideas for optimisations of the presented base version, which may reduce the size of the representation and the cost of domain operations. We apply *thresholds* to the construction of the widening operator for the domain. We introduce two versions of the widening operator: a generic one, and the second one with a theorem about one-step precision of the operator depending on the choice of *threshold* points.

4.1.1 Related Work

Abstract interpretation successfully takes advantage of techniques used in various fields. There have been works, which employ different graph-based algorithms [53, 83]. Also lately *quadtrees*, a data structure used in computational geometry was proposed to be adapted for the abstract interpretation [68]. The current research investigates advantages and disadvantages of a new technique in the field of abstract interpretation — the *sweeping line technique* [12], which is one of the key techniques of the computational geometry. The introduction of the new concept brings new insights. We use the concept of a sweeping line to represent elements of the domain of *boxes*, which is a disjunctive refinement of the domain of *intervals*.

There have been proposed some ways to build a disjunctive refinement. We have briefly presented a few techniques in Section 3.4. They usually use special strategies of controlling the disjuncts [10, 79, 88], but most of them do not scale to a large number of disjuncts. Also achieving a satisfactory precision of the widening is hard. Recently, a new implementation of the domain of *boxes* has been proposed [60]. The solution by Gurfinkel and Chaki is based on Linear Decision Diagrams (LDD) and easily scales to large number of disjuncts. Additionally, quite high precision of widening was presented. We propose here a different approach to the same domain of *boxes*. Our construction is more generic, the implementation of the domain that uses LDDs can be regarded as an optimisation of the technique presented in the current thesis. The widening operator introduced here uses *thresholds* [14] to gain precision. Single application of the widening operator gives more precise result than the one by Gurfinkel and Chaki.

4.2 Problem Definition

The basic version of the domain of *intervals* makes it possible to represent only convex sets (as described in Section 3.4). We would like to extend the domain to make it possible to represent finite disjunctions of *intervals*. We define this more formally in what follows. First, we define the new numerical abstract domain and then introduce representation of domain elements, algorithms for domain operators and widening operator as well as transfer functions.

We extend the construction from Section 3.3.2 to represent finite sets of elements from the domain of *intervals*. The domain of *boxes* is a tuple:

$$\langle \mathbb{BS}_n, \subseteq, \emptyset, \mathbb{I}^n, \cup, \cap \rangle,$$

where:

- $\mathbb{BS}_n = \{\mathcal{BS} \in \mathbb{I}^n \mid \text{ there exist } \mathcal{B}_1, \mathcal{B}_2, \dots, \mathcal{B}_k \in \mathbb{B}_n \text{ such that } \mathcal{BS} = \bigcup_{i=1}^{k} \mathcal{B}_i\}$,
- $\subseteq$ is the subset ordering,
- $\emptyset$ is the empty set that is also $\bot$ of the ordering,
- $\mathbb{I}^n$ is the whole n-variable space, which is the $\top$ of the ordering,

- $\cup$ is the join operator for the domain, which is an exact sum of two elements,
- $\cap$ is the meet operator for the domain, which is an exact intersection.

The improvement compared to the domain of *intervals* is that in the domain of *boxes* the operation $\cup$ is exactly set theoretical sum and does not require any kind of closure. Note that for elements $\mathcal{BS} \in \mathbb{BS}_n$ there may exist multiple splits into separate boxes from the domain of *intervals*.

Our goal is to efficiently represent elements of the domain of *boxes* and efficiently perform domain operations on such elements. With the chosen technique we directly describe element $\mathcal{BS} \in \mathbb{BS}_n$, not as a set of separate boxes. However, in some situations we require to split the element into separate boxes. Then, we choose only one "unique" split that is maximal in some sense, which is a consequence of the chosen representation.

Interval Constraints

The domain of *intervals*, as presented in Section 3.3.2, uses only non-strict inequalities. Since we would like to accept both strict and non-strict inequalities, we introduce a few definitions that are used in the presented construction.

Let $\mathcal{B} \in \mathbb{B}_n$ for some $n \geq 0$ be a single box from the domain of *intervals*. Let $\mathsf{Var} = \{v_1, \dots, v_n\}$ be the set of variables. The element $\mathcal{B}$ is either empty, i.e. $\bot_v$, or it is defined by a finite set C of *interval constraints* such that:

- every constraint $c \in \mathrm{C}$ limits only one variable,
- there are exactly two constraints that limit every variable $v \in \mathsf{Var}$:
  - a *lower boundary interval constraint* c_L for variable v — an inequality of form $c \equiv a < v$, where $a \in \mathbb{I} \cup \{-\infty\}$, or $c \equiv a \leq v$, where $a \in \mathbb{I}$,
  - a *higher boundary interval constraint* c_H for variable v — an inequality of form $c \equiv v < b$, where $b \in \mathbb{I} \cup \{\infty\}$, or $c \equiv v \leq b$, where $b \in \mathbb{I}$,
  - both constraints define a non-empty set of possible values for v, i.e. if either of c_L or c_H is a non-strict inequality then it holds that $a < b$ or $a \leq b$ otherwise.

If $c_L \equiv -\infty < v$ we say there is no lower boundary interval constraint on the variable v. Dually, if $c_H \equiv v < \infty$ we say there is no higher boundary interval constraint on the variable v. If there is no lower boundary interval constraint and no higher boundary interval constraint for v, we say that there are no interval restrictions on v. We say that $\vec{x} \in \mathbb{I}^n$ such that $\vec{x} = \langle x_n, \dots, x_1 \rangle$ fulfils interval constraints defining $\mathcal{B}$ if for every $i \in \{1, \dots, n\}$ it holds that x_i fulfils interval constraints for the variable v_i.

Since every constraint $c \in C$ limits only one variable, constraints for different variables are independent, which is stated by Remark 4.2.1.

Remark 4.2.1. Let $n > 0$, $\vec{x} \in \mathbb{I}^n$ be such that $\vec{x} = \langle x_n, \dots, x_1 \rangle$, $\mathcal{B} \in \mathbb{B}_n$ and $\mathsf{Var} = \{v_1, \dots, v_n\}$ be the set of variables. It holds that:

$$\begin{aligned}&\vec{x} \text{ fulfils all interval constraints defining } \mathcal{B} \iff \\ &\qquad\qquad \forall i \in \{1, \dots, n\} : x_i \text{ fulfils interval constraints for variable } v_i.\end{aligned}$$

4.3 Adaptation of the Sweeping Line Technique

The general idea of the sweeping line technique uses a conceptual *sweeping line* or *sweeping surface* to solve various problems in Euclidean space. It is one of the key techniques used in the computational geometry. The idea behind algorithms of this type is to imagine that a line (usually a vertical one) is swept across a plane, stopping at some points. A data structure, which is associated with the sweeping line, is updated every time the line is stopped. When the line has swept across all the points, a result of interest is computed from the final data structure.

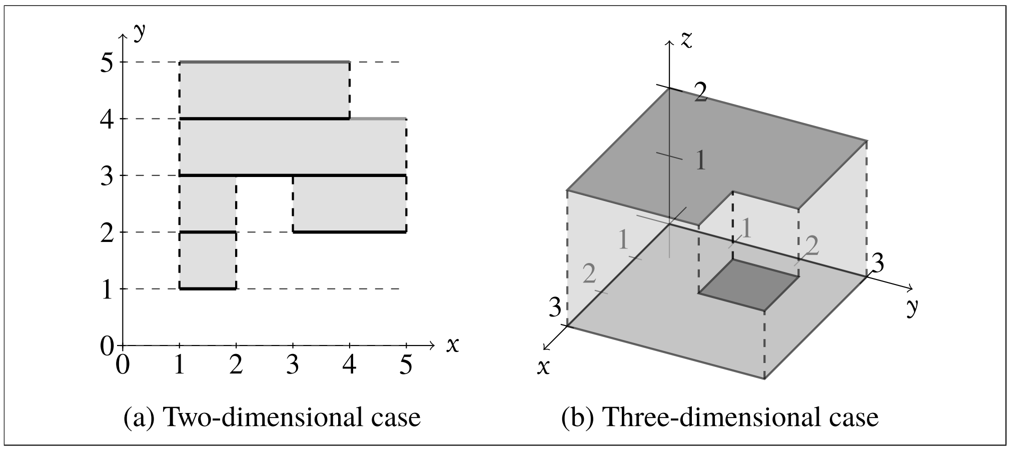


Figure 4.1: The idea of the way the sweeping line technique is used to represent domain elements.

To demonstrate the idea how to adapt the technique to construct the domain of *boxes*, we describe two and three-dimensional examples presented in Fig. 4.1. Let us assume that we already have a representation of the one-dimensional version of the domain, which is a set of possible values for a single variable x. Now, let us focus on the two-dimensional version presented in Fig. 4.1(a). We sweep through the values of the variable y starting from $-\infty$ to $+\infty$, and observe what happens with values of the variable x. As the data associated with the sweeping line — SL_1, we store the set of values of the variable x that are possible for the current value of y. This is in fact a one-dimensional version of the domain. We stop at values of the variable y, where the set changes, and update the data structure SL_1 accordingly. Consider the example in Fig. 4.1(a). The set of stop points for the variable y is $y \in \{-\infty, 1, 2, 3, 4\}$. For every such value the set of corresponding values of the variable x is displayed in black. The process of sweeping is summarized in Table 4.1.

Sweeping line location (value of y)	**Data associated with the sweeping line: SL_1 (possible values for x)**
$y = -\infty$	$\emptyset$
$y = 1$	$[1, 2]$
$y = 2$	$[1, 2] \cup [3, 5]$
$y = 3$	$[1, 5]$
$y = 4$	$[1, 4]$
$y = 5$	$\emptyset$

Table 4.1: Details of the sweeping process for the example from Fig. 4.1(a).

We proceed analogically for the three-dimensional version from Fig. 4.1(b). When we sweep through the values of the variable z, the data associated with the sweeping line — SL_2, is a two-dimensional version of the domain. Thus, for $z < 0$ we have that SL_2 is empty, for $z = 0$ it becomes an area described by the following two-dimensional representation: for $y < 0$ the structure SL_1 is empty, for $y = 0$ it is changed to an interval $[0, 3]$, and for $y > 3$ the data SL_1 becomes empty again. For $z = 1$ the area described by SL_2 becomes a little more complicated (a small square is removed) and for $z > 2$ the structure SL_2 becomes empty.

The representation of elements of the domain of *boxes* is based on the technique described above. The difference is that we actually collect whole history of sweeping. For example, for the sweeping process from Fig. 4.1(a) we collect all the data from Table 4.1. We obtain a list of pairs, where first elements are values of the variable y and second elements are possible values for the variable x.

4.4 Representation of Domain Elements

In order to manage both strict and non-strict inequalities, special points that are encountered during the process of sweeping are described as pairs $(x, b) \in \mathbb{I} \times \mathbb{PM}$, where $\mathbb{PM} = \{\oplus, \ominus\}$. Each such pair describes the beginning of an interval. In case $(x, \oplus)$ the number x is included in the interval and in case $(x, \ominus)$ the value x is excluded from it. We define the set of pairs as follows:

$$\mathbb{P} = \mathbb{I} \times \mathbb{PM}.$$

Additionally, when $\mathbb{I} = \mathbb{Z}$ we add a restriction that $\ominus$ is not used. We introduce an ordering on elements of $\mathbb{P}$, that is $\prec \subseteq \mathbb{P} \times \mathbb{P}$, defined as:

$$(x, b) \prec (x', b') \iff x < x' \text{ or } x = x' \land b = \oplus \land b' = \ominus \tag{4.1}$$

and $\preceq \subseteq \mathbb{P} \times \mathbb{P}$, which is the reflexive closure of $\prec$. Note that these are lexicographical orderings on pairs in case when $\mathbb{PM}$ is ordered as $\oplus < \ominus$. When $\mathbb{I} = \mathbb{Z}$, since second elements of pairs are always $\oplus$, the ordering is isomorphic to the ordering on first elements of pairs only. Unfortunately, the set $\mathbb{P}$ is not sufficient to describe all possible beginnings of intervals. Therefore, we define $\mathbb{P}_\infty = \mathbb{P} \cup \{-\infty\}$ and we extend the ordering $\prec$ to an ordering on $\mathbb{P}_\infty$ in the natural fashion, so that $\forall p \in \mathbb{P} : -\infty \prec p$. The ordering $\preceq$ is extended analogously.

We introduce a relation $in \subseteq \mathbb{P}_\infty \times \mathbb{I} \times \mathbb{P}$ defined as follows:

$$in(p, x, p') \iff p \preceq (x, \oplus) \prec p'. \tag{4.2}$$

Intuitively, it states that $x \in \mathbb{I}$ belongs to the interval described by p and p', where p is the beginning of the interval and p' is the first element of $\mathbb{P}_\infty$ that does not belong to the interval. For example:

- the interval $(-\infty, 10)$ is represented by a pair $\langle -\infty, (10, \oplus)\rangle$,
- the interval $[3, 7)$ is represented by a pair $\langle (3, \oplus), (7, \oplus)\rangle$,
- the interval $(3, 7]$ is represented by:
  - a pair $\langle (3, \ominus), (7, \ominus)\rangle$ if $\mathbb{I} \in \{\mathbb{R}, \mathbb{Q}\}$,
  - or by a pair $\langle (4, \oplus), (8, \oplus)\rangle$ if $\mathbb{I} = \mathbb{Z}$.

  In the second case we cannot use $\ominus$, therefore we add 1 to the first coordinate describing the interval. This is similar to regular intervals, i.e. the interval $(3, 7]$ represents the same set of integer values as the interval $[4, 8)$.

Note that the second element p' does not describe the end of the interval directly. This may seem odd but such approach is more appropriate for describing partitions of $\mathbb{I}$, which we aim to do. Also, note that the interval $(4, +\infty)$ cannot be described by the relation in. This is not a problem for us since we will not use the relation in to describe the last interval in the partition.

Lemma 4.4.1 states that any two different elements in $\mathbb{P}_\infty$ and the relation *in* can be used to describe a non-empty interval, i.e. that the interval contains at least one element from $\mathbb{I}$. This is important, since we use a subset of $\mathbb{P}_\infty$ to describe partitions of $\mathbb{I}$ and every two adjacent elements in the subset can be used to describe a proper interval.

Lemma 4.4.1 (Density). *For all $p, p' \in \mathbb{P}_\infty$, if $p \prec p'$ then there is $i \in \mathbb{I}$ such that $in(p, i, p')$.*

Proof. If $p = -\infty$ then by the extended definition of $\prec$ we have that $p' \in \mathbb{P}$, hence $p' = (j', b')$ for some $j' \in \mathbb{I}$ and $b' \in \mathbb{PM}$. Then it holds that $p = -\infty \prec (j' - 1, \oplus)$. Additionally, $(j' - 1, \oplus) \prec (j', b') = p'$, therefore $-\infty \preceq (j' - 1, \oplus) \prec p'$, which means that $in(p, j' - 1, p')$ and $i = j' - 1$ meets desired criteria.

Otherwise, if $p \neq -\infty$ then $p = (j, b)$ and $p' = (j', b')$ for some $j, j' \in \mathbb{I}$ and $b, b' \in \mathbb{PM}$. If $b = \oplus$ then $p \preceq (j, b) \prec p'$, hence $in(p, j, p')$ and we choose $i = j$. Otherwise, if $b = \ominus$ then by the definition of $\prec$ we have that $j < j'$. Additionally, $\ominus$ was not allowed for $\mathbb{I} = \mathbb{Z}$, therefore $\mathbb{I} = \mathbb{R}$ or $\mathbb{I} = \mathbb{Q}$. Then there is $i \in \mathbb{I}$ such that $j < i < j'$. By the definition of $\prec$ it holds that $p = (j, b) \prec (i, \oplus)$ and $(i, \oplus) \prec (j', b') = p'$, thus $in(p, i, p')$. $\square$

We use the ordering $\prec$ as the base to construct a representation for elements of *boxes*. Let us define an infinite sequence of sets $\mathbb{S}_0, \mathbb{S}_1, \ldots$ as follows:

$$\begin{aligned}
\mathbb{S}_0 &= \{\epsilon, \top_0\}, \\
\mathbb{S}_{n+1} &= \{\epsilon\} \cup \{\big((p_1, v_1), (p_2, v_2), \ldots, (p_m, v_m)\big) \mid v_1, \ldots, v_m \in \mathbb{S}_n, v_1 \neq \epsilon, \\
&\qquad p_1, \ldots, p_m \in \mathbb{P}_\infty, \forall_{j \in \{1, \ldots, m-1\}} p_j \prec p_{j+1} \wedge v_j \neq v_{j+1}\},
\end{aligned} \tag{4.3}$$

where ϵ is the empty sequence. Every sequence $S_n \in \mathbb{S}_n$ for $n > 0$ represents data collected by sweeping through the space of the n-th variable. The sequence uniquely describes a segmentation of the space for the variable so that every segment has a value in $\mathbb{S}_{n-1}$. Examples of such sequences are illustrated in Fig. 4.2.

First elements of pairs in a sequence $S \in \mathbb{S}_{n+1}$ for $n \geq 0$ describe special points encountered during the process of sweeping through the $(n+1)$-st dimension. The set $\{p_1, \ldots, p_m\}$ defines the partition of the space for the $(n+1)$-st variable. Values $v_1, \ldots, v_m$ correspond to the values of the n-dimensional space encountered while sweeping through values of $(n + 1)$-st variable. The restriction that $\ominus$ is not used for $\mathbb{I} = \mathbb{Z}$ in the description of special points is introduced, because we want the representation of domain elements to be unique — so that each domain element has only one possible representation. This can be easily achieved and it simplifies domain operations.

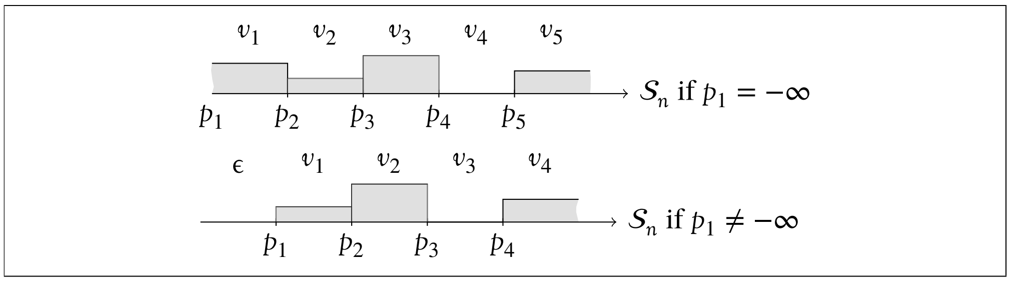

v_1 v_2 v_3 v_4 v_5
p_1 p_2 p_3 p_4 p_5
$\mathcal{S}_n$ if $p_1 = -\infty$
ϵ v_1 v_2 v_3 v_4
p_1 p_2 p_3 p_4
$\mathcal{S}_n$ if $p_1 \neq -\infty$

Figure 4.2: Example sequences and corresponding segmentations of the space of the n-th variable.

We define a function *find* $: \mathbb{I} \times \bigcup_{n>0} \mathbb{S}_n \to \mathbb{N}$ such that for any $i \in \mathbb{I}$, $n \geq 1$ and $\mathcal{S} \in \mathbb{S}_n$ it holds that:

$$\mathit{find}(i, \mathcal{S}) = \begin{cases} 0 & \text{if } \mathcal{S} = \epsilon \text{ or } \mathcal{S} = \big((p_1, v_1), \dots, (p_m, v_m)\big) \text{ and } (i, \oplus) \prec p_1 \\ k & \text{if } \mathcal{S} = \big((p_1, v_1), \dots, (p_m, v_m)\big) \text{ and} \\ & \qquad k = \max(j \mid j \in \{1, \dots, m\} \text{ and } p_j \preceq (i, \oplus)). \end{cases} \tag{4.4}$$

The function gives the index of the pair in the sequence $\mathcal{S}$ that corresponds to the interval in the partition that contains i. If the sequence is empty or i is before the first value in the partition — p_1, then the function returns 0. If $\mathit{find}(i, \mathcal{S}) = k$ and $0 < k < m$ then $\mathit{in}(p_k, i, p_{k+1})$. At last, when i belongs to the last interval in the partition then *find* outputs m.

The function *find* is well-defined, which is a consequence of the fact that first elements of pairs in the sequence $\mathcal{S} \in \mathbb{S}_n$ form a partition of the space of the n-th variable. Lemma 4.4.2 states this in a more detailed fashion.

Lemma 4.4.2 (*find* is Well-defined Function). *For all $i \in \mathbb{I}$, $\mathcal{S} \in \mathbb{S}_n$ and $n > 0$ there is only one $k \in \mathbb{N}$ such that $\mathit{find}(i, \mathcal{S}) = k$.*

Proof. The lemma holds trivially for $\mathcal{S} = \epsilon$. From now on we assume that $\mathcal{S} \neq \epsilon$. Let $\mathcal{S} = \big((p_1, v_1), \dots, (p_m, v_m)\big) \in \mathbb{S}_n$ for some $n > 0$, $m > 0$ and $i \in \mathbb{I}$. Assume there are indexes $k, k' \in \mathbb{N}$ such that $k \neq k'$, $\mathit{find}(i, \mathcal{S}) = k$ and $\mathit{find}(i, \mathcal{S}) = k'$. There are three possibilities:

- If $k = m$, then by the definition of *find* the only option is that $p_m \preceq (i, \oplus)$, therefore also $k' = m$.
- If $k = 0$ then $k' = 0$.
- Both $0 < k < m$ and $0 < k' < m$. Without loss of generality we assume that $k < k'$. By the definition of $\mathbb{S}_n$ it holds that $p_k \prec p_{k'}$ and $p_{k+1} \prec p_{k'+1}$. By the

definition of *find* we obtain $p_{k'} \preceq (i, \oplus)$, therefore $p_k \prec p_{k'} \preceq (i, \oplus)$. Also, by the definition of *find*, it holds that $p_k \preceq (i, \oplus)$ but $p_{k+1} \npreceq (i, \oplus)$. Since $\preceq$ is a total order, it holds that $(i, \oplus) \preceq p_{k+1}$. Thus, we obtain:

$$p_k \prec p_{k'} \preceq (i, \oplus) \prec p_{k+1} \prec p_{k'+1}.$$

So between elements p_k and p_{k+1} in the sequence $\mathcal{S}$ there should be another element $p_{k'}$, which is a contradiction.

In all the examined cases we have obtained a contradiction, therefore *find* is a well-defined function. □

The function *find*$(i, \mathcal{S})$ gives the index of the interval in the partition by the first elements of pairs in $\mathcal{S}$ that contains i. We are usually interested in the value corresponding to the interval that contains i, not the index of the interval. Here we introduce an auxiliary *access notation* that can be used to directly get the value associated with the correct interval:

$$\mathcal{S}[i] = \begin{cases} \epsilon & \text{if } \mathit{find}(i, \mathcal{S}) = 0 \\ v_{\mathit{find}(i,\mathcal{S})} & \text{otherwise.} \end{cases} \tag{4.5}$$

The value $\mathcal{S}[i]$ is ϵ when the sequence $\mathcal{S}$ is empty or when i is before the first special point in the sequence $\mathcal{S}$. Otherwise we use *find* to search for the index of the correct pair in the sequence $\mathcal{S}$ and return the value, i.e. the second element of the pair at the index.

Now we define a relation that holds when an element of $\mathbb{I}^n$ belongs to the domain element represented by $\mathcal{S} \in \mathbb{S}_n$. In order to simplify the definition we assume that $\mathbb{I}^0 = \{\vec{\epsilon}\}$. The *satisfiability relation* $sat_n \subseteq \mathbb{I}^n \times \mathbb{S}_n$ is defined for $n \geq 0$ as follows:

$$\begin{aligned} sat_n(\vec{\epsilon}, \mathcal{S}) &\iff \mathcal{S} \in \mathbb{S}_n \wedge \mathcal{S} = \top_0, \\ sat_{n+1}(\langle i_{n+1}, i_n, \ldots, i_1 \rangle, \mathcal{S}) &\iff \mathcal{S}[i_{n+1}] = w \wedge sat_n(\langle i_n, \ldots, i_1 \rangle, w). \end{aligned} \tag{4.6}$$

Here we present a few examples of sequences for the one-dimensional space (from $\mathbb{S}_1$) and equivalent intervals:

Sequence from $\mathbb{S}_1$	**Equivalent intervals in $\mathbb{R}$**
ϵ	$\varnothing$
$\big(((0,\oplus),\top_0)\big)$	$[0,\infty)$
$\big(((0,\oplus),\top_0),((1,\oplus),\epsilon)\big)$	$[0,1)$
$\big(((0,\oplus),\top_0),((1,\ominus),\epsilon)\big)$	$[0,1]$
$\big((-\infty,\top_0),((0,\oplus),\epsilon),((3,\ominus),\top_0)\big)$	$(-\infty,0)\cup(3,\infty)$
$\big((-\infty,\top_0),((0,\ominus),\epsilon),((2,\oplus),\top_0),((5,\oplus),\epsilon)\big)$	$(-\infty,0]\cup[2,5)$

With the help of the *sat* relation we define the concretisation function $\gamma : \mathbb{S}_n \to \mathscr{P}(\mathbb{I}^n)$ for any $n > 0$. For any domain element $\mathcal{S} \in \mathbb{S}_n$ the function is defined as follows:

$$\gamma(\mathcal{S}) = \{\vec{i} \in \mathbb{I}^n \mid sat_n(\vec{i}, \mathcal{S})\}. \tag{4.7}$$

The relation $sat(\vec{i}, \mathcal{S})$ holds only for vectors $\vec{i} \in \mathbb{I}^n$ that belong to the representation $\mathcal{S} \in \mathbb{S}_n$, i.e. for vectors that are in the element of *boxes* that corresponds to $\mathcal{S}$. This is proved in what follows.

Theorem 4.4.3 states that the proposed representation is unique in terms of the *sat* relation. Therefore, the representation proposed in the thesis does not require the non-redundancy reduction of the representation, which is one of the drawbacks of the generic powerset construction presented in Section 3.4.3.

Theorem 4.4.3 (Uniqueness of Representation). *For all* $n \geq 0$, $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_n$, *if* $\mathcal{S} \neq \mathcal{S}'$ *then there is* $\vec{i} \in \mathbb{I}^n$ *such that only one of* $sat_n(\vec{i}, \mathcal{S})$ *and* $sat_n(\vec{i}, \mathcal{S}')$ *holds.*

Before proceed with the proof of Theorem 4.4.3 we introduce a few auxiliary lemmas. First, Lemma 4.4.4 states that all elements of a non-empty sequence are needed, i.e. that every element in the sequence corresponds to a non-empty interval in the partition. This feature is required for the representation to be unique.

Lemma 4.4.4. *Let* $\mathcal{S} \in \mathbb{S}_n$ *for some* $n > 0$ *and* $\mathcal{S} \neq \epsilon$. *For each* $k \in \{1, \ldots, |\mathcal{S}|\}$ *there is* $i \in \mathbb{I}$ *such that* $find(i, \mathcal{S}) = k$.

Proof. Let $\mathcal{S} = \big((p_1, v_1), \ldots, (p_m, v_m)\big) \in \mathbb{S}_n$ and $p_k = (i_k, b_k)$. If $k = m$ then as our witness we choose $i_k + 1$ such that $p_k = (i_k, b_k) \prec (i_k + 1, \ominus)$, therefore by the definition of *find* we have that $find(i_k + 1, \mathcal{S}) = m$. Otherwise, $0 < k < m$ and we have two possibilities:

- If $b_k = \oplus$ (this closes the case $\mathbb{I} = \mathbb{Z}$) then our witness is i_k such that $p_k = (i_k, \oplus)$ and by the definition of $\mathbb{S}_n$ we know that $(i_k, \oplus) \prec p_{k+1}$, therefore by the definition of *find* we obtain $find(i_k, \mathcal{S}) = k$.

- If $b_k = \ominus$ then $\mathbb{I} = \mathbb{R}$ or $\mathbb{I} = \mathbb{Q}$. Let $p_{k+1} = (i_{k+1}, b_{k+1})$. By the definition of $\mathbb{S}_n$ we have that $p_k \prec p_{k+1}$, therefore by the definition of $\prec$ we obtain $i_k < i_{k+1}$. Taking into consideration the restriction on $\mathbb{I}$, there is $i \in \mathbb{I}$ such that $i_k < i < i_{k+1}$. Because $p_k = (i_k, \ominus) \prec (i, \oplus) \prec (i_{k+1}, b_{k+1}) = p_{k+1}$ we can use i as the witness: $find(i, \mathcal{S}) = k$. $\square$

Lemma 4.4.5 below states that every non-empty sequence $\mathcal{S} \in \mathbb{S}_n$ for some $n > 0$ is not-empty in terms of the predicate *sat*. The sequence $\mathcal{S}$ represents some subset of $\mathbb{I}^n$ that contains at least one element.

Lemma 4.4.5. *For each $n \geq 0$ and $\mathcal{S} \in \mathbb{S}_n$ it holds that:*

$$\mathcal{S} \neq \epsilon \iff \exists \vec{\boldsymbol{i}} \in \mathbb{I}^n \text{ such that } sat_n(\vec{\boldsymbol{i}}, \mathcal{S}). \tag{4.8}$$

Proof. We prove this by induction on n. For $n = 0$ we first assume the left hand side of (4.8): if $\mathcal{S} \neq \epsilon$ then by (4.3) we have that $\mathcal{S} = \top_0$. By the definition of *sat* we obtain that $sat_0(\vec{\boldsymbol{i}}, \mathcal{S})$ holds. Now assume the right hand side. By the definition of *sat* we have that $\mathcal{S} = \top_0 \neq \epsilon$.

Now assume that the current lemma holds for $n \geq 0$. We prove that it also holds for $n + 1$. First, assume the left hand side, that $\mathcal{S} \neq \epsilon$. Then we can consider the first element (p_1, v_1) of the sequence $\mathcal{S}$. By Lemma 4.4.4, there is $i \in \mathbb{I}$ such that $find(i, \mathcal{S}) = 1$ thus $\mathcal{S}[i] = v_1$. Additionally, by (4.3) we have that $v_1 \neq \epsilon$. By the induction hypothesis we obtain that there is $\vec{\boldsymbol{i}} = \langle i_n, \dots, i_1 \rangle$ such that $sat_n(\vec{\boldsymbol{i}}, v_1)$ holds. Then, by the definition of sat_{n+1} also $sat_{n+1}(\langle i, i_n, \dots, i_1 \rangle, \mathcal{S})$ holds. Now assume the right hand side, that there is $\vec{\boldsymbol{i}} \in \mathbb{I}^{n+1}$ such that $sat_{n+1}(\vec{\boldsymbol{i}}, \mathcal{S})$ holds. Let $\vec{\boldsymbol{i}} = \langle i_{n+1}, i_n, \dots, i_1 \rangle$. Then, by the definition of sat_{n+1} we have that $\mathcal{S}[i_{n+1}] = v$ for some $v \in \mathbb{S}_n$ such that $sat_n(\langle i_n, \dots, i_1 \rangle, v)$ holds. By the induction hypothesis we obtain that $v \neq \epsilon$. Then, by (4.3) the sequence $\mathcal{S}$ must have at least one element, therefore $\mathcal{S} \neq \epsilon$. $\square$

Lemma 4.4.6 is a little more complex. Assume that we have an arbitrary sequence $\mathcal{S} \in \mathbb{S}_n$ and any element of the sequence. The element is a pair with some $p_i \in \mathbb{I}$ as the first element. Lemma 4.4.6 states that if we choose any $p \in \mathbb{I}$ that is strictly greater that p_i in terms of the ordering $\prec$, then we can always find $i \in \mathbb{I}$, which belongs to the interval represented by p_i and p (in terms of the predicate *in*).

Lemma 4.4.6. *For all $n \geq 1$, $\mathcal{S} \in \mathbb{S}_n$, where $\mathcal{S} = \big((p_1, v_1), \dots, (p_m, v_m)\big)$, for $m \geq 1$, $k \in \{1, \dots, m\}$ and any $p \in \mathbb{P}$ such that $p_k \prec p$ there is $i \in \mathbb{I}$ such that $in(p_k, i, p)$ and $find(i, \mathcal{S}) = k$.*

Proof. Let $n, \mathcal{S}, k, p$ be as in the current lemma assumptions. First, consider a case when $k = m$. By Lemma 4.4.1, there is i such that $in(p_m, i, p)$. By the definition of *in* we have $p_m \preceq (i, \oplus)$, therefore $find(i, \mathcal{S}) = m$. Assume now that $k < m$. We have two possibilities:

- If $p_{k+1} \preceq p$, then because $p_k \prec p_{k+1}$, there is i such that $in(p_k, i, p_{k+1})$ (see Lemma 4.4.1). Then $find(i, \mathcal{S}) = k$ but also $p_k \prec (i, \oplus) \prec p_{k+1} \preceq p$, therefore $in(p_k, i, p)$.

- If $p \prec p_{k+1}$, then there exists i such that $in(p_k, i, p)$ (by Lemma 4.4.1), but also $p_k \preceq (i, \oplus) \prec p \prec p_{k+1}$, therefore $find(i, \mathcal{S}) = k$. $\square$

Finally, we prove Theorem 4.4.3. Just to remind: we want to show that for any $n \geq 0$, $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_n$, if $\mathcal{S} \neq \mathcal{S}'$ then we can distinguish these sequences by the predicate sat_n. We do this by finding $\vec{\boldsymbol{i}} \in \mathbb{I}^n$ such that exactly one of $sat_n(\vec{\boldsymbol{i}}, \mathcal{S})$ and $sat_n(\vec{\boldsymbol{i}}, \mathcal{S}')$ holds.

Proof. The case when one of the sequences $\mathcal{S}, \mathcal{S}'$ is equal to ϵ follows immediately by Lemma 4.4.5. From now on we assume that both sequences are not empty. The rest of the proof is by induction on n. For $n = 0$ if $\mathcal{S} \neq \mathcal{S}'$ then either $\mathcal{S} = \epsilon$ or $\mathcal{S}' = \epsilon$, which is already handled.

Now assume that the current lemma holds for some $n \geq 0$, where $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_n$. We prove that it also holds for $n + 1$. Let $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_{n+1}$ be such that $\mathcal{S} \neq \mathcal{S}'$, $\mathcal{S} = \big((p_1, v_1), \dots, (p_{|\mathcal{S}|}, v_{|\mathcal{S}|})\big)$, and $\mathcal{S}' = \Big((p'_1, v'_1), \dots, (p'_{|\mathcal{S}'|}, v'_{|\mathcal{S}'|})\Big)$.

First, consider the case when the sequences differ at some index. Formally, we assume that there is an index $j \in \{1, \dots, \min(|\mathcal{S}|, |\mathcal{S}'|)\}$ such that $(p_j, v_j) \neq (p'_j, v'_j)$. Let k be the smallest such index, that is for all $1 \leq i < k$ it holds that $(p_i, v_i) = (p'_i, v'_i)$ and $(p_k, v_k) \neq (p'_k, v'_k)$. We have the following cases:

1. Elements at index k differ at first element of the pair, that is $p_k \neq p'_k$. Then, without loss of generality we may assume that $p_k \prec p'_k$. By Lemma 4.4.6 there is $i \in \mathbb{I}$ such that $find(i, \mathcal{S}) = k$ and $in(p_k, i, p'_k)$. There are two possibilities:

   - If the sequences differ at the first element, i.e. $k = 1$, then by (4.3) we have that $v_1 \neq \epsilon$. By Lemma 4.4.5 there is $\vec{\boldsymbol{i}} = \langle i_n, \dots i_1 \rangle$ such that $sat_n(\vec{\boldsymbol{i}}, v_1)$ holds. Because $find(i, \mathcal{S}) = 1$, by the definition of sat_{n+1} we obtain that $sat_{n+1}(\langle i, i_n, \dots, i_1 \rangle, \mathcal{S})$ also holds. Additionally, since $in(p_1, i, p'_1)$ we have that $(i, \oplus) \prec p'_1$. By (4.4) we obtain $find(i, \mathcal{S}') = 0$, which by (4.5) we can write using the *access notation* as $\mathcal{S}'[i] = \epsilon$. Therefore, $sat_{n+1}(\langle i, i_n, \dots, i_1 \rangle, \mathcal{S}')$ does not hold.

   - If the sequences $\mathcal{S}$ and $\mathcal{S}'$ differ at some index $k > 1$ then $v_{k-1} = v'_{k-1}$ and $p_{k-1} = p'_{k-1}$. By the definition of $\mathbb{S}_{n+1}$ we have that $v_k \neq v_{k-1}$. We use the induction hypothesis for v_k and v'_{k-1}. We obtain a vector $\langle i_n, \dots, i_1 \rangle$ such that:

   $$sat_n(\langle i_n, \dots, i_1 \rangle, v_k) \iff \neg sat_n(\langle i_n, \dots, i_1 \rangle, v'_{k-1}). \tag{4.9}$$

   Since $p_k \preceq (i, \oplus) \prec p'_k$ (by $in(p_k, i, p'_k)$) and $p'_{k-1} \prec p_k$ (by (4.3)) then also $p'_{k-1} \preceq (i, \oplus) \prec p'_k$, therefore $in(p'_{k-1}, i, p'_k)$. By the definition of *find* we obtain that $find(i, \mathcal{S}') = k - 1$, and using the *access notation* we have $\mathcal{S}'[i] = v'_{k-1} = v_{k-1}$. By the definition of sat_{n+1} we obtain:

   $$sat_{n+1}(\langle i, i_n, \dots, i_1 \rangle, \mathcal{S}') \iff sat_n(\langle i_n, \dots, i_1 \rangle, v'_{k-1}). \tag{4.10}$$

Also we have that $\mathit{find}(i, \mathcal{S}) = k$, thus $\mathcal{S}[i] = v_k$ and:

$$sat_{n+1}(\langle i, i_n, \dots, i_1\rangle, \mathcal{S}) \iff sat_n(\langle i_n, \dots, i_1\rangle, v_k). \tag{4.11}$$

Therefore, by (4.9) combined with (4.10) and (4.11), exactly one of $sat_{n+1}(\langle i, i_n, \dots, i_1\rangle, \mathcal{S})$ and $sat_{n+1}(\langle i, i_n, \dots, i_1\rangle, \mathcal{S}')$ holds.

2. Otherwise, elements of sequences $\mathcal{S}$ and $\mathcal{S}'$ at index k differ on the second element of pair, i.e. $p_k = p'_k$, $v_k \neq v'_k$ and for all $1 < j < k$ it holds that $(p_j, v_j) = (p'_j, v'_j)$. We use the induction hypothesis for v_k and v'_k to obtain $\vec{\boldsymbol{i}} = \langle i_n, \dots, i_1\rangle$ such that exactly one of $sat_n(\vec{\boldsymbol{i}}, v_k)$ and $sat_n(\vec{\boldsymbol{i}}, v'_k)$ holds. All we need to do is to find $i \in \mathbb{I}$ such that $\mathit{find}(i, \mathcal{S}) = \mathit{find}(i, \mathcal{S}') = k$. In that case exactly one of $sat_{n+1}(\langle i, i_n, \dots, i_1\rangle, \mathcal{S})$ and $sat_{n+1}(\langle i, i_n, \dots, i_1\rangle, \mathcal{S}')$ holds. We have the following possibilities:
   - If $k = |\mathcal{S}| = |\mathcal{S}'|$ then by Lemma 4.4.4 there exists $i \in \mathbb{I}$ such that $\mathit{find}(i, \mathcal{S}) = k$. Then $x'_k = x_k \preceq i$, therefore $\mathit{find}(i, \mathcal{S}') = k$.
   - If $k = |\mathcal{S}| < |\mathcal{S}'|$ then by Lemma 4.4.6 there exists $i \in \mathbb{I}$ such that $in(p_k, i, p'_{k+1})$ and $\mathit{find}(i, \mathcal{S}) = k$. Then $p'_k = p_k \preceq i \prec p_{k+1}$, therefore $\mathit{find}(i, \mathcal{S}') = k$.
   - If $k = |\mathcal{S}'| < |\mathcal{S}|$ then we proceed analogically to the case above.
   - Otherwise, it holds that $k < \min(|\mathcal{S}|, |\mathcal{S}'|)$. We use Lemma 4.4.6 for $p = \min_{\prec}(p_{k+1}, p'_{k+1})$ and obtain $i \in \mathbb{I}$ such that $in(p_k, i, p)$ and $\mathit{find}(i, \mathcal{S}) = k$. Recall that $p_k = p'_k$, by the definition of in we obtain that $in(p'_k, i, p'_{k+1})$ holds. Therefore, $\mathit{find}(i, \mathcal{S}') = k$.

We have covered cases, where the sequences $\mathcal{S}$ and $\mathcal{S}'$ differ at some index. If there is no such index, i.e. for any $k \in \{1, \dots, \min(|\mathcal{S}|, |\mathcal{S}'|)\}$ it holds that $(p_k, v_k) = (p'_k, v'_k)$, then one sequence is a prefix of the other. Without loss of generality we assume $\mathcal{S}$ is the prefix of $\mathcal{S}'$, thus $|\mathcal{S}| < |\mathcal{S}'|$. Let m be the length of the shorter sequence $\mathcal{S}$, i.e. $m = |\mathcal{S}|$. By Lemma 4.4.4 for the sequence $\mathcal{S}'$ and $k = m + 1$ we obtain that there exists $i \in \mathbb{I}$ such that $\mathit{find}(i, \mathcal{S}') = m + 1$. By (4.5) we can write this using the *access notation*, i.e. $\mathcal{S}'[i] = v'_{m+1}$. We have that $p_m = p'_m \preceq p_{m+1}$, therefore $\mathit{find}(i, \mathcal{S}) = m$ and by (4.5) we obtain $\mathcal{S}[i] = v_m$. Since $\mathcal{S}$ is the prefix of $\mathcal{S}'$, it holds that $v_m = v'_m$, thus by (4.3) we obtain $v_m \neq v'_{m+1}$. By the induction hypothesis there is vector $\vec{\boldsymbol{i}} = \langle i_n, \dots, i_1\rangle$ such that exactly one of $sat_n(\vec{\boldsymbol{i}}, v_m)$ and $sat_n(\vec{\boldsymbol{i}}, v'_{m+1})$ holds. By the definition of sat_{n+1} and the fact that $\mathcal{S}[i] = v_m$ we have:

$$sat_{n+1}(\langle i, i_n, \dots, i_1\rangle, \mathcal{S}) \iff sat_n(\langle i_n, \dots i_1\rangle, v_m).$$

Analogically, since $\mathcal{S}'[i] = v'_{m+1}$ it holds that:

$$sat_{n+1}(\langle i, i_n, \dots, i_1\rangle, \mathcal{S}') \iff sat_n(\langle i_n, \dots i_1\rangle, v'_{m+1}).$$

Therefore, by the definition of sat_{n+1} exactly one of $sat_{n+1}(\langle i, i_n, \dots, i_1\rangle, S)$ and $sat_{n+1}(\langle i, i_n, \dots, i_1\rangle, S')$ holds. □

We have proved that elements of the sequence $\mathbb{S}_n$ represent unique subsets of $\mathbb{I}^n$, but we still have not proved that they can be used to represent elements of *boxes*. This is done in the next section.

Representation of *boxes* based on Linear Decision Diagrams

An LDD [22] is a binary decision diagram with two terminal nodes: *true* and *false*, in which non-terminal nodes (decisions) are linear constraints. It happens that an LDD is also a DAG. Additionally, there is a total order $\preceq_{LDD}$ on all nodes, i.e. order on linear constraints extended to cover also constants *true*, *false*. The order $\preceq_{LDD}$ corresponds to an order, in which decisions in the LDD are made, i.e. if $v \preceq_{LDD} u$ then the node v is in the LDD graph before the node u. From every linear constraint node in the LDD, there is a path to both *true* and *false* nodes. An example of an LDD is presented in Fig. 4.3. Note that in the example first we make decisions about values of the variable x, and then about values of the variable y, i.e. there are no edges from nodes with the variable y to nodes with the variable x. This is a consequence of the ordering $\preceq_{LDD}$.

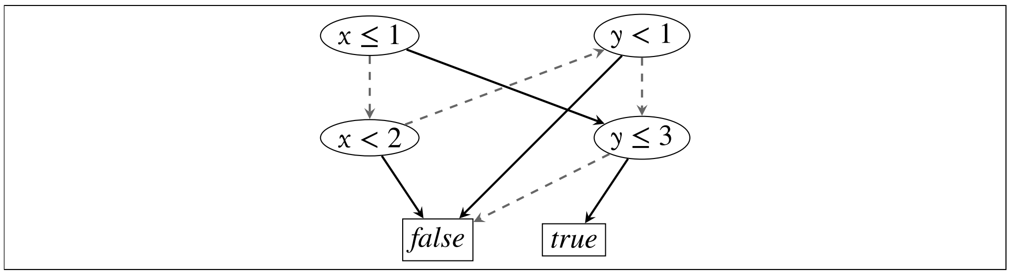


Figure 4.3: An example of a Linear Decision Diagram — ellipse nodes represent non-terminal (decision) nodes, square *true* an *false* nodes are terminal nodes, positive decisions (node test results) are denoted as solid arrows, while negative ones as dashed grey ones.

The representation of *boxes* proposed by Gurfinkel and Chaki uses LDDs with relational interval constraints (both strict and non-strict). Let $\preceq \subseteq \mathit{Var} \times \mathit{Var}$ be a total order on variables. The order $\preceq_{LDD}$ on interval constraints that is used in the implementation of *boxes*, for any $x_1, x_2 \in \mathit{Var}$ and $k_1, k_2 \in \mathbb{I}$ is defined as follows:

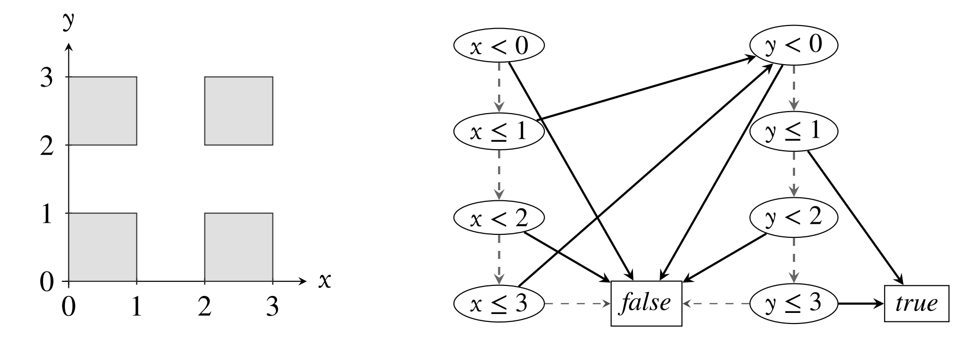
$$(x_1 \lesssim_1 k_1) \preceq_{LDD} (x_2 \lesssim_2 k_2) \iff (x_1 \preceq x_2) \vee ((x_1 \lesssim_1 k_1) \Rightarrow (x_2 \lesssim_2 k_2)),$$

where $\lesssim_1, \lesssim_2$ are strict or non-strict inequalities, i.e. $\lesssim_1, \lesssim_2 \in \{<, \leq\}$ and $\Rightarrow$ is constraint inference. For example, $x < 0 \Rightarrow x < 2$, but $x \leq 0 \nRightarrow x < 0$ because

$x = 0$ fulfils the left hand side and does not fulfil the right hand side. The ordering is additionally extended to *true* and *false* nodes as follows:

$$u \preceq_{LDD} v \iff (v \in \{true, false\}) \vee (u \notin \{true, false\} \wedge u \preceq_{LDD} v).$$

LDDs that satisfy certain ordering and reduction constraints (see [22, Sec. IID]) are canonical representations of propositional formulæ. The example LDD in Fig. 4.3 is an LDD for *boxes*, where the variable order is $x \preceq y$: all the nodes with variable x appear before nodes with variable y.



(a) Geometric representation (b) LDD based representation

$$\xrightarrow{\hspace{6cm}} y$$

$(0, \oplus), (((0, \oplus), \top_0), ((1, \ominus), \epsilon), ((2, \oplus), \top_0), ((3, \ominus), \epsilon))$

$(1, \ominus), \epsilon$

$(2, \oplus), (((0, \oplus), \top_0), ((1, \ominus), \epsilon), ((2, \oplus), \top_0), ((3, \ominus), \epsilon))$

$(3, \ominus), \epsilon$

$\downarrow x$

(c) Sweeping line representation

Figure 4.4: Comparison of the proposed representation (c) and the representation based on LDDs (b) for the example element in (a).

The proposed representation that is based on the sweeping line technique can be considered as a generalisation of the representation using LDDs. The ordering proposed by Gurfinkel and Chaki sorts nodes in an LDD by variables first, and then by the entailment of constraints for a variable (see Fig. 4.4(b)). The sweeping line representation sorts in the same way (see Fig. 4.4(c)) — every special point corresponds to a constraint (node) in the LDD. The difference is that the LDD DAG is an optimised version of the sweeping line technique tree representation, where duplicate subtrees are merged together. That is why an LDD is a DAG instead of a tree. In the example in Fig. 4.4(c), values (sequences) for $x = (0, \oplus)$ and $x = (2, \oplus)$ are exactly the same, thus in the corresponding LDD solid arrows from $x \leq 1$ and $x \leq 3$ lead to the same node. The LDD-based representation is one of the optimisations that can be applied to the sweeping line representation. One could use some graph algorithms or compression techniques used in graphical applications. For example, two-dimensional case can be represented as a compressed image using any lossless image compression algorithm.

4.5 Domain Operations

In this section, we describe an implementation of exact $\cup$, $\cap$ and $\subseteq$ operations on elements of sequences $\mathbb{S}_0, \mathbb{S}_1, \ldots$ We use these algorithms to prove the correspondence between elements in our representation and the domain of *boxes*, i.e. that the representation using the sweeping line technique can be used to describe elements of the domain.

The main idea behind the definition of domain operators is to define operators on $\mathbb{S}_0$ and provide a generic, inductive construction that can be used to extend them to $\mathbb{S}_n$ for any $n \geq 0$. Such extension is defined as follows:

Definition 4.5.1 (♦-extension). Let $\blacklozenge : \mathbb{S}_0 \times \mathbb{S}_0 \to \mathbb{S}_0$ be some base operator defined for $\mathbb{S}_0$. We define ♦-extension $\blacklozenge : \mathbb{S}_n \times \mathbb{S}_n \to \mathbb{S}_n$ for any $n \geq 0$ such that for $n = 0$ it is equal to the base operator and for $n > 0$, any $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_n$ it holds that $\mathcal{S} \blacklozenge \mathcal{S}' = \mathcal{R}$, where:

$$\forall i \in \mathbb{I} : \mathcal{R}[i] = \mathcal{S}[i] \blacklozenge \mathcal{S}'[i]. \tag{4.12}$$

The definition of ♦-extension function is correct, which is stated by Theorem 4.5.2.

Theorem 4.5.2. *For any function* $\blacklozenge : \mathbb{S}_0 \times \mathbb{S}_0 \to \mathbb{S}_0$*, there is only one function that is a ♦-extension, that is for any* $n > 0$ *and* $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_n$ *there is only one* $\mathcal{R} \in \mathbb{S}_n$ *such that* (4.12) *holds.*

In the proof of Theorem 4.5.2, which is presented after few auxiliary lemmas, we construct a function that is a ♦-extension, and then show that it is the only one possible. For $n > 0$ and two sequences $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_n$, the result sequence $\mathcal{R} = \mathcal{S} \blacklozenge \mathcal{S}'$ is built from pairs such that:

- first elements of pairs in $\mathcal{R}$ come from one of the two sequences $\mathcal{S}$ or $\mathcal{S}'$,
- second elements of pairs in $\mathcal{R}$ are computed by an inductive application of the ♦-extension for second elements of the input sequences $\mathcal{S}$ and $\mathcal{S}'$.

The implementation of ◆-extension first creates an *unnormalized* version of the result sequence, which is similar to (4.3) but without the constraints about second elements of pairs (i.e. values). Then the sequence is normalized. We define a sequence normalisation procedure:

Definition 4.5.3 (Sequence Normalisation). Let Q' be a finite, non-empty sequence of pairs:

$$Q' = \Big((p'_1, v'_1), (p'_2, v'_2), \dots, (p'_{|Q'|}, v'_{|Q'|})\Big), \tag{4.13}$$

where for any $j \in \{1, \dots, |Q'|\}$ it holds that $p'_j \in \mathbb{P}_\infty$, $n > 0$, $v_j \in \mathbb{S}_n$ and for any $j \in \{1, \dots, |Q'| - 1\}$ it holds that $p'_j \prec p'_{j+1}$. We define a *normalised version* of the sequence Q' as a sequence Q:

$$Q = \big((p_1, v_1), \dots, (p_{|Q|}, v_{|Q|})\big) \tag{4.14}$$

such that:

1. The sequence Q is a subsequence of Q'. It means that it is built from elements of the sequence Q', i.e. for every $j \in \{1, \dots, |Q|\}$ there is $k \in \{1, \dots, |Q'|\}$ such that $(p_j, v_j) = (p'_k, v'_k)$. Also elements of the sequence Q are in the same order as in the sequence Q', i.e.
$$\forall j \in \{1, \dots, |Q| - 1\} : p_j \prec p_{j+1}. \tag{4.15}$$
2. The first element in Q' — the pair (p'_1, v'_1), appears in Q only if it has a non-empty value, that is:
$$v'_1 = \epsilon \iff p'_1 \notin \{p_1, \dots, p_{|Q|}\}. \tag{4.16}$$
3. A pair from Q' appears in Q only if it has a different value than the pair preceding it in Q':
$$\forall j \in \{1, \dots, |Q'| - 1\} : v'_j \neq v'_{j+1} \iff p'_{j+1} \in \{p_1, \dots, p_{|Q|}\}. \tag{4.17}$$

Now we prove a few lemmas that we use in the proof of Theorem 4.5.2. Consider a block of consecutive pairs removed during the sequence normalisation procedure. All pairs in such block have equal values, which is stated by Lemma 4.5.4.

Lemma 4.5.4. *Let Q' and Q be as in Definition 4.5.3. For every $k \geq 1$ and $l \in \{k, k+1, \dots, |Q'|\}$. Then it holds that:*

$$\big(\forall k \leq j \leq l : p'_j \notin \{p_1, \dots, p_{|Q|}\}\big) \Rightarrow v'_k = v'_{k+1} = \dots = v'_l. \tag{4.18}$$

Proof. Let Q, Q', k, l be as in the current lemma assumptions. Assume further the left hand side of eq. (4.18). If $l = k$ then the current lemma trivially holds since there is only one value on the right hand side. Now assume that $l > k$. Let m be any index such that $k \leq m \leq l - 1$. Since $p'_{m+1} \notin \{p_1, \dots, p_{|Q|}\}$, by eq. (4.17) it holds that $v'_m = v'_{m+1}$. Since it holds for every $m \in \{k, \dots, l-1\}$, using simple induction we obtain $v'_k = v'_{k+1} = \dots = v'_l$. □

Now we proceed to the next lemma that states what is the purpose of the normalisation procedure introduced in Definition 4.5.3 — the normalised version belongs to $\bigcup_{n>0} \mathbb{S}_n$.

Lemma 4.5.5. *Let Q' and Q be as introduced in Definition 4.5.3 and for some $n \geq 0$. Then it holds that $Q \in \mathbb{S}_{n+1}$.*

Proof. Let Q, Q' be as in the current lemma assumptions. We prove that Q fulfills all conditions in the definition of $\mathbb{S}_{n+1}$, see eq. (4.3). If $Q = \epsilon$ then trivially $Q \in \mathbb{S}_{n+1}$. For the rest of the proof we assume that $Q \neq \epsilon$. By eq. (4.15) we directly obtain one of the conditions from eq. (4.3) — that first elements of Q are sorted. Therefore, in the rest of the proof we focus only on values, i.e. second elements of pairs.

Note that by Definition 4.5.3 second elements of pairs in the sequence Q' are from $\mathbb{S}_n$, i.e. for any $j \in \{1, \dots, |Q'|\}$ it holds that $v'_j \in \mathbb{S}_n$. Since Q is a subsequence of Q', for every $j \in \{1, \dots, |Q|\}$ it holds that $v_j \in \mathbb{S}_n$.

We prove that $v_1 \neq \epsilon$. Assume the contrary, that $v_1 = \epsilon$. Let us present Q, Q' as follows:

$$\begin{array}{rl} Q' = & (\,(p'_1, v'_1),\ (p'_2, v'_2),\ \dots,\ (p'_l, v_l),\ (p'_{l+1}, v'_{l+1}),\ \dots\,), \\ Q = & (\qquad\qquad\qquad\qquad\qquad\qquad\quad (p_1, v_1),\qquad \dots\,). \end{array}$$

By eq. (4.16) the first pair in Q — (p_1, v_1) is not the first one in the sequence Q', thus $p_1 \neq p'_1$. Since Q is a subsequence of Q' it holds that $p'_1 \prec p_1$. Additionally, since $p'_1 \notin \{p_1, \dots, p_{|Q|}\}$ by eq. (4.16) it holds that $v'_1 = \epsilon$. By Lemma 4.5.4 for $k = 1$ and $l = \max(j \mid p'_j \prec p_1)$ it holds that $v'_1 = \dots = v'_l$, thus $v'_l = \epsilon$. Additionally, $p'_{l+1} = p_1 = \epsilon$ and by eq. (4.17) we obtain $p'_{l+1} \notin \{p_1, \dots, p_{|Q|}\}$, which is a contradiction.

Now we prove that for every $j \in \{1, \dots, |Q| - 1\}$ it holds that $v_j \neq v_{j+1}$. Assume the contrary, that there exists index j such that $v_j = v_{j+1}$. Let us present Q, Q' as follows:

$$\begin{array}{rl} Q' = & (\,\dots,\ (p'_i, v'_i),\ (p'_{i+1}, v'_{i+1}),\ \dots,\ (p'_l, v_l),\ (p'_{l+1}, v'_{l+1}),\ \dots\,), \\ Q = & (\,\dots,\ (p_j, v_j),\qquad\qquad\qquad\qquad\qquad\quad (p_{j+1}, v_{j+1}),\ \dots\,). \end{array}$$

Since $\mathcal{Q}$ is a subsequence of $\mathcal{Q}'$, there exists index $i \in \{1, \dots, |\mathcal{Q}'| - 1\}$ such that $(p_j, v_j) = (p'_i, v'_i)$. Note that $p'_{i+1} \prec p_{j+1}$. Otherwise, if $p_{j+1} \prec p'_{i+1}$ then we obtain a contradiction with the fact that $\mathcal{Q}$ is a subsequence of $\mathcal{Q}'$ and if $p_{j+1} = p'_{i+1}$, by eq. (4.17) it would hold that $p_{j+1} \notin \{p_1, \dots, p_{|\mathcal{Q}|}\}$. By Lemma 4.5.4 for $k = i + 1$ and $l = \max(m \mid p'_m \prec p_{j+1})$ it holds that $v'_{i+1} = \dots = v'_l$. Additionally, $p'_{l+1} = p_{j+1}$. By eq. (4.17) for index i and $i + 1$ we have that $v_j = v'_i = v'_{i+1}$. Similarly, by eq. (4.17) for index l and $l + 1$ we have that $v'_l \neq v'_{l+1} = v_{j+1}$. Finally, we obtain $v_j = v'_i = v'_{i+1} = v'_l \neq v'_{l+1} = v_{j+1}$, which is a contradiction.

We have proved that $\mathcal{Q}$ fulfils all the conditions in eq. (4.3), therefore $\mathcal{Q} \in \mathbb{S}_{n+1}$.□

In the proof of Theorem 4.5.2 we use a concept of a *special point*. We define a function $spec_L : \bigcup_{n>0} \mathbb{S}_n \to \mathcal{P}(\mathbb{P}_\infty)$, which outputs a set of *local special points* for a given sequence as follows:

$$spec_L(\mathcal{S}) = \begin{cases} \emptyset & \text{if } \mathcal{S} = \epsilon \\ \{p_1, \dots, p_k\} & \text{otherwise, for } \mathcal{S} = \big((p_1, v_1), \dots, (p_k, v_k)\big) . \end{cases} \tag{4.19}$$

The set of *local special points* of sequence $\mathcal{S}$ is the set of first elements of pairs in $\mathcal{S}$, which are stop points encountered during the process of sweeping (see Section 4.3). Now we proceed with the proof of Theorem 4.5.2.

Proof. Let $\blacklozenge$ be a base operator on $\mathbb{S}_0$ as in Definition 4.5.1. The proof is by induction on n. For $n = 0$ the correctness results directly from the definition as the $\blacklozenge$-extension is the same as the base $\blacklozenge$ operator. Now assume that the definition is correct for some $n \geq 0$, we prove that it is also correct for $n + 1$. More specifically, we prove that for any $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_{n+1}$ there is only one $\mathcal{R} \in \mathbb{S}_{n+1}$ such that eq. (4.12) holds. We construct $\mathcal{R} \in \mathbb{S}_{n+1}$ that fulfills the definition.

Let $\mathbb{X}$ be the set of special points from both sequences $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_{n+1}$ with additional point $-\infty$, that is:

$$\mathbb{X} = \{-\infty\} \cup spec_L(\mathcal{S}) \cup spec_L(\mathcal{S}')$$

and $p'_1, \dots, p'_{|\mathbb{X}|}$ be consecutive elements of $\mathbb{X}$, that is $\mathbb{X} = \{p'_1, \dots, p'_{|\mathbb{X}|}\}$, where for any $j \in \{1, \dots |\mathbb{X}| - 1\}$ it holds that $p'_j \prec p'_{j+1}$. By Lemma 4.4.1, for every j such that $j \in \{1 \dots |\mathbb{X}| - 1\}$ there is i_j such that $in(p'_j, i_j, p'_{j+1})$. Additionally, there is $i_{|\mathbb{X}|}$ such that $p_{|\mathbb{X}|} \preceq (i_{|\mathbb{X}|}, \oplus)$. These elements $i_1, \dots, i_{|\mathbb{X}|}$ are representatives of consequent segments in the segmentation of $\mathbb{I}$ by special points from $\mathbb{X}$. We have:

$$-\infty = p'_1 \preceq (i_1, \oplus) \prec p'_2 \preceq \dots \prec p_{|\mathbb{X}|} \preceq (i_{|\mathbb{X}|}, \oplus).$$

For every $j \in \{0, \dots, |\mathbb{X}| - 1\}$ and $i \in \mathbb{I}$ such that $in(p'_j, i, p'_{j+1})$ it holds that $\mathcal{S}[i] = \mathcal{S}[i_j]$ and $\mathcal{S}'[i] = \mathcal{S}'[i_j]$. Additionally, for every $i \in \mathbb{I}$ such that $p_{|\mathbb{X}|} \preceq (i, \oplus)$ it holds that $\mathcal{S}[i] = \mathcal{S}[i_{|\mathbb{X}|}]$ and $\mathcal{S}'[i] = \mathcal{S}'[i_{|\mathbb{X}|}]$.

We construct a sequence of pairs $\mathcal{R}'$ as follows:

$$\mathcal{R}' = \Big((p'_1, v'_1), (p'_2, v'_2), \dots, (p'_{|\mathbb{X}|}, v'_{|\mathbb{X}|})\Big), \tag{4.20}$$

where $v'_j = \mathcal{S}[i_j] \blacklozenge \mathcal{S}'[i_j]$ for any $j \in \{1, \dots, |\mathbb{X}|\}$. Note that by the induction hypothesis v'_j is well defined and belongs to $\mathbb{S}_n$. Also, the sequence $\mathcal{R}'$ does not depend on the choice of the representatives $i_1, \dots, i_{|\mathbb{X}|}$, i.e. it is the same for every such choice. The construction of $\mathcal{R}'$ is a kind of pointwise application of $\blacklozenge$ to segments that appear in the segmentation of $\mathbb{I}$ by the set of special points — $\mathbb{X}$. Let $\mathcal{R}$ be a normalised version of the sequence. By Lemma 4.5.5 it holds that $\mathcal{R} \in \mathbb{S}_{n+1}$.

Now we prove that for any $i \in \mathbb{I}$ it holds that $\mathcal{R}[i] = \mathcal{S}[i] \blacklozenge \mathcal{S}'[i]$. Let $i \in \mathbb{I}$. Since $p_1 = -\infty$, we may compute $j = \max(m \mid p'_m \preceq (i, \oplus))$. By the construction of $\mathbb{X}$ it holds that $\mathcal{S}[i] = \mathcal{S}[i_j]$ and $\mathcal{S}'[i] = \mathcal{S}'[i_j]$, therefore $\mathcal{S}[i] \blacklozenge \mathcal{S}'[i] = \mathcal{S}[i_j] \blacklozenge \mathcal{S}'[i_j] = v'_j$. We prove that $\mathcal{R}[i] = v'_j$. Let $\mathit{find}(\mathcal{R}, i) = k$. Consider the following cases:

- If $k = 0$ and $\mathcal{R} = \epsilon$ then all pairs were removed from the non-empty sequence $\mathcal{R}'$ during the normalisation procedure. By Lemma 4.5.4 for $k = 1$ and $l = |\mathcal{R}'|$ it holds that all removed pairs had equal values, thus $v'_j = v'_1$ and by eq. (4.16) we have $v'_j = v'_1 = \epsilon$, and finally by eq. (4.5) we obtain $\mathcal{R}[i] = \epsilon = v'_j$.
- If $k = 0$ and $\mathcal{R} \neq \epsilon$ then by eq. (4.5) it holds that $\mathcal{R}[i] = \epsilon$. By eq. (4.4) we have $(i, \oplus) \prec p_1$. Since $\mathcal{R}$ is a subsequence of $\mathcal{R}'$, during the normalisation procedure a whole block of elements preceding (p_1, v_1) was removed. By Lemma 4.5.4 for $k = 1$ and $l = \max(m \mid p'_m \prec p_1)$ all these pairs had equal values and by eq. (4.16) these values were equal to ϵ. Since $p'_j \preceq (i, \oplus) \prec p_1$, it holds that $1 \leq j \leq l$, therefore $v'_j = \epsilon$.
- If $k > 0$ then by the definition of *find* it holds that $p_k \preceq (i, \oplus)$. Additionally, since $\mathcal{R}$ is a subsequence of $\mathcal{R}'$, it holds that $p_k \preceq p'_j$. If $p_k = p'_j$ then the element at index j was not removed from $\mathcal{R}'$ and $\mathcal{R}[i] = v_k = v'_j$. Otherwise, if $p_k \prec p'_j$, the element at index j was removed from $\mathcal{R}'$. Let us present $\mathcal{R}, \mathcal{R}'$ and i as follows:

$$\begin{array}{rlll} \mathcal{R}' = & (\dots (p'_{k'}, v'_{k'}), & \dots, (p'_j, v'_j), & \dots\ ), \\ \mathcal{R} = & (\dots (p_k, v_k), & & \dots\ ). \\ & & & i \end{array}$$

  Since $\mathcal{R}$ is a subsequence of $\mathcal{R}'$, there is index $k' \in \{1, \dots, |\mathbb{X}|\}$ such that $(p'_{k'}, v'_{k'}) = (p_k, v_k)$. Because $p_k \preceq p'_j$, then all items with indexes in $\{k'+1, k'+2, \dots, j'\}$ were removed during the normalisation procedure and by Lemma 4.5.4 it holds that $v'_{k'+1} = \dots = v'_j$. By eq. (4.17) we obtain $v_{k'} = v_{k'+1} = v'_j$, therefore $\mathcal{R}[i] = v_k = v'_j$.

The uniqueness of $\mathcal{R}$ is a direct consequence of the definition of the normalisation procedure in *Definition* 4.5.3 and Theorem 4.4.3. □

Let $\|\cdot\| : \bigcup_{n\geq 0} \mathbb{S}_n \to \mathbb{N}$ be the function that returns the size of a domain element (cumulative length of all sequences), i.e. for any $\mathcal{S} \in \mathbb{S}_n$ such that $n \geq 0$ it is defined as follows:

$$\|\mathcal{S}\| = \begin{cases} 1 & \text{if } \mathcal{S} = \epsilon \text{ or } \mathcal{S} = \top_0 \\ k + \sum_{i=1}^{k} \|s_i\| & \text{otherwise, for } \mathcal{S} = \big((p_1, s_1), \dots, (p_k, s_k)\big) \text{ and } k > 0. \end{cases}$$

Then the $\blacklozenge$-extension can be implemented with time complexity $O(\|\mathcal{S}\| * \|\mathcal{S}'\|)$. This is also a limit of the size of the result.

Lemma 4.5.6 (Domain Operations). *Exact domain operations can be defined as $\blacklozenge$-extensions of the operations for $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_0$:*

- ***join*** *— by an extension of $\mathcal{S} \blacklozenge \mathcal{S}' = \top_0 \iff \mathcal{S} = \top_0$ or $\mathcal{S}' = \top_0$;*
- ***meet*** *— by an extension of $\mathcal{S} \blacklozenge \mathcal{S}' = \top_0 \iff \mathcal{S} = \mathcal{S}' = \top_0$;*
- ***inclusion*** *— for $\subseteq$ we first compute the extension of an auxiliary operator $\mathcal{S} \blacklozenge \mathcal{S}' = \top_0 \iff \mathcal{S} = \top_0$ and $\mathcal{S}' = \epsilon$; for $\mathcal{S}_n, \mathcal{S}'_n \in \mathbb{S}_n$ it holds that $\mathcal{S}_n \blacklozenge \mathcal{S}'_n = \epsilon \iff \gamma(\mathcal{S}_n) \subseteq \gamma(\mathcal{S}'_n)$, where γ is the concretisation function defined in eq.* (4.7).

Proof. We follow the cases from the formulation of the lemma:

join We prove that for $n \geq 0$ and $\mathcal{S}, \mathcal{S}', \mathcal{R} \in \mathbb{S}_n$ if $\mathcal{R} = \mathcal{S} \blacklozenge \mathcal{S}'$ then for any $\vec{\boldsymbol{i}} \in \mathbb{I}^n$ it holds that:

$$sat_n(\vec{\boldsymbol{i}}, \mathcal{R}) \iff sat_n(\vec{\boldsymbol{i}}, \mathcal{S}) \text{ or } sat_n(\vec{\boldsymbol{i}}, \mathcal{S}'). \tag{4.21}$$

We prove this by induction on n. For $n = 0$ the operator is defined as $\mathcal{S} \blacklozenge \mathcal{S}' = \top_0 \iff \mathcal{S} = \top_0$ or $\mathcal{S}' = \top_0$. Thus, (4.21) holds directly by the definition of sat_0. Now assume that (4.21) holds for $n \geq 0$. We prove that it also holds for $n + 1$. Let $\vec{\boldsymbol{i}} = \langle i_{n+1}, i_n, \dots, i_1 \rangle$.

First, assume the right hand side. Without loss of generality we may assume that $sat_{n+1}(\vec{\boldsymbol{i}}, \mathcal{S})$ holds. By the definition of sat_{n+1} we obtain $\mathcal{S}[i_{n+1}] = v$ for some $v \in \mathbb{S}_n$ such that $sat_n(\langle i_n, \dots, i_1 \rangle, v)$ holds. By (4.12) we have that $\mathcal{R}[i_{n+1}] = \mathcal{S}[i_{n+1}] \blacklozenge \mathcal{S}'[i_{n+1}] = v \blacklozenge \mathcal{S}'[i_{n+1}]$. Since $sat_n(\langle i_n, \dots, i_1 \rangle, v)$ holds, by the induction hypothesis we obtain that $sat_n(\langle i_n, \dots, i_1 \rangle, \mathcal{R}[i_{n+1}])$ holds. Finally, by the definition of sat_{n+1} we have the left hand side: $sat_{n+1}(\vec{\boldsymbol{i}}, \mathcal{R})$.

Now assume the left hand side of eq. (4.21), i.e. that $sat_{n+1}(\vec{\boldsymbol{i}}, \mathcal{R})$ holds. By the definition of sat_{n+1} we have that $\mathcal{R}[i_{n+1}] = v$ for some $v \in \mathbb{S}_n$ such that $sat_n(\langle i_n, \dots, i_1 \rangle, v)$ holds. By the construction of the $\blacklozenge$-extension we have that $v = \mathcal{R}[i_{n+1}] = \mathcal{S}[i_{n+1}] \blacklozenge \mathcal{S}'[i_{n+1}]$. Since $sat_n(\langle i_n, \dots, i_1 \rangle, v)$ holds, by the induction hypothesis we obtain that $sat_n(\langle i_n, \dots, i_1 \rangle, \mathcal{S}[i_{n+1}])$ or $sat_n(\langle i_n, \dots, i_1 \rangle, \mathcal{S}'[i_{n+1}])$ holds. Finally, by the definition of sat_{n+1} also $sat_{n+1}(\vec{\boldsymbol{i}}, \mathcal{S})$ or $sat_{n+1}(\vec{\boldsymbol{i}}, \mathcal{S}')$ holds.

meet A proof by induction on n is analogical to the proof of the previous case.

inclusion We prove that for $n \geq 0$ and $\mathcal{S}, \mathcal{S}', \mathcal{R} \in \mathbb{S}_n$ if $\mathcal{R} = \mathcal{S} \blacklozenge \mathcal{S}'$ then:

$$\mathcal{R} = \epsilon \iff \forall \vec{\boldsymbol{i}} \in \mathbb{I}^n : sat_n(\vec{\boldsymbol{i}}, \mathcal{S}) \Rightarrow sat_n(\vec{\boldsymbol{i}}, \mathcal{S}'). \tag{4.22}$$

We prove this by induction on n. If $n = 0$ the operator is defined as $\mathcal{S} \blacklozenge \mathcal{S}' = \top_0 \iff \mathcal{S} = \top_0$ and $\mathcal{S}' = \epsilon$. Simple case analysis of possible values of $\mathcal{S}$,$\mathcal{S}'$ and $\mathcal{R}$ proves (4.22). Now assume that (4.22) holds for $n \geq 0$. We prove that it holds for $n + 1$.

First, assume that the left hand side of (4.22) holds. By the definition of sat_{n+1} for any $i_{n+1} \in \mathbb{I}$ it holds that $\mathcal{R}[i_{n+1}] = \epsilon$. We apply the induction hypothesis to $\mathcal{R}[i_{n+1}]$ and obtain that for any $\vec{\boldsymbol{i}} = \langle i_n, \dots, i_1 \rangle \in \mathbb{I}^n$ it holds that $sat_n(\vec{\boldsymbol{i}}, \mathcal{S}[i_{n+1}]) \Rightarrow sat_n(\vec{\boldsymbol{i}}, \mathcal{S}'[i_{n+1}])$. Then, by the definition of sat_{n+1} also $sat_{n+1}(\langle i_{n+1}, i_n \dots, i_1 \rangle, \mathcal{S}) \Rightarrow sat_{n+1}(\langle i_{n+1}, i_n \dots, i_1 \rangle, \mathcal{S}')$ holds. Therefore, the right hand side of (4.22) holds.

Now assume the right hand side of (4.22), that is that for any vector $\vec{\boldsymbol{i}} = \langle i_{n+1}, i_n, \dots, i_1 \rangle$ it holds that $sat_{n+1}(\vec{\boldsymbol{i}}, \mathcal{S}) \Rightarrow sat_{n+1}(\vec{\boldsymbol{i}}, \mathcal{S}')$. By the definition of sat_{n+1} we have that $sat_n(\langle i_n, \dots, i_1 \rangle, \mathcal{S}[i_{n+1}]) \Rightarrow sat_n(\langle i_n, \dots, i_1 \rangle, \mathcal{S}'[i_{n+1}])$. By the construction of the $\blacklozenge$-extension and the induction hypothesis we obtain $\mathcal{R}[i_{n+1}] = \epsilon$. Thus, by the definition of sat_{n+1} we have that $\mathcal{R} = \epsilon$. □

Lemma 4.5.6 states how domain operations can be defined with the $\blacklozenge$-extension. Now we finally introduce Theorem 4.5.7, which states that the proposed representation of domain elements using the sweeping line technique, together with the operations defined with $\blacklozenge$-extensions, can be used to implement the domain of *boxes*.

Theorem 4.5.7 (Representation of *Boxes*). *For every $n > 0$ there exists a function $F : \mathbb{BS}_n \to \mathbb{S}_n$ such that for every $\mathcal{BS} \in \mathbb{BS}_n$ and $\vec{\boldsymbol{i}} \in \mathbb{I}^n$ it holds that:*

$$\vec{\boldsymbol{i}} \in \mathcal{BS} \iff sat_n(\vec{\boldsymbol{i}}, F(\mathcal{BS})) \tag{4.23}$$

and F preserves domain operations, that is for every $\mathcal{BS}, \mathcal{BS}' \in \mathbb{BS}_n$ it holds that:

$$\begin{aligned}
&F(\varnothing) = \epsilon,\\
&\mathcal{BS} \subseteq \mathcal{BS} \iff F(\mathcal{BS}) \subseteq_{\blacklozenge} F(\mathcal{BS}'),\\
&F(\mathcal{BS} \cup \mathcal{BS}') = F(\mathcal{BS}) \cup_{\blacklozenge} F(\mathcal{BS}'),\\
&F(\mathcal{BS} \cap \mathcal{BS}') = F(\mathcal{BS}) \cap_{\blacklozenge} F(\mathcal{BS}').
\end{aligned}$$

The idea of the proof is to prove first that representation of a single box is correct, and then to use domain operations to construct the target element of domain of *boxes*. We present a few supplementary lemmas that help in the proof. First, we express interval constraints in terms of our denotations. We use interval constraints terminology introduced in Section 4.2.

Definition 4.5.8 (Higher Boundary). Let variable $v \in \mathit{Var}$ be limited by a higher boundary interval constraint c_H such that $c_H \neq v < \infty$. We say that v is limited by a *higher boundary* $p_H \in \mathbb{P}$ such that:

$$p_H = \begin{cases} (b, \oplus) & \text{if } c_H \equiv v < b\\ (b, \ominus) & \text{if } c_H \equiv v \leq b, \mathbb{I} \neq \mathbb{Z}\\ (b+1, \oplus) & \text{if } c_H \equiv v \leq b, \mathbb{I} = \mathbb{Z}, \end{cases} \tag{4.24}$$

where $b \in \mathbb{I}$.

The next Lemma 4.5.9 states that Definition 4.5.8 is a correct translation of a *higher boundary interval constraint* to our denotations.

Lemma 4.5.9. *Let v, p_H, b and c_H be as in Definition 4.5.8. It holds that:*

$$x \in \mathbb{I} \textit{ fulfils constraint } c_H \iff (x, \oplus) \prec p_H.$$

Proof. Let v, p_H, b and c_H be as in the current lemma assumptions. A simple case analysis of possible constraints c_H, which correspond to the cases in eq. (4.24). In every case the equivalence is proved by the definition of the ordering $\prec$. □

Analogically to Definition 4.5.8, we express a *lower boundary interval constraint* in terms of our denotations:

Definition 4.5.10 (Lower Boundary). Let variable $v \in \mathit{Var}$ be limited by a lower boundary interval constraint c_L such that $c_L \neq -\infty < v$. We say that v is limited by a *lower boundary* $p_L \in \mathbb{P}$ such that:

$$p_L = \begin{cases} (a, \oplus) & \text{if } c_L \equiv a \leq v\\ (a, \ominus) & \text{if } c_L \equiv a < v \text{ and } \mathbb{I} \neq \mathbb{Z}\\ (a+1, \oplus) & \text{if } c_L \equiv a < v \text{ and } \mathbb{I} = \mathbb{Z}, \end{cases} \tag{4.25}$$

where $a \in \mathbb{I}$.

The next Lemma 4.5.11 states that Definition 4.5.10 is a correct translation of a *lower boundary interval constraint* to our denotations.

Lemma 4.5.11. *Let v, p_L, a, and c_L be as in Definition 4.5.10. It holds that:*

$$x \in \mathbb{I} \text{ fulfils constraint } c_L \iff p_L \preceq (x, \oplus).$$

Proof. Let v, p_L, a, and c_L be as in the current lemma assumptions. A simple case analysis of possible constraints c_L, which correspond to the cases in eq. (4.25). In every case the equivalence is proved by the definition of the ordering $\prec$. □

Lemma 4.5.12 presents a construction of an element of $\mathbb{S}_m$ that represents a single non-empty interval box from given interval constraints on variables from *Var*.

Lemma 4.5.12. *Let $\mathcal{B} \in \mathbb{B}_m$ for some $m > 0$ be a non-empty interval box, $Var = \{v_1, \dots, v_m\}$ be a set of variables, $C = \{c_{L,1}, c_{H,1}, \dots, c_{L,m}, c_{H,m}\}$ be a set of interval constraints describing $\mathcal{B}$. Let $\mathcal{S}_0, \mathcal{S}_1, \dots \mathcal{S}_m$ be a sequence defined by the following inductive construction:*

$$\mathcal{S}_0 = \top_0,$$

$$\mathcal{S}_n = \begin{cases} \big((-\infty, \mathcal{S}_{n-1})\big) & \text{if there are no restrictions on } v_n \\ \big((p_{L,n}, \mathcal{S}_{n-1})\big) & \text{if there is only a lower interval constraint for } v_n \\ \big((-\infty, \mathcal{S}_{n-1}), (p_{H,n}, \epsilon)\big) & \text{if there is only a higher interval constraint for } v_n \\ \big((p_{L,n}, \mathcal{S}_{n-1}), (p_{H,n}, \epsilon)\big) & \text{if there are both interval boundaries on } v_n, \end{cases}$$

where $p_{L,n}$ is a lower boundary *for v_n resulting from $c_{L,n}$ as in Definition 4.5.10, and $p_{H,n}$ is a* higher boundary *for v_n resulting from $c_{H,n}$ as in Definition 4.5.8. Then for any $n \in \{1, \dots, m\}$ it holds that for $\mathcal{S}_n \in \mathbb{S}_n$ and $i \in \mathbb{I}$, $\mathcal{S}_n$ fulfils the following condition:*

$$\mathcal{S}_n[i] = \begin{cases} \mathcal{S}_{n-1} & \text{if value } i \text{ fulfils interval constraints on } v_n \\ \epsilon & \text{otherwise.} \end{cases} \tag{4.26}$$

Proof. Let *Var* and C be defined as in the current lemma assumptions. The fact that $\mathcal{S}_n \in \mathbb{S}_n$ for any $n \in 1, \dots, m$ can be easily proved by induction. Let $n \in \{1, \dots, m\}$ and $i \in \mathbb{I}$. We prove eq. (4.26). We analyse cases from the definition of $\mathcal{S}_n$. We have the following possibilities:

1. If there are no restrictions on the variable v_n then $\mathcal{S}_n = \big((-\infty, \mathcal{S}_{n-1})\big)$ and lower and higher interval constraints for v_n are respectively $c_{L,n} \equiv -\infty < v_n$ and $c_{H,n} \equiv v_n < \infty$. The value i trivially fulfils both $c_{L,n}$ and $c_{H,n}$. Also, it holds that $\mathit{find}(i, \mathcal{S}_n) = 1$. Therefore $\mathcal{S}_n[i] = \mathcal{S}_{n-1}$, which proves eq. (4.26).

2. If there is only a lower boundary interval constraint on the variable v_n then $\mathcal{S}_n = \big((p_{L,n}, \mathcal{S}_{n-1})\big)$. Since $c_{H,n} = v_n < \infty$, the value i trivially fulfils the constraint. If i is such that $p_{L,n} \preceq (x, \oplus)$ then by the definition of *find* we have that $\mathit{find}(i, \mathcal{S}_n) = 1$ thus $\mathcal{S}_n[i] = \mathcal{S}_{n-1}$. By Lemma 4.5.11 we have that the constraint $c_{L,n}$ is fulfilled, therefore eq. (4.26) holds. Otherwise, if $(i, \oplus) \prec p_{L,n}$, by Lemma 4.5.11 the constraint $c_{L,n}$ is not fulfilled. By the definition of *find* it holds that $\mathit{find}(x, \mathcal{S}_n) = 0$, thus $\mathcal{S}_n[x] = \epsilon$ and eq. (4.26) holds.

3. If the variable v_n is restricted only by a higher boundary interval constraint then $\mathcal{S}_n = \big((-\infty, \mathcal{S}_{n-1}), (p_{H,n}, \epsilon)\big)$. Since $c_{L,n} = -\infty < v_n$, the value i trivially fulfils the constraint. If i is such that $(i, \oplus) \prec p_{H,n}$ then by Lemma 4.5.9 the constraint $c_{H,n}$ is fulfilled. By the definition of *find* we have that $\mathit{find}(i, \mathcal{S}_{n+1}) = 1$, thus $\mathcal{S}_n[i] = \mathcal{S}_{n-1}$ and eq. (4.26) holds. Otherwise, if $p_{H,n} \preceq (i, \oplus)$ then by Lemma 4.5.9 the constraint $c_{H,n}$ is not fulfilled. Also, we have $\mathit{find}(i, \mathcal{S}_n) = 2$, therefore $\mathcal{S}_n[x] = \epsilon$ and eq. (4.26) holds.

4. If there are both lower and higher boundary interval constraints on v_n then $\mathcal{S}_n = \big((p_{L,n}, \mathcal{S}_{n-1}), (p_{H,n}, \epsilon)\big)$. If $(i, \oplus) \prec p_{L,n}$, by Lemma 4.5.11 the constraint $\mathrm{C}_{L,n}$ is not fulfilled. By the definition of *find* we obtain $\mathit{find}(i, \mathcal{S}_n) = 0$ and $\mathcal{S}_n[i] = \epsilon$, therefore eq. (4.26) holds. If $p_{L,n} \preceq (i, \oplus) \prec p_{H,n}$, by Lemma 4.5.11 and Lemma 4.5.9 both constraints $c_{H,n}$ and $c_{L,n}$ hold. Then we have $\mathit{find}(i, \mathcal{S}_n) = 1$ and $\mathcal{S}_n[x] = \mathcal{S}_{n-1}$, thus eq. (4.26) holds. Finally, if $p_{H,n} \preceq (x, \oplus)$ then by Lemma 4.5.9 the constraint $c_{H,n}$ is not fulfilled. Also, we have $\mathit{find}(i, \mathcal{S}_n) = 2$ and $\mathcal{S}_n[x] = \epsilon$, therefore eq. (4.26) holds. □

Next we prove Lemma 4.5.13, which is an auxiliary lemma, that shows the relation of $sat_k(x, \mathcal{S})$ to the interval constraints for variables from *Var*. The lemma states that for any interval box $\mathcal{B} \in \mathbb{B}_n$ defined by interval constraints from C and any $k \in \{1, \dots, n\}$ we can find an element $\mathcal{S} \in \mathbb{S}_k$ that can be used to represent a smaller box created from variables $v_1, \dots, v_k$ and constraints for these variables.

Lemma 4.5.13. *For* $n > 0$, $\mathcal{B} \in \mathbb{B}_n$, *and* $k \in \{0, \dots, n\}$ *there is* $\mathcal{S} \in \mathbb{S}_k$ *such that for all* $\vec{\imath} \in \mathbb{I}^k$, *where* $\vec{\imath} = \langle i_k, \dots, i_1 \rangle$, *it holds that:*

$$sat_k(\vec{\imath}, \mathcal{S}) \iff \forall_{j \in \{1,\dots,k\}} i_j \textit{ fulfils interval constraints for } v_i \textit{ in } \mathcal{B}. \tag{4.27}$$

Proof. Let $n > 0$, and $\mathcal{B} \in \mathbb{B}_n$ be as in the current lemma assumptions. We define $\mathcal{S} = \mathcal{S}_n$, where $\mathcal{S}_n$ is created using the construction from Lemma 4.5.12. We prove the current lemma by induction on k.

If $k = 0$ then the right hand side of eq. (4.27) is always fulfilled. We choose $\mathcal{S} = \top_0 \in \mathcal{S}_0$, therefore by the definition of sat_0 the left hand side holds.

Now assume that the current lemma holds for $k \geq 0$. We prove that it also holds for $k + 1$. Let $\vec{\boldsymbol{i}} = \langle i_{k+1}, \dots, i_1 \rangle$. First assume the left hand side of eq. (4.27), i.e. that $sat_{k+1}(\vec{\boldsymbol{i}}, \mathcal{S}_{k+1})$ holds. By the definition of sat_{k+1} we have that $\mathcal{S}_{k+1}[i_{k+1}] = \mathcal{S}_k$ for some $\mathcal{S}_k \in \mathbb{S}_k$ such that $sat_k(\langle i_k, \dots, i_1 \rangle, \mathcal{S}_k)$ holds. By Lemma 4.4.5 we have $\mathcal{S}_k \neq \epsilon$. We use the induction hypothesis for $\langle i_k, \dots, i_1 \rangle$ and $\mathcal{S}_k$ to obtain that for all $j \in \{1, \dots, k\}$ it holds that i_j fulfils interval constraints for variable v_j in $\mathcal{B}$. By Lemma 4.5.12 we obtain that i_{k+1} fulfils interval constraints for v_{k+1}, thus the right hand side of eq. (4.27) holds.

Now assume the right hand side of eq. (4.27). Since i_{k+1} fulfils interval constraints for v_{k+1}, by Lemma 4.5.12 we have $\mathcal{S}_{k+1}[i_{k+1}] = \mathcal{S}_k$. We use the induction hypothesis to obtain that $sat_k(\langle i_k, \dots, i_1 \rangle, \mathcal{S}_k)$ holds, therefore by the definition of sat_{k+1} the property $sat_{k+1}(\vec{\boldsymbol{i}}, \mathcal{S}_{k+1})$ holds. □

Finally, Lemma 4.5.14 states that elements of the sequence $\mathbb{S}_n$ can be used to represent elements of domain of intervals.

Lemma 4.5.14. *For $n > 0$ and $\mathcal{B} \in \mathbb{B}_n$ there is exactly one $\mathcal{S} \in \mathbb{S}_n$ such that for all $\vec{\boldsymbol{i}} \in \mathbb{I}^n$ it holds that:*

$$sat_n(\vec{\boldsymbol{i}}, \mathcal{S}) \iff \vec{\boldsymbol{i}} \in \mathcal{B}, \tag{4.28}$$

which is equivalent to:

$$sat_n(\vec{\boldsymbol{i}}, \mathcal{S}) \iff \vec{\boldsymbol{i}} \textit{ fulfils all interval constraints defining } \mathcal{B}. \tag{4.29}$$

Proof. Let $n > 0$ and $\mathcal{B} \in \mathbb{B}_n$. First, we prove that there exists $\mathcal{S} \in \mathbb{S}_n$ that fulfils eq. (4.29). Let $\vec{\boldsymbol{i}} = \langle i_n, \dots, i_1 \rangle$. By Lemma 4.5.13 for $k = n$ there exists $\mathcal{S} \in \mathbb{S}_n$ such that:

$$sat_n(\vec{\boldsymbol{i}}, \mathcal{S}) \iff \forall_{j \in \{1, \dots, n\}} i_j \text{ fulfils interval constraints for } v_j \text{ in } \mathcal{B}.$$

We use Remark 4.2.1 to obtain that:

$$sat_n(\vec{\boldsymbol{i}}, \mathcal{S}) \iff \vec{\boldsymbol{i}} \text{ fulfils all interval constraints defining } \mathcal{B}.$$

Now we only have to prove that there is exactly one $\mathcal{S} \in \mathbb{S}_n$ that fulfils eq. (4.29). We assume the contrary, that there exist $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_n$ such that $\mathcal{S} \neq \mathcal{S}'$ and both fulfil eq. (4.29). By Theorem 4.4.3 there exists $\vec{\boldsymbol{i}} \in \mathbb{I}^n$ such that exactly one of $sat_n(\vec{\boldsymbol{i}}, \mathcal{S})$ and $sat_n(\vec{\boldsymbol{i}}, \mathcal{S}')$ holds. Without loss of generality we may assume that $sat_n(\vec{\boldsymbol{i}}, \mathcal{S})$ holds. Then the right hand side of eq. (4.29) holds. We have that $sat_n(\vec{\boldsymbol{i}}, \mathcal{S}')$ does not hold, therefore the right hand side of eq. (4.29) does not hold. We have obtained a contradiction. □

We proceed with the proof of Theorem 4.5.7. The idea is to define a function that for a single box returns an element of $\mathbb{S}_n$ as defined in Lemma 4.5.14, and then use **join** operation as defined in Lemma 4.5.6 to construct more complex elements of $\mathbb{BS}_n$.

Proof. Let $n > 0$ and $\mathcal{BS}, \mathcal{BS}' \in \mathbb{BS}_n$. First we define operations $\subseteq_\blacklozenge$, $\cup_\blacklozenge$ and $\cap_\blacklozenge$ on $\mathbb{S}_n$ as in Lemma 4.5.6. Let $\mathcal{BS} = \bigcup_{i\in\{1,\dots,k\}} \mathcal{B}_i$, where $\mathcal{B}_i \in \mathbb{B}_n$ and $k \geq 0$. Note that this construction is equivocal, but it does not interfere with our proof: any choice of interval boxes that compose $\mathcal{BS}$ leads to the same result. We define a function $F : \mathbb{BS}_n \to \mathbb{S}_n$ as follows:

$$F(\mathcal{BS}) = \begin{cases} \epsilon & \text{if } k = 0 \\ \mathcal{S}_n & \text{if } k = 1 \\ F(\bigcup_{i\in\{1,\dots,k-1\}} \mathcal{B}_i) \cup_\blacklozenge F(\mathcal{B}_k) & \text{if } k > 1, \end{cases} \tag{4.30}$$

where $\mathcal{S}_n$ is an application of Lemma 4.5.12 for a box $\mathcal{B}_1 \in \mathbb{B}_n$. The function is well-defined because of Theorem 4.5.2.

First, we show that for any $\mathcal{BS} \in \mathbb{BS}_n$ eq. (4.23) holds. We prove this by induction on the number of interval boxes in $\mathcal{BS}$. Let $\mathcal{BS} = \bigcup_{i\in\{1,\dots,k\}} \mathcal{B}_i$, where $\mathcal{B}_i \in \mathbb{B}_n$. If $k = 1$ then $\mathcal{BS} = \mathcal{B}_1$ and $F(\mathcal{BS}) = \mathcal{S}_n$ as defined in Lemma 4.5.12. By Lemma 4.5.14 eq. (4.23) holds. Now assume that eq. (4.23) holds for $k > 0$. We prove that it also holds for $k+1$. Let $\mathcal{BS} = \bigcup_{i\in\{1,\dots,k+1\}} \mathcal{B}_i = (\bigcup_{i\in\{1,\dots,k\}} \mathcal{B}_i) \cup \mathcal{B}_{k+1}$. By the induction hypothesis there is exactly one $\mathcal{S}' \in \mathbb{S}_n$ such that:

$$sat_n(\vec{\boldsymbol{i}}, \mathcal{S}') \iff \vec{\boldsymbol{i}} \in \bigcup_{j\in\{1,\dots,k\}} \mathcal{B}_j.$$

By eq. (4.23) there is exactly one $\mathcal{S} \in \mathbb{S}_n$ such that:

$$sat_n(\vec{\boldsymbol{i}}, \mathcal{S}) \iff \vec{\boldsymbol{i}} \in \mathcal{B}_{k+1}.$$

Now we use the definition of F and apply $\blacklozenge$-extension operator for **join** as defined in Lemma 4.5.6 to obtain $\mathcal{R} \in \mathbb{S}_n$ such that:

$$sat_n(\vec{\boldsymbol{i}}, \mathcal{R}) \iff sat_n(\vec{\boldsymbol{i}}, \mathcal{S}) \text{ or } sat_n(\vec{\boldsymbol{i}}, \mathcal{S}').$$

The element $\mathcal{R}$ is unique by Theorem 4.4.3, therefore eq. (4.23) holds.

By the definition of F it holds that $F(\varnothing) = \epsilon$. Let $\mathcal{BS}, \mathcal{BS}' \in \mathbb{BS}_n$ and $\vec{\boldsymbol{i}} \in \mathbb{I}^n$. By eq. (4.23) and Lemma 4.5.6 we obtain that:

$$sat_n(\vec{\boldsymbol{i}}, F(\mathcal{BS} \cup \mathcal{BS}')) \iff sat_n(\vec{\boldsymbol{i}}, F(\mathcal{BS}) \cup_\blacklozenge F(\mathcal{BS}')).$$

Finally, by Theorem 4.4.3 we obtain that $F(\mathcal{BS} \cup \mathcal{BS}') = F(\mathcal{BS}) \cup_\blacklozenge F(\mathcal{BS}')$. The proof for $F(\mathcal{BS} \cap \mathcal{BS}') = F(\mathcal{BS}) \cap_\blacklozenge F(\mathcal{BS}')$ and $\mathcal{BS} \subseteq \mathcal{BS}' \iff F(\mathcal{BS}) \subseteq_\blacklozenge F(\mathcal{BS}')$ is analogical. □

Theorem 4.5.7 does not state that F is surjective. In fact, the function F as defined in eq. (4.30) is surjective. We present a construction that can be used to prove this fact: for any element $\mathcal{S} \in \mathbb{S}_n$ we give a set of corresponding disjoint interval boxes. The construction is based on an observation that element $\mathcal{S} \in \mathbb{S}_n$ can be considered as a decision tree, where on a dimension k we make a decision about variable v_k: which interval we fall into (the function *find* returns the index of the interval). When we get to a leaf in the tree we have already made decisions about all the variables. If an interval at every decision was not empty, we have created a single interval box.

Split into disjoint boxes

Here we present a construction that can be used to split an element of boxes represented by $\mathcal{S} \in \mathbb{S}_n$ for some $n > 0$ into single, disjoint elements of the domain of *intervals*. The element $\mathcal{S} \in \mathbb{S}_n$ corresponds to a set of disjoint boxes $\mathcal{B}_1, \dots, \mathcal{B}_k$ for some $k \geq 0$ such that:

- for any $i \in \{1, \dots, k\}$ it holds that $\mathcal{B}_i \in \mathbb{B}_n$,
- for any $1 \leq i < j \leq k$ it holds that $\mathcal{B}_i \cap_v \mathcal{B}_j = \bot_v$,
- the sum of these boxes combines to the element $\mathcal{S}$: $\mathcal{S} = \bigcup_{i=1}^{k} \mathcal{B}_i$.

For any $i \in \{1, \dots, k\}$, the box $\mathcal{B}_i$ is described by a path in the tree $\mathcal{S}$ that ends with a leaf $\top_0$. Otherwise, if path ends with ϵ, it describes a kind of "hole box" — a box that does not belong to $\mathcal{S}$.

In the description, we identify a single interval box $\mathcal{B} \in \mathbb{B}_n$ for any $n \geq 0$ with the set of interval constraints defining $\mathcal{B}$. Let C be the set of all possible interval constraints for variables from Var. We introduce an auxiliary function $lb : \mathsf{Var} \times \mathbb{P}_\infty \to \mathrm{C}$ that translates our special point into a lower boundary interval constraint:

$$lb(v, p) = \begin{cases} -\infty < v & \text{if } p = -\infty \\ a \leq v & \text{if } p = (a, \oplus) \\ a < v & \text{otherwise, if } p = (a, \ominus), \end{cases}$$

where $a \in \mathbb{I}$. Analogously, we define a function $hb : \mathsf{Var} \times (\mathbb{P} \cup \{\infty\}) \to \mathrm{C}$ that translates our special point to a corresponding higher boundary interval constraint:

$$hb(v, p) = \begin{cases} v < \infty & \text{if } p = \infty \\ v \leq b & \text{if } p = (b, \ominus) \\ v \leq b - 1 & \text{if } p = (b, \oplus) \text{ and } \mathbb{I} = \mathbb{Z} \\ v < b & \text{otherwise, if } p = (b, \oplus) \text{ and } \mathbb{I} \neq \mathbb{Z}, \end{cases}$$

where $b \in \mathbb{I}$. We introduce a series of functions $split_i : \mathbb{S}_i \to \mathscr{P}(\mathbb{B}_i)$ for $i \in \{0, \dots, n\}$ such that $split_j(\epsilon) = \varnothing$ for any $j \geq 0$ and:

$$split_1(\top_0) = \{\varnothing\},$$
$$split_j(\mathcal{S}_j) = \bigcup_{i=0}^{k} \left\{ \mathrm{C} \cup \{lb(p_i), hb(p_{i+1})\} \mid \mathrm{C} \in split_{i-1}(s_i) \right\},$$

where $j > 0$, $\mathcal{S}_j \in \mathbb{S}_j$ is such that $\mathcal{S}_j = \big((p_1, s_1), (p_2, s_2), \dots, (p_k, s_k)\big)$ for some $k \geq 0$ and $(p_{k+1}, s_{k+1}) = (\infty, \epsilon)$. The function $split_n(\mathcal{S})$ returns the set of disjoint boxes (described as sets of constraints) corresponding to $\mathcal{S} \in \mathbb{S}_n$ as described above. The $split_j$ function simply goes through all paths in the tree $\mathcal{S} \in \mathbb{S}_j$ and collects real boxes and omits "holes".

The presented construction can be used to prove that the function F is surjective: for any element $\mathcal{BS} \in \mathbb{BS}_n$ we can construct a corresponding element $\mathcal{S} \in \mathbb{S}_n$ and using the function *split* go back to $\mathcal{BS}$ showing a possible (a specific one) set of boxes that create $\mathcal{BS}$. Observe that the function F is also injective: since the introduced representation is unique by Theorem 4.4.3, it happens that for any two elements $\mathcal{BS}, \mathcal{BS}' \in \mathbb{BS}_n$ if $\gamma(\mathcal{BS}) = \gamma(\mathcal{BS}')$ then $F(\mathcal{BS}) = F(\mathcal{BS}')$.

4.6 Widening Operator

In the construction of the widening operator for the domain of *boxes* we use a variant of the classical definition of the widening operator, where the second argument is greater or equal to the first one (see Definition 2.4.7). We recall the definition:

Definition 4.6.1 (Widening operator variant). Let $\langle \mathcal{D}, \leq, \bot, \cup \rangle$ be an upper semilattice. A *widening operator* is a partial operator $\triangledown : \mathcal{D} \times \mathcal{D} \rightharpoonup \mathcal{D}$ if and only if the following properties hold:

1) *over-approximation* — for every $d, d' \in \mathcal{D}$, $d \leq d'$ implies that $d \triangledown d'$ is defined and $d' \leq d \triangledown d'$,

2) *stabilisation* — for every increasing chain $d_0 \leq d_1 \leq d_2 \dots$ the increasing chain defined by:

$$\begin{aligned} y_0 &= d_0, \\ y_{n+1} &= y_n \triangledown (y_n \cup d_{n+1}) \text{ for } n \geq 0 \end{aligned} \tag{4.31}$$

is not strictly increasing, i.e. there exists $i \in \mathbb{N}$ such that $y_i = y_{i+1}$.

The sequence $y_0, y_1, \dots$ of consequent results of the widening from eq. (4.31) is called a *widening sequence*.

4.6.1 Widening thresholds

The main idea of the proposed enhancement of the construction of the widening operator is to compute, before every step of the computation of the *widening sequence*, for every variable $v \in \mathsf{Var}$, a special finite set of *threshold points* (or simply *thresholds*). Then, if a refinement is needed, we take into consideration these points. The notion of *widening step thresholds* is introduced formally in Definition 4.6.2.

Definition 4.6.2 (Widening step thresholds). A function $spec_\triangledown : \mathsf{Var} \to 2^{\mathbb{P}}$ is called *widening step thresholds* if for every variable $v \in \mathsf{Var}$ the set $spec_\triangledown(v)$ is finite.

In general, it is not possible to apply *widening step thresholds* for every step of the widening since it might break the infinite chain property. Therefore, we introduce a restriction that the sequence of *widening step thresholds* must also be stationary, i.e. from some point it cannot introduce new *threshold points*. A sequence of *widening step thresholds* that can be used in the proposed widening operator is defined in Definition 4.6.3.

Definition 4.6.3 (Widening sequence thresholds). An infinite sequence of *widening step thresholds* functions: $spec_{\triangledown,1}, spec_{\triangledown,2}, \ldots$ is called *widening sequence thresholds* if there exists $k \geq 0$ such that for any $n > k$ and every $v \in \mathsf{Var}$ it holds that:

$$spec_{\triangledown,n}(v) \subseteq \bigcup_{i=1}^{k} spec_{\triangledown,i}(v).$$

Definition 4.6.3 states that a sequence of *widening step thresholds* starting from the step k cannot introduce new threshold points, i.e. any point for any variable must have been a threshold point in some *widening step threshold* function until step k.

The construction of the widening is recursive. Let $spec_\triangledown$ be a *widening step thresholds* function used in the current step of the widening sequence. A first step of the computation of $\mathcal{R} = \mathcal{S} \triangledown \mathcal{S}'$ for $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_n$ and $n > 0$ is to prepare two sets:

- A smaller set that contains local special points (see eq. (4.19)) from the first argument $\mathcal{S}$, threshold points for the variable v_n and $-\infty$, i.e. $\mathcal{X}_\triangledown = \{-\infty\} \cup spec_L(\mathcal{S}) \cup spec_\triangledown(v_n)$.
- A bigger set, which is the set $\mathcal{X}_\triangledown$ extended by local special points that appear in the second argument of the widening $\mathcal{S}'$, i.e. $\mathcal{X}'_\triangledown = \mathcal{X}_\triangledown \cup spec_L(\mathcal{S}')$.

The computation of the widening is similar to the computation of the ◆-extension. First, we compute a sequence $\mathcal{R}'$ such that first elements of pairs in $\mathcal{R}'$ form the set $\mathbb{X}'_\nabla$, and second elements (values) of pairs are computed *almost* by a pointwise application of the widening for the previous dimension. Then, we perform a sequence normalisation (see Definition 4.5.3) to obtain the result $\mathcal{R}$.

4.6.2 Pointwise approach

In order to compute the value of the ◆-extension for some $x \in \mathbb{I}$ we take values of arguments at the point x, i.e. $\mathcal{S}[x]$ and $\mathcal{S}'[x]$ (see eq. (4.12)), and obtain the result as an application of the ◆-extension to these values. We cannot do the same for the widening operator since we could obtain an infinitely increasing sequence. As an example, consider the following strictly increasing sequence of elements from $\mathbb{S}_1$ and equivalent intervals in $\mathbb{R}$:

Sequence from $\mathbb{S}_1$	**Equivalent interval in $\mathbb{R}$**
$\mathcal{Q}_0 = \big(\,((0,\oplus),\top_0),((1,\ominus),\epsilon)\,\big)$	$[0,1]$
$\mathcal{Q}_1 = \big(\,((0,\oplus),\top_0),((2,\ominus),\epsilon)\,\big)$	$[0,2]$
$\mathcal{Q}_2 = \big(\,((0,\oplus),\top_0),((3,\ominus),\epsilon)\,\big)$	$[0,3]$
$\dots$	$\dots$

Assume that $spec_\nabla = \lambda v\,.\,\varnothing$. When we compute $\mathcal{R}_1 = \mathcal{Q}_0 \nabla \mathcal{Q}_1$ we have:

$$\mathbb{X}_\nabla = \{-\infty,(0,\oplus),(1,\ominus)\} \text{ and } \mathbb{X}'_\nabla = \{-\infty,(0,\oplus),(1,\ominus),(2,\ominus)\}$$

and the result sequence before normalisation is:

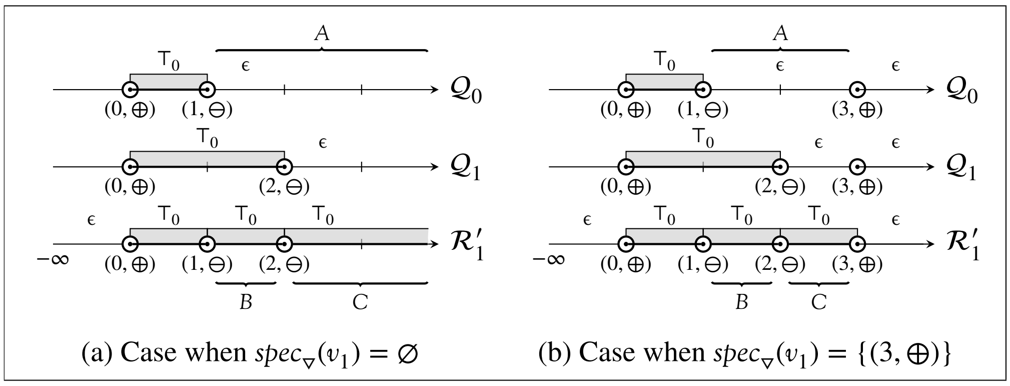
$$\mathcal{R}'_1 = \big((-\infty,w_1),((0,\oplus),w_2),((1,\ominus),w_3)),((2,\ominus),w_4)\big)\,.$$

Now we analyse the situation to figure out the proper definition of the widening. First, assume that we compute values w_1, w_2, w_3, w_4 analogically to the ◆-extension performing a pointwise computation. We obtain $w_1 = \epsilon\nabla\epsilon$, $w_2 = \top_0\nabla\top_0$, $w_3 = \epsilon\nabla\top_0$, and $w_4 = \epsilon\nabla\epsilon = w_1$. Since the widening operator is an over-approximation of both arguments it holds that $w_2 = w_3 = \top_0$. If we say that $w_1 = w_4 = \top_0$ then we obtain $\mathcal{R} = \big((-\infty,\top_0)\big)$, which is the top element in $\mathbb{S}_1$, thus a very imprecise result. Otherwise, if $w_1 = w_4 = \epsilon\nabla\epsilon = \epsilon$, we have a strictly increasing widening sequence:

$$\mathcal{R}_1 = \mathcal{Q}_1 \preceq \mathcal{R}_2 = \mathcal{Q}_2 \dots$$

Therefore, we distinguish w_1 from w_4 and say that $w_1 = \epsilon$ while $w_4 = \top_0$. The situation is presented in Fig. 4.5(a). Intuitively, since the beginning of the interval $[0,1]$ remains unchanged (it is not growing to the left), we do not have to touch it.

The end of the interval is growing and we have to act to ensure termination. This is why we use a refined value for w_4. Here we explain the principle in detail. We focus on the interval $(1, \infty)$, denoted by A in Fig. 4.5(a). The interval A appears in $\mathcal{Q}_0$ but in the result $\mathcal{R}'_1$ it is split into two intervals: $(1, 2]$ denoted as B, and $(2, \infty)$ denoted as C. The split is caused by the point $(2, \ominus)$ from $\mathcal{Q}_1$. The value w_4 is the value associated with the interval C in the result $\mathcal{R}'$. The problem with termination is because the interval C is contained in the interval A, but the value of it's neighbour in A (interval B) has changed, i.e. it is strictly greater. In order to ensure termination, as the refined value for the interval C we take a value that is strictly greater than the value of C in $\mathcal{Q}_1$, i.e. instead of ϵ we use $\top_0$. This rule is generalised by Remark 4.6.4.



(a) Case when $spec_{\nabla}(v_1) = \varnothing$ (b) Case when $spec_{\nabla}(v_1) = \{(3, \oplus)\}$

Figure 4.5: Example application of a single step of the widening operator on $\mathbb{S}_1$.

Remark 4.6.4 (The need of refinement in the widening operator). Assume we have a widening arguments $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_n$ for some $n > 0$ such that $\mathcal{S} \sqsubseteq \mathcal{S}'$. When some interval C in $\mathcal{S}'$ is contained in a bigger interval A in $\mathcal{S}$ with the same value associated then a refinement is required. When we compute the value of the widening for the interval C, as the second value we take a value that is strictly greater than the value of C in $\mathcal{S}'$.

In the situation, when $spec_{\nabla}(v_1) = \{(3, \oplus)\}$, presented in Fig. 4.5(b), both segmentations are more precise, i.e. some intervals are split into smaller ones. The special point $(3, \oplus)$ acts as a threshold in the widening step and the single step result is more precise. Instead of going straight to ∞, the point $(3, \oplus)$ becomes the right end in the result. Of course, when we proceed with the next widening steps for the sequence $\mathcal{Q}_0, \mathcal{Q}_1, \ldots$ we still obtain the ∞, since the *widening sequence thresholds* (see Definition 4.6.3) add only a finite number of threshold points for every variable.

In the abstract interpretation of a program, the widening is applied when loops appear. If we have a **for** loop to an integer constant (see Section 5.5.2) we could add the constant as a threshold for the control variable and the result of the whole widening sequence could be more precise.

4.6.3 Generic *widening operator* for *boxes*

In this section we propose a widening operator for the domain of *boxes* that employs *widening sequence thresholds*. First, we introduce notion of segmentation of a numeric space $\mathbb{I}$.

Definition 4.6.5 (Segmentation of $\mathbb{I}$). Let $\mathbb{X} = \{p_1, \dots, p_m\}$ for some $m > 0$ be a finite subset of $\mathbb{P}_\infty$ such that $-\infty = p_1 \prec p_2 \prec \dots \prec p_m$. We define a finite sequence $I_1, \dots, I_m$ of disjoint, non-empty subsets of $\mathbb{I}$ as follows:

$$I_i = \begin{cases} \{x \in \mathbb{I} \mid in(p_i, x, p_{i+1})\} & \text{when } 1 \leq i < m \\ \{x \in \mathbb{I} \mid p_m \preceq (i, \oplus)\} & \text{if } i = m. \end{cases}$$

The set $\mathcal{I}(\mathbb{X}) = \{I_1, \dots, I_m\}$ is called a *segmentation by* $\mathbb{X}$ and elements of the set are called *segments*. We say that element $x \in \mathbb{I}$ is a *representative of i-th segment in the segmentation by* $\mathbb{X}$ if $x \in I_i$.

For any segmentation $\mathcal{I}$ of $\mathbb{I}$ it holds that any $x \in \mathbb{I}$ is a member of exactly one of the segments in the segmentation. By $\mathcal{I}(\mathbb{X})[x]$ we denote the segment I_j in the segmentation $\mathcal{I}(\mathbb{X})$ such that $x \in I_j$.

Remark 4.6.6 states a relationship between segments of two segmentations of $\mathbb{I}$ by sets $\mathbb{X}$,Y such that $\mathbb{X} \subseteq Y$.

Remark 4.6.6. Let there be two segmentations of $\mathbb{I}$: $\mathcal{I}(\mathbb{X})$ and $\mathcal{I}(Y)$, where $\mathbb{X} \subseteq Y$. The segmentation $\mathcal{I}(Y)$ is more precise than $\mathcal{I}(\mathbb{X})$, i.e. for any segment $I \in \mathcal{I}(\mathbb{X})$ it holds that:

- either $I \in \mathcal{I}(Y)$,
- or there is a finite number of segments $I_1, \dots, I_k$ in $\mathcal{I}(Y)$ such that $I = \bigcup_{i=0}^{k} I_k$ — the segment I is split into smaller segments $I_1, \dots, I_k$.

For two segmentations $\mathcal{I}(\mathbb{X})$, $\mathcal{I}(Y)$ such that $\mathbb{X} \subseteq Y$ for any $x \in \mathbb{I}$ it holds that $\mathcal{I}(Y)[x] \subseteq \mathcal{I}(\mathbb{X})[x]$. When $\mathcal{I}(Y)[x] \subset \mathcal{I}(\mathbb{X})[x]$ then the segment $\mathcal{I}(\mathbb{X})$ is split into smaller segment in the more precise segmentation $\mathcal{I}(Y)$.

Now we formally define the widening operator. Let $spec_\triangledown$ be the *widening step thresholds* for the current step of the application of the widening. Let $\triangledown : \mathbb{S}_0 \times \mathbb{S}_0 \rightarrow$

$\mathbb{S}_0$ be such that $\mathcal{S}_0 \nabla \mathcal{S}'_0 = \mathcal{S}_0 \cup_{\blacklozenge} \mathcal{S}'_0$ for $\mathcal{S}_0, \mathcal{S}'_0 \in \mathbb{S}_0$. We extend the operator to $\mathbb{S}_n$ for any $n > 0$ by an inductive construction. Let $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_n$. We compute:

$$\mathfrak{X}_\nabla = \{-\infty\} \cup spec_L(\mathcal{S}) \cup spec_\nabla(v_n), \qquad \mathfrak{X}'_\nabla = \mathfrak{X}_\nabla \cup spec_L(\mathcal{S}').$$

Let $\mathfrak{X}'_\nabla = \{p_1, p_2, \dots, p_m\}$ for some $m > 1$ such that $p_1 \prec p_2 \prec \dots, p_m$. Let elements $x_1, \dots, x_m$ be representatives of consequent segments $I_1, \dots, I_m$ in the segmentation $\mathcal{I}(\mathfrak{X}'_\nabla)$, i.e:

$$-\infty = p_1 \preceq (x_1, \oplus) \prec p_2 \preceq \dots \prec p_m \preceq (x_m, \oplus).$$

We define ∇-extension, $\nabla : \mathbb{S}_{n+1} \times \mathbb{S}_{n+1} \rightharpoonup \mathbb{S}_{n+1}$ as $\mathcal{S} \nabla \mathcal{S}' = \mathcal{R}$ such that $\mathcal{R} = \mathcal{S}'$ if $\mathcal{S} = \epsilon$, and otherwise we define:

$$\mathcal{R}' = \big((p_1, v_1), \dots, (p_m, v_m)\big)$$

such that for any $i \in \{1, \dots, m\}$:

$$v_i = \begin{cases} \mathcal{S}[x_i] \nabla_n \mathcal{S}'[x_i] & \text{if } \mathcal{S}[x_i] \neq \mathcal{S}'[x_i] & (4.32a) \\ \mathcal{S}[x_i] & \text{if } \mathcal{S}[x_i] = \mathcal{S}'[x_i] \text{ and } I_i \in \mathcal{I}(\mathfrak{X}_\nabla) & (4.32b) \\ \mathcal{S}[x_i] \nabla_n W_i & \text{otherwise,} & (4.32c) \end{cases}$$

where $W_i \in \mathbb{S}_n$ is such that $\mathcal{S}[x_i] \subset W_i$.

The first case eq. (4.32a) is straightforward. The second case eq. (4.32b) handles situation when the segment I_i that contains element x_i appears in both segmentations $\mathcal{I}(\mathfrak{X}_\nabla)$ and $\mathcal{I}(\mathfrak{X}'_\nabla)$. Then we just do a pointwise application of the widening. We could compute the widening as $\mathcal{S}[x_i] \nabla_n \mathcal{S}'[x_i]$, but since arguments are equal the result would be the same so the case is simplified. The last case eq. (4.32c) is when $I_j = \mathcal{I}(\mathfrak{X}'_\nabla)[x_i] \subset \mathcal{I}(\mathfrak{X}_\nabla)[x_i]$, thus a refinement is required. Here we present a generic condition for the refinement that is enough for ∇ to be a widening operator. Note that refined arguments: W_i for $i \in \{1, \dots, m\}$, are specific for this particular application of the widening operator (similarily to $spec_\nabla$). Theorem 4.6.10 applies to this generic construction, however in Section 4.6.4 we present an instance of the widening that has some interesting properties.

Example: one step of the application of the *widening operator*

We explain the cases of the construction on the example presented in Fig. 4.6. We have:

$$spec_L(\mathcal{S}) = \{p_3, p_6\}, \qquad spec_L(\mathcal{S}') = \{p_2, p_4, p_7\}.$$

Since $spec_\nabla(v) = \{p_5\}$ and $p_1 = -\infty$, we compute:

$$\mathfrak{X}_\nabla = \{-\infty, p_3, p_5, p_6\}, \qquad \mathfrak{X}'_\nabla = \{-\infty, p_2, p_3, p_4, p_5, p_6, p_7\}.$$

Next, we compute $\mathcal{R}' = \big((p_1, v_1), \dots, (p_7, v_7)\big)$ (we recall $p_1 = -\infty$), where:

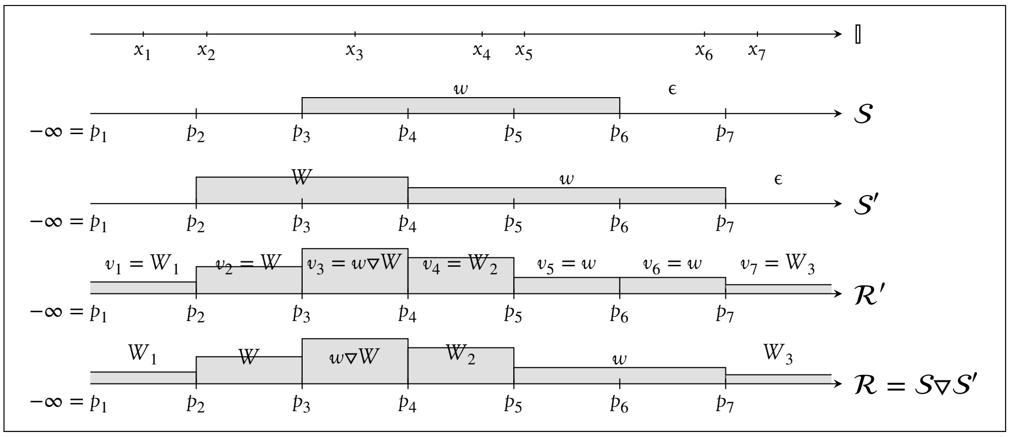

x_1 x_2 x_3 x_4 x_5 x_6 x_7
$\mathbb{I}$
w
ϵ
$\mathcal{S}$
$-\infty = p_1$ p_2 p_3 p_4 p_5 p_6 p_7
W
w
ϵ
$\mathcal{S}'$
$v_1 = W_1$ $v_2 = W$ $v_3 = w\nabla W$ $v_4 = W_2$ $v_5 = w$ $v_6 = w$ $v_7 = W_3$
$\mathcal{R}'$
W_1 W $w\nabla W$ W_2 w W_3
$\mathcal{R} = \mathcal{S}\nabla\mathcal{S}'$

Figure 4.6: Example that covers different cases in a single step of the generic construction of the widening operator for the *widening step thresholds* function: $spec_\nabla(v) = \{p_5\}$ for any $v \in Var$. Heights of bars correlate to values.

- Value v_1: $\mathcal{S}[x_1] = \mathcal{S}'[x_1]$. Since $p_2 \notin \mathcal{X}_\nabla$ it holds that $\mathcal{I}(\mathcal{X}'_\nabla)[x_1] \subset \mathcal{I}(\mathcal{X}_\nabla)[x_1]$ and we apply eq. (4.32c). As the second argument we use value W_1 such that $\epsilon \subset W_1$.
- Value v_2: since $\mathcal{S}[x_2] = \epsilon \subset W = \mathcal{S}'[x_2]$ we apply eq. (4.32a).
- Value v_3: since $\mathcal{S}[x_3] = w \subset W = \mathcal{S}'[x_3]$ we apply eq. (4.32a).
- Value v_4: $\mathcal{S}[x_4] = \mathcal{S}'[x_4]$. Since $p_4 \notin \mathcal{X}_\nabla$ it holds that $\mathcal{I}(\mathcal{X}'_\nabla)[x_4] \subset \mathcal{I}(\mathcal{X}_\nabla)[x_4]$ and we apply eq. (4.32c). As the second argument we use value W_2 such that $w \subset W_2$.
- Value v_5: $\mathcal{S}[x_5] = \mathcal{S}'[x_5]$. Since both $p_5 \in \mathcal{X}_\nabla$ and $p_6 \in \mathcal{X}_\nabla$ it holds that $\mathcal{I}(\mathcal{X}'_\nabla)[x_5] = \mathcal{I}(\mathcal{X}_\nabla)[x_5]$ and we apply eq. (4.32b).
- Value v_6: since $\mathcal{S}[x_6] = \epsilon \subset w = \mathcal{S}'[x_6]$ we apply eq. (4.32a).
- Value v_7: $\mathcal{S}[x_7] = \mathcal{S}'[x_7]$. Since $p_7 \notin \mathcal{X}_\nabla$ it holds that $\mathcal{I}(\mathcal{X}'_\nabla)[x_7] = \mathcal{I}(\mathcal{X}_\nabla)[x_7]$ and we apply eq. (4.32c). As the second argument we use value W_3 such that $\epsilon \subset W_3$.

The final result $\mathcal{R}$ is a normalised version of $\mathcal{R}'$, i.e.

$$\mathcal{R} = \big((p_1, W_1), (p_2, W), (p_3, w\nabla W), (p_4, W_2), (p_5, w), (p_7, W_3)\big) .$$

Next, we present a few remarks about the proposed construction.

Remark 4.6.7. When a refinement is done, a segment from $\mathcal{S}$ is replaced by finitely many segments, for which value is strictly greater than the original one.

Remark 4.6.8 states that every first element in pairs in the sequence that is the result of the application the widening comes either from one of the two arguments or from *widening step thresholds*.

Remark 4.6.8. Let $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_n$ for $n > 0$ be such that $\mathcal{S} \subseteq_{\blacklozenge} \mathcal{S}'$, $spec_{\triangledown}$ be *widening step thresholds* for the widening step and v_n be the variable for n-th dimension. Then let $\mathcal{R} = \mathcal{S} \triangledown \mathcal{S}'$. It holds that:

$$spec_L(\mathcal{R}) \subseteq spec_L(\mathcal{S}) \cup spec_L(\mathcal{S}') \cup spec_{\triangledown}(v_n). \tag{4.33}$$

The remark is a consequence of the construction of the widening. The unnormalised sequence $\mathcal{R}'$ is created, so that first elements come from either of the arguments and the normalisation might remove some of the pairs in $\mathcal{R}'$.

Consider a widening sequence with some *widening sequence thresholds*. Assume that after first k steps the *widening sequence thresholds* stabilises, i.e. no new threshold points are added anymore. Then, in the following widening steps the set of special points depends only on special points from arguments. The fact is stated formally by Remark 4.6.9.

Remark 4.6.9. Let m be the index from the definition of *widening sequence thresholds* and j be the step of the widening sequence such that $j > m$. Let arguments of the step j be $\mathcal{S}, \mathcal{S}'$. Then all new special points that appear in the result sequence $\mathcal{R} = \mathcal{S} \triangledown \mathcal{S}'$ come from one of the arguments, i.e.: $spec_L(\mathcal{R}) \subseteq spec_L(\mathcal{S}) \cup spec_L(\mathcal{S}')$.

Theorem 4.6.10 (Widening). *The $\triangledown$-extension is a proper widening operator, i.e. for any $n \geq 0$ the tuple $\langle \mathbb{S}_n, \subseteq_{\blacklozenge}, \epsilon, \cup_{\blacklozenge} \rangle$ fulfils the conditions of Definition 4.6.1.*

First, we present a lemma that we use in the proof of Theorem 4.6.10:

Lemma 4.6.11 (König's Lemma). *Let T be a rooted directed tree. If each vertex in T has finite degree but there are arbitrary long rooted paths in T, then T contains an infinite path.*

Proof. See [71, Chapter VI, Lemma 10]. □

We proceed with the proof of Theorem 4.6.10:

Proof. We prove properties from Definition 4.6.1:

Over-approximation

First, we prove that over-approximation property (1) from Definition 4.6.1 holds for the proposed $\triangledown$-extension. Proof by induction on n. For $n = 0$ and $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_0$

a simple case analysis proves the current property. Now assume the property holds for $n \geq 0$. We prove that it also holds for $n + 1$. Let $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_{n+1}$, where $\mathcal{S} \subseteq_{\blacklozenge} \mathcal{S}'$. If $\mathcal{S} = \epsilon$ then $\mathcal{S} \nabla \mathcal{S}' = \mathcal{S}'$, thus the property holds. Otherwise, let $x \in \mathbb{I}$ and $\mathcal{R} = \mathcal{S} \nabla \mathcal{S}'$. Then in any case of the definition of the ∇-extension, it holds that $\mathcal{S}'[x] \subseteq_{\blacklozenge} \mathcal{R}[x]$ (by the induction hypothesis), therefore $\mathcal{S}' \subseteq_{\blacklozenge} \mathcal{R}$.

Stabilisation
Let $\mathcal{Q}_1 \subseteq_{\blacklozenge} \mathcal{Q}_2 \subseteq_{\blacklozenge} \mathcal{Q}_3 \dots$ be an infinite increasing sequence of elements from $\mathbb{S}_n$ for some $n \geq 0$, as in property 2) of Definition 4.6.1. We construct two infinite sequences $\mathcal{S}_1, \mathcal{S}_2, \dots$ and $\mathcal{R}_1, \mathcal{R}_2, \dots$ made of elements from $\mathbb{S}_n$. The first one consists of second arguments from *widening sequence* in eq. (4.31) and the second one is *widening sequence* itself. More formally:

$$\mathcal{S}_k = \begin{cases} \mathcal{Q}_1 & \text{if } k = 1 \\ \mathcal{R}_{k-1} \cup_{\blacklozenge} \mathcal{Q}_k & \text{if } k > 1, \end{cases} \qquad \mathcal{R}_k = \begin{cases} \mathcal{S}_1 & \text{if } k = 1 \\ \mathcal{R}_{k-1} \nabla \mathcal{S}_k & \text{if } k > 1. \end{cases}$$

Note that it follows directly from the definition that $\mathcal{R}_{k-1} \subseteq_{\blacklozenge} \mathcal{S}_k$. Moreover, the sequence $\mathcal{S}_1, \mathcal{S}_2, \dots$ is increasing, i.e. for any $i \geq 0$ it holds that $\mathcal{S}_i \subseteq_{\blacklozenge} \mathcal{S}_{i+1}$. The *over-approximation property* implies that $\mathcal{R}_{k-1} \subseteq_{\blacklozenge} \mathcal{R}_k$. We prove that the sequence $\mathcal{R}_1, \mathcal{R}_2, \dots$ is a proper *widening sequence*, i.e. it is not strictly increasing. In the proof we focus on consequent arguments appearing of the sequence, that is on elements of $\mathcal{S}_1, \mathcal{S}_2, \dots$ Let $spec_{\nabla,1}, spec_{\nabla,2}, \dots$ be *widening sequence thresholds* used for computing the sequence $\mathcal{R}_1, \mathcal{R}_2, \dots$

Proof by induction on n, which is the dimension of elements in the *widening sequence*. For $n = 0$ there is no infinite strictly increasing sequence, because there are only two elements in the domain, thus the stabilisation property holds.

Now assume the stabilisation property holds for n, we prove that it also holds for $n+1$. Proof by contradiction — we assume that the sequence $\mathcal{R}_1, \mathcal{R}_2, \dots$ is strictly increasing. We create a sequence of sets $T_0, T_1, \dots$ as follows:

$$T_k = \begin{cases} \{-\infty, \infty\} & \text{if } k = 0 \\ spec_{\nabla,k}(v_{n+1}) \cup spec_L(\mathcal{S}_i) \cup T_{k-1} & \text{if } k > 0. \end{cases} \tag{4.34}$$

The set T_k for $k > 0$ is the set of special points that might appear in the sequence $\mathcal{R}_k$. It is the set of all special points that appeared in elements $\mathcal{S}_i$ for $i \in \{1, \dots, k\}$ extended by threshold points for the variable $v_{n+1} \in \mathit{Var}$ from all widening steps from 1 to k, extended by $\{\infty\}$, i.e.:

$$T_k = \{\infty\} \cup \bigcup_{i=1}^{k} spec_L(\mathcal{S}_i) \cup spec_{\nabla,i}(v_{n+1}).$$

By Remark 4.6.8 and eq. (4.34) for any $k > 0$ it holds that $spec_L(\mathcal{R}_k) \subseteq T_k$. We introduce the following notation for elements of T_k for any $k \geq 0$:

$$T_k = \{p_{k,1}, \dots, p_{k,|T_k|}\}, \text{ where } \forall j \in \{1, \dots, |T_k| - 1\} : p_{k,j} \prec p_{k,j+1}.$$

The graph of direct refinements

Now we create a graph of refinements that are applied to segments of elements in the sequence $\mathcal{R}_1, \mathcal{R}_2, \dots$ The graph $\mathrm{G}(V, E)$ is such that:

Vertices. The set of vertices V is the set of quadruples (p, w, q, i) for $i \geq 0$ that describe all possible segments that might have appeared during the widening sequence. The quadruple (p, w, q, k) corresponds to the segment $I^k_{p,q}$ that appears in the segmentation by the set T_k (the segmentation $\mathcal{I}(T_k)$) in the computation the result of the k-th step of the widening sequence – $\mathcal{R}_k$. More formally:

$$I^k_{p,q} = \{x \in \mathbb{I} \mid in(p, x, q)\} \text{ for } q \neq \infty,$$
$$I^k_{p,\infty} = \{x \in \mathbb{I} \mid p \preceq (x, \oplus)\}.$$

The segments $I^k_{p_{k,1},p_{k,2}}, I^k_{p_{k,2},p_{k,3}}, \dots, I^k_{p_{k,|T_k|-1},p_{k,|T_k|}}$ are consequent segments in the segmentation $\mathcal{I}(T_k)$.

The set of vertices describes *possible* segments, i.e. the segment $I^k_{p,q}$ need not appear directly in the segmentation $\mathcal{I}(spec_L(\mathcal{R}_k))$ but since $spec_L(\mathcal{R}_k) \subseteq T_k$, by Remark 4.6.6, the segment is a subsegment of some segment in that segmentation. All elements $x \in I^k_{p,q}$ have the same value in $\mathcal{R}_k$, i.e. there exists $w \in \mathbb{S}_{n-1}$ such that for any $x \in I^k_{p,q}$ it holds $\mathcal{R}_k[x] = w$. The value w is the second element in the quadruple.

Formally, the set of all vertices in G is defined as follows:

$$V = \bigcup_{i \geq 0} V_k,$$

where $V_k = \{(p, w, q, k) \mid I^k_{p,q} \in \mathcal{I}(T_k) \text{ and } w = \mathcal{R}_k[x], \text{ where } x \in I^k_{p,q}\}$.

Edges. The set of edges $E \subseteq V \times V$ describes refinements that are performed when the sequence $\mathcal{R}_1, \mathcal{R}_2, \dots$ is computed. We connect a vertex (p, w, q, i) with a vertex (p', w', q', j), when the source vertex is from the step 0 and target from step 1 or all the following requirements are fulfilled:

1. The target vertex comes from the step following (not necessarily directly) the step of the source vertex, i.e. $i < j$.

2. The segment associated with the target vertex is a subset of the segment associated with the source, i.e. $I^j_{p',q'} \subseteq I^i_{p,q}$.

3. There was a refinement — the target vertex has value strictly greater than the source one, i.e. $w \subset_\blacklozenge w'$.

4. In the intermediate vertices between the two connected ones there was no other refinement done. The step j is the first step since i, where the refinement for the segment $I^j_{p',q'}$ was made, i.e. for any step $k \in \{i+1, \ldots, j-1\}$ and $x \in I^j_{p',q'}$ it holds that $\mathcal{R}_k[x] = w$.

5. A vertex has no output edges when it has no input edges, i.e. graph input degree of the source vertex must be greater than 0: $\sharp_{in}((p, w, q, i)) > 0$.

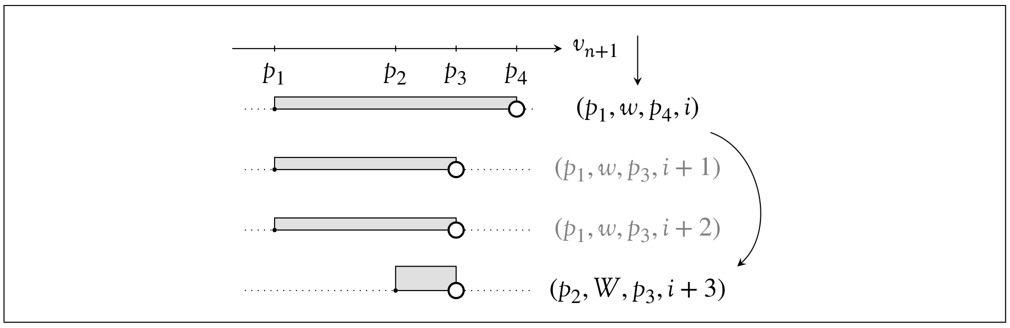


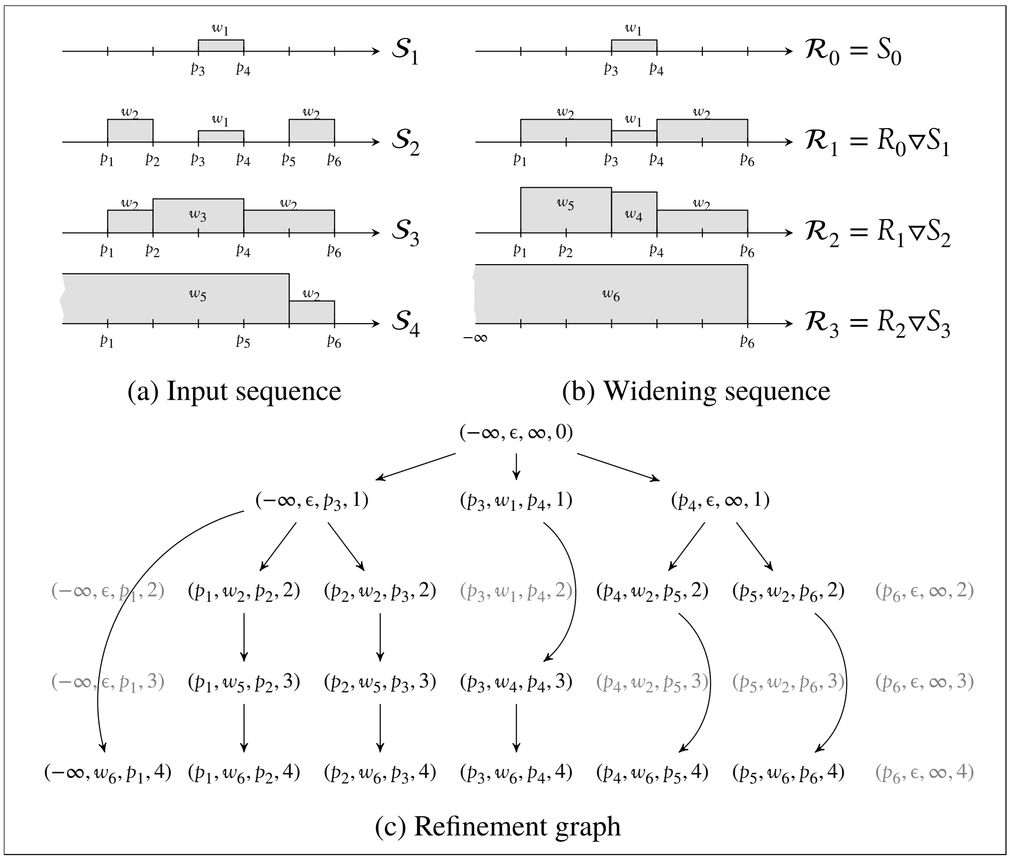
Figure 4.7: Illustration of how the set of edges E is created.

A simple illustration of how the set E is constructed is presented in Fig. 4.7. We say that the vertex (p_1, w, p_4, i) is the *first vertex*, the following two vertices $(p_1, w, p_3, i+1)$ and $(p_1, w, p_3, i+2)$ are *intermediate vertices*, and finally the vertex $(p_2, W, p_3, i+3)$ is the *last vertex*. We assume the *first vertex* has some input edge. All *intermediate vertices* and the *last vertex* describe segments that are subsets of the one described by the *first vertex*, therefore according to the condition 2 there are potential edges going from the *first vertex* to any of them. Since two *intermediate vertices* have the same value as the *first vertex*, there are no edges going to them (condition 4). Since there are no edges going to *intermediate vertices*, there are no edges going out (condition 5) — that is why these vertices are displayed in grey. There is an edge from the *first vertex* to the *last vertex* since all intermediate vertices have value for the segment displayed by the *last vertex* the same as the value of the *first vertex*: w.

To sum up, the set of edges E is defined as follows:

$$\begin{aligned}E = \{ \big((p,w,q,i),(p',w',q',j)\big) \mid\ & (i=0 \text{ and } j=1) \text{ or} \\ & (0<i<j \text{ and } I^j_{p',q'} \subseteq I^i_{p,q} \text{ and } w \subset_{\blacklozenge} w' \text{ and} \\ & \forall_{k\in\{i,\dots,j-1\}} \mathcal{R}_k[x] = \mathcal{R}_i[x], \text{ where } x \in I^j_{p',q'} \text{ and} \\ & \sharp_{in}((p,w,q,i)) > 0)\}.\end{aligned} \tag{4.35}$$

An example of a graph created for some widening sequence is presented in Fig. 4.8. Note that the graph $G(V',E')$ created from $G(V,E)$ such that $V' = \{v \in V \mid \sharp_{in}(v) + \sharp_{out}(v) > 0\}$ and $E' = E$ is a tree.


w_1
p_3 p_4
$\mathcal{S}_1$
w_2 w_1 w_2
p_1 p_2 p_3 p_4 p_5 p_6
$\mathcal{S}_2$
w_2 w_3 w_2
p_1 p_2 p_4 p_6
$\mathcal{S}_3$
w_5 w_2
p_1 p_5 p_6
$\mathcal{S}_4$
$\mathcal{R}_0 = S_0$
w_2 w_1 w_2
p_1 p_3 p_4 p_6
$\mathcal{R}_1 = R_0 \triangledown S_1$
w_5 w_4 w_2
p_1 p_2 p_4 p_6
$\mathcal{R}_2 = R_1 \triangledown S_2$
w_6
$-\infty$ p_6
$\mathcal{R}_3 = R_2 \triangledown S_3$
$(-\infty,\epsilon,\infty,0)$
$(-\infty,\epsilon,p_3,1)$ $(p_3,w_1,p_4,1)$ $(p_4,\epsilon,\infty,1)$
$(-\infty,\epsilon,p_1,2)$ $(p_1,w_2,p_2,2)$ $(p_2,w_2,p_3,2)$ $(p_3,w_1,p_4,2)$ $(p_4,w_2,p_5,2)$ $(p_5,w_2,p_6,2)$ $(p_6,\epsilon,\infty,2)$
$(-\infty,\epsilon,p_1,3)$ $(p_1,w_5,p_2,3)$ $(p_2,w_5,p_3,3)$ $(p_3,w_4,p_4,3)$ $(p_4,w_2,p_5,3)$ $(p_5,w_2,p_6,3)$ $(p_6,\epsilon,\infty,3)$
$(-\infty,w_6,p_1,4)$ $(p_1,w_6,p_2,4)$ $(p_2,w_6,p_3,4)$ $(p_3,w_6,p_4,4)$ $(p_4,w_6,p_5,4)$ $(p_5,w_6,p_6,4)$ $(p_6,\epsilon,\infty,4)$

(a) Input sequence (b) Widening sequence

(c) Refinement graph

Figure 4.8: An example of a refinement graph.

We prove that every node in G has finitely many children.

Every node in* G *has finite output degree

Let m be an iteration number from the definition of *widening sequence thresholds* such that since the step m no new threshold points are added. Let $(p, w, q, i) \in V$ be a vertex for the step $i \geq 0$ and I denote the segment described by the vertex, i.e. $I = I^i_{p,q}$. We prove that the vertex (p, w, q, i) has finitely many children. Proof by contradiction. Assume the contrary, that the vertex has infinitely many children in G. We have the following cases:

- First we assume that $i > m$. By Remark 4.6.9 all new points that appear in T_j for $j \geq i$ come from the second argument of the widening in step j and *widening sequence thresholds* does not introduce any new split points. Let j be the lowest index among all the children of (p, w, q, i). We prove that all children have the same index j.

  In steps between i and j there was no refinement done to the segment I. This might have only happened by the application of the case eq. (4.32b) to the segment, in which I is included in these intermediate steps. Therefore, the segment I is not partitioned in the intermediate steps. Thus, it holds that $I \in T_{j-1}$. Let P be the set of partition points of the segment I in the step j. It holds that $P \subseteq T_j \setminus T_{j-1}$.

  If $P = \emptyset$ then whole segment I is refined at once, thus there is only one child of (p, w, q, i) and we obtain contradiction. Otherwise $P \neq \emptyset$. Then I is split into smaller subsegments. Let I_l be one of subsegments in the split. It holds that $I_l \subseteq I'_l$, where I'_l is a segment in the segmentation $\mathcal{I}(\mathbb{P}'_\nabla)$ from the step j. One of the cases in eq. (4.32) is applied to I'_l. Note that since the split was performed, the case eq. (4.32b) cannot be applied. Therefore, either the case eq. (4.32a) or eq. (4.32c) is applied. In either of them, the segment I'_l is refined. Since $I_l \subseteq I'_l$, the segment I_l is also refined, and thus all subsegments of I are refined in the step j. Therefore, the vertex (p, w, q, i) has finite number of children (every one corresponds to one of subsegments of I in the split in step j).

- Otherwise $i \leq m$. We consider the situation after step m. The segment I may be split into a number of subsegments during intermediate steps $i + 1, \dots, m$. Since T_m is finite, the number of these subsegments is finite. For every subsegment $I_k \subseteq I$ that was not refined until step m we apply reasoning analogical to the previous case. All subsegments of I that are included in I_k are refined in the same step of the widening. Therefore, (p, w, q, i) has finite number of children in G.

Termination

We recall our assumption that the infinite sequence $\mathcal{R}_1, \mathcal{R}_2, \dots$ created from elements from $\mathbb{S}_{n+1}$ is strictly increasing. Therefore, in every widening step a refinement is done and at least one new edge is added to G. Hence, in G there is a finite path of any length. The graph of refinements $G(V', E')$ obtained from $G(V, E)$ such that $V' = \{v \in V \mid \sharp_{in}(v) + \sharp_{out}(v) > 0\}$ and $E' = E$ is a tree. The tree G' is created from the graph G by removing vertices without any edge. By König's lemma (see Lemma 4.6.11) there is an infinite path in G'. Let the path be $(-\infty, \epsilon, \infty, 0), (p_1, w_1, q_1, i_1), (p_2, w_2, q_2, i_2), \dots$ By eq. (4.35) we obtain that the sequence $w_1, w_2, \dots$ is strictly increasing, i.e. $w_1 \subset_{\blacklozenge} w_2 \subset_{\blacklozenge} \dots$ The sequence is created from elements from $\mathbb{S}_n$. We use the induction hypothesis to obtain a contradiction — there is no infinite, strictly increasing widening sequence in $\mathbb{S}_n$. Therefore, the sequence $\mathcal{R}_1, \mathcal{R}_2, \dots$ is not strictly increasing, and thus $\triangledown$ is a proper widening operator. □

Note that the proposed definition of the widening for *boxes* is very generic. There are two places, where different strategies may be applied. The first one is the choice of the value in the case eq. (4.32c). We propose a new strategy in Section 4.6.4. The second place is the choice of *widening sequence thresholds*. We give a few ideas for building the sequence of functions.

Choosing *widening sequence thresholds*
There are many ways how to build *widening sequence thresholds*. It might be computed once at the beginning of the whole widening sequence, e.g.:

- One could gather *global special points* from the first argument that appears in the widening sequence. Let $n > 0$. The *global special points* function $spec_G : \bigcup_{i=1}^{n} \mathbb{S}_i \to Var \to 2^{\mathbb{P}_\infty}$ is defined as follows:

$$spec_G(\mathcal{S})(v_i) = \begin{cases} \emptyset & \text{if } \mathcal{S} = \epsilon \\ spec_L(\mathcal{S}) & \text{if } \mathcal{S} \in \mathbb{S}_i \\ \bigcup_{j \in \{1,\dots,k\}} spec_G(v, s_j) & \text{otherwise,} \end{cases} \tag{4.36}$$

  where $\mathcal{S} = \big((p_1, s_1), \dots, (p_k, s_k)\big)$ for some $k \geq 1$. Let $\mathcal{S}_0 \in \mathbb{S}_n$ be the first argument that appears in the widening sequence. Then we define $spec_{\triangledown,j} = spec_G(\mathcal{S}_0)$ for any $j >= 1$.

- The function $spec_{\triangledown,j}$ for any $j \geq 1$ of special points can be based on the source code of the analysed software, e.g. it may contain all constants that appear in the code.

The definition of *widening sequence thresholds* does not prohibit dynamic computation of consequent elements based on the result of previous widening step. The only restriction is that the sequence must be stationary at some point. One of the proposed strategies is to dynamically build the sequence. In the following example we update consequent elements with global special points of widening arguments $k > 0$ times. Let $n > 0$ and $\mathcal{S}_0, \mathcal{S}_1, \dots$ be the sequence of widening arguments. We define *widening sequence thresholds* by the following inductive construction:

$$spec_{\nabla,i} = \begin{cases} spec_G(\mathcal{S}_0) & \text{if } i = 1 \\ \lambda v \,.\, spec_{\nabla,i-1}(v) \cup spec_G(\mathcal{S}_{i-1})(v) & \text{if } 1 < i \leq k \\ spec_{\nabla,i-1} & \text{otherwise.} \end{cases}$$

It is also possible to combine code analysis and the dynamic computation of *widening sequence thresholds*.

4.6.4 Refinement strategy for the *widening operator*

In this section we present a specific implementation of the refinement strategy in eq. (4.32)(c). First, we recall notation from Section 4.6.3. Let $spec_\nabla$ be the *widening step thresholds* for the current step of the application of the widening. The widening $\nabla : \mathbb{S}_0 \times \mathbb{S}_0 \to \mathbb{S}_0$ is such that $\mathcal{S}_0 \nabla \mathcal{S}'_0 = \mathcal{S}_0 \cup_\blacklozenge \mathcal{S}'_0$ for $\mathcal{S}_0, \mathcal{S}'_0 \in \mathbb{S}_0$. We extend the operator to $\mathbb{S}_n$ for any $n > 0$ by an inductive construction. Let $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_n$. We compute:

$$\mathcal{X}_\nabla = \{-\infty\} \cup spec_L(\mathcal{S}) \cup spec_\nabla(v_n), \qquad \mathcal{X}'_\nabla = \mathcal{X}_\nabla \cup spec_L(\mathcal{S}').$$

Let $\mathcal{X}'_\nabla = \{p_1, p_2, \dots, p_m\}$ for some $m > 1$ such that $p_1 \prec p_2 \prec \dots, p_m$. Let elements $x_1, \dots, x_m$ be representatives of consequent segments $I_1, \dots, I_m$ in the segmentation $\mathcal{I}(\mathcal{X}'_\nabla)$, i.e.:

$$-\infty = p_1 \preceq (x_1, \oplus) \prec p_2 \preceq \dots \prec p_m \preceq (x_m, \oplus).$$

We define ∇-extension, $\nabla : \mathbb{S}_{n+1} \times \mathbb{S}_{n+1} \rightharpoonup \mathbb{S}_{n+1}$ as $\mathcal{S} \nabla \mathcal{S}' = \mathcal{R}$ such that $\mathcal{R} = \mathcal{S}'$ if $\mathcal{S} = \epsilon$. Otherwise we define:

$$\mathcal{R}' = \big((p_1, v_1), \dots, (p_m, v_m)\big)$$

such that for any $i \in \{1, \dots, m\}$:

$$v_i = \begin{cases} \mathcal{S}[x_i] \nabla_n \mathcal{S}'[x_i] & \text{if } \mathcal{S}[x_i] \neq \mathcal{S}'[x_i] & (4.37a) \\ \mathcal{S}[x_i] & \text{if } \mathcal{S}[x_i] = \mathcal{S}'[x_i] \text{ and } I_i \in \mathcal{I}(\mathcal{X}_\nabla) & (4.37b) \\ \mathcal{S}[x_i] \nabla_n \mathcal{S}'[x_k] & \begin{array}{l}\text{if } \mathcal{S}[x_i] = \mathcal{S}'[x_i], I_i \notin \mathcal{I}(\mathcal{X}_\nabla), \\ \quad k \text{ exists and } p_k \notin spec_L(\mathcal{S})\end{array} & (4.37c) \\ \mathcal{S}[x_i] \nabla_n \mathcal{S}'[x_{i-1}] & \text{otherwise,} & (4.37d) \end{cases}$$

where $k = \min\{j \mid i < j \leq m \text{ and } p_j \in spec_L(\mathcal{S}) \cup spec_L(\mathcal{S}')\}$. The element x_k is the first element in $spec_L(\mathcal{S}) \cup spec_L(\mathcal{S}')$ that is greater than x_i. The final result $\mathcal{R}$ of the widening is the normalised version of $\mathcal{R}'$.

The proposed refinement strategy extends eq. (4.32c) into two cases eq. (4.37c) and eq. (4.37d). As the refined value of the widening we take a value of a neighbour segment in the segmentation by $\{-\infty\} \cup spec_L(\mathcal{S}) \cup spec_L(\mathcal{S}')$ that belongs to the same segment in the segmentation by $\{-\infty\} \cup spec_L(\mathcal{S})$.

Example application of the proposed refinement strategy

In Fig. 4.9 we illustrate the proposed strategy. We have:

$$spec_L(\mathcal{S}) = \{p_2, p_7\}, \qquad spec_L(\mathcal{S}') = \{p_2, p_3, p_5, p_6\}, \qquad spec_\nabla(v) = \{p_4\}.$$

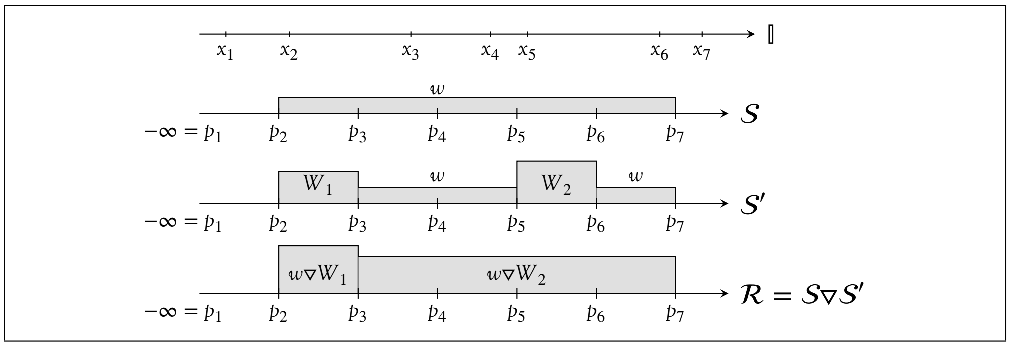


Figure 4.9: Example application of the proposed refinement strategy. Heights of bars correlate to values.

We compute:

$$\mathcal{X}_\nabla = \{-\infty, p_2, p_4, p_7\}, \qquad \mathcal{X}'_\nabla = \{-\infty, p_2, p_3, p_4, p_5, p_6, p_7\}.$$

Next, we compute $\mathcal{R}' = \big((p_1, v_1), \ldots, (p_6, v_6)\big)$. Values v_1, v_2, v_5, v_7 are computed with generic construction presented in Section 4.6.3, therefore we focus only on application of cases eq. (4.37c) and eq. (4.37d). The segmentations used in these cases are as follows:

- for $\{-\infty\} \cup spec_L(\mathcal{S})$: $\mathcal{I} = [-\infty, p_2), [p_2, p_7), [p_7, \infty)$,
- for $\{-\infty\} \cup spec_L(\mathcal{S}) \cup spec_L(\mathcal{S}')$: $\mathcal{I}' = [-\infty, p_2), [p_2, p_3), [p_3, p_5), [p_5, p_6)$, $[p_7, \infty)$.

Therefore, the values v_3, v_4, and v_6 are computed as follows:

- Value v_3: since $\mathcal{S}[x_3] = \mathcal{S}'[x_3]$ and $p_3 \notin \mathcal{X}_\triangledown$ we compute index $k = 5$. Since $p_5 \notin spec_L(\mathcal{S})$ we use case eq. (4.37c) to obtain $v_3 = \mathcal{S}[x_3]\triangledown_n\mathcal{S}'[x_5] = w\triangledown_n W_2$. The element x_3 belongs to the segment $[p_3, p_5)$ in the segmentation $\mathcal{I}'$. In $\mathcal{I}$ the segment is included in the segment $[p_2, p_7)$. In $\mathcal{I}'$ there is a segment that is on the right of $[p_3, p_5)$ and is also included in the same segment $[p_2, p_7)$. As the refined value we take value of the right neighbour — segment $[p_5, p_6)$.

- Value v_4: since $\mathcal{S}[x_4] = \mathcal{S}'[x_4]$ and $p_5 \notin \mathcal{X}_\triangledown$ we compute index $k = 5$. Since $p_5 \notin spec_L(\mathcal{S})$ we use case eq. (4.37c) to obtain $v_4 = \mathcal{S}[x_4]\triangledown_n\mathcal{S}'[x_5] = w\triangledown_n W_2$. Since x_4 is in the same segment as x_3 in the segmentation by $\{-\infty\} \cup spec_L(\mathcal{S}) \cup spec_L(\mathcal{S}')$, as the refinement we use the same value as in the previous case. Since x_4 belongs to the same segment as x_3 in the segmentation $\mathcal{I}'$, the situation is analogical as in the previous case.

- Value v_6: since $\mathcal{S}[x_6] = \mathcal{S}'[x_6]$ and $p_6 \notin \mathcal{X}_\triangledown$ we compute index $k = 7$. Since $p_7 \in spec_L(\mathcal{S})$ we use case eq. (4.37d) to obtain $v_6 = \mathcal{S}[x_6]\triangledown_n\mathcal{S}'[x_5] = w\triangledown_n W_2$. The element x_6 belongs to the segment $[p_6, p_7)$ in the segmentation $\mathcal{I}'$. In $\mathcal{I}$ the segment is included in $[p_2, p_7)$, but it is the rightmost fragment of the segment $[p_2, p_7)$. Therefore, there is no right side neighbour. But there is a left hand side neighbour segment $[p_5, p_6)$ in $\mathcal{I}'$ that is included in $[p_2, p_7)$. As a refined value we take the value of this segment.

Lemma 4.6.12 states, that the proposed refinement strategy is an implementation of the widening operator proposed in Section 4.6.3.

Lemma 4.6.12. *The last two steps eq. (4.37c) and eq. (4.37d) in the definition of the current operator fulfil the condition eq. (4.32c) from the definition of the generic widening operator, i.e. the refinement of the second argument in both cases is strictly greater than the value of the first argument.*

Proof. At the beginning we recall that we use the definition of the widening operator, where the second argument is greater or equal to the first one. Therefore, it holds that $\mathcal{S} \subseteq_\blacklozenge \mathcal{S}'$.

First, we analyse the case eq. (4.37c). We prove that $\mathcal{S}[x_i] \subset_\blacklozenge \mathcal{S}'[x_k]$. The situation is illustrated in Fig. 4.10a. Note that it holds that $\mathcal{S}[x_i] = \mathcal{S}'[x_i]$. The element p_i does not necessarily have to be either in $spec_L(\mathcal{S})$ or in $spec_L(\mathcal{S}')$ (hence, the question mark "?"). The element p_k is the smallest element from the set $spec_L(\mathcal{S}) \cup spec_L(\mathcal{S}')$ that is greater than p_i, and furthermore, it holds that $p_k \in spec_L(\mathcal{S}') \setminus spec_L(\mathcal{S})$. Since $p_k \notin spec_L(\mathcal{S})$ and there is no special point in $spec_L(\mathcal{S})$ that is between p_i and p_k, it holds that $\mathcal{S}[x_i] = \mathcal{S}[x_k]$. The element $p_k \in spec_L(\mathcal{S}')$ in the

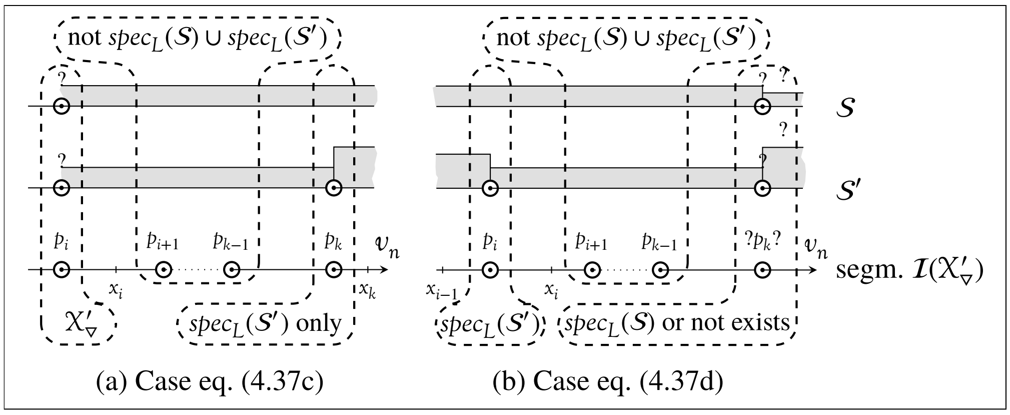


Figure 4.10: Illustration of the last two cases in eq. (4.37).

sequence $\mathcal{S}'$ is the beginning of a segment that has a value different than the previous segment, thus $\mathcal{S}'[x_k] \neq \mathcal{S}[x_k]$. Since $\mathcal{S} \subseteq_\blacklozenge \mathcal{S}'$, we obtain the desired property, i.e. $\mathcal{S}[x_i] \subset_\blacklozenge \mathcal{S}'[x_k]$.

Now we analyse the case eq. (4.37d). First, we prove that we may write $\mathcal{S}'[x_{i-1}]$, i.e. that $i > 1$. Assume the contrary, that $i = 1$. Then $p_1 = -\infty$. If k exists, by the definition of the current case it holds that $p_k \in spec_L(\mathcal{S})$, therefore $I_1 \in \mathcal{I}(\mathcal{X}_\nabla)$ and the eq. (4.37d) cannot be used to compute v_i — we obtain a contradiction. Otherwise, if k does not exist then there are no other special points in $\mathcal{S}$ or $\mathcal{S}'$, hence also $I_1 \in \mathcal{I}(\mathcal{X}_\nabla)$ and we obtain contradiction. Now we prove that $\mathcal{S}[x_i] \subset_\blacklozenge \mathcal{S}'[x_{i-1}]$. The situation is illustrated in Fig. 4.10b. Note that the element p_k, if exists, is the beginning of a new segment in both $\mathcal{S}$ and $\mathcal{S}'$. The element p_i is the end of some segment in $\mathcal{S}'$ and it does not appear in $spec_L(\mathcal{S})$ (otherwise, the case eq. (4.37b) would be used to compute v_i). Therefore $\mathcal{S}'[x_{i-1}] \neq \mathcal{S}[x_{i-1}]$. Since $\mathcal{S} \subseteq_\blacklozenge \mathcal{S}'$ we obtain the desired property, i.e. $\mathcal{S}[x_i] \subset_\blacklozenge \mathcal{S}'[x_{i-1}]$. □

By Lemma 4.6.12 and Theorem 4.6.10 the current operator is a proper widening operator.

Single-step precision analysis

The single-step precision of the widening operator proposed in eq. (4.37) depends on the choice of the *widening step thresholds* function. A larger sets of threshold points may increase the precision. The fact is stated formally by Theorem 4.6.13.

Theorem 4.6.13 (Single step precision of the widening). *Let $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_n$ for some $n > 0$ be such that $\mathcal{S} \subseteq_\blacklozenge \mathcal{S}'$. Let there be two widening operators ∇ and $\nabla^\star$ defined by eq.* (4.37) *that use different* widening step thresholds *$spec_\nabla$ and $spec_{\nabla^\star}$*

respectively to compute result of widening for arguments $\mathcal{S}$ and $\mathcal{S}'$. Let $spec_\triangledown$ and $spec_{\triangledown^\star}$ be such that:

$$\forall_{v\in Var}\, spec_\triangledown(v) \subseteq spec_{\triangledown^\star}(v). \tag{4.38}$$

Then it holds that:

$$\mathcal{S}\triangledown^\star\mathcal{S}' \subseteq_\blacklozenge \mathcal{S}\triangledown\mathcal{S}'. \tag{4.39}$$

Proof. We assume the notation as in the formulation of the theorem. The proof is by induction on n. For $n = 0$ both widening operators are defined exactly the same way since the *widening step threshold* is not used at all. Therefore (4.39) holds. Now assume that the theorem holds for $n-1$, where $n \geq 1$. We prove that it also holds for n.

First, we introduce auxiliary denotations. Let $\mathcal{X}_\triangledown$, $\mathcal{X}'_\triangledown$ be sets from the definition of the widening $\triangledown$:

$$\mathcal{X}_\triangledown = \{-\infty\} \cup spec_L(\mathcal{S}) \cup spec_\triangledown(v_n), \qquad \mathcal{X}'_\triangledown = \mathcal{X}_\triangledown \cup spec_L(\mathcal{S}')$$

and $\mathcal{R}'$ be as in the definition of $\triangledown$, that is:

$$\mathcal{R}' = \big((p_1, v_1), \ldots, (p_l, v_l)\big)$$

for some $l > 0$, and $x_i \in \mathbb{I}$ for $i \in \{1, \ldots, l\}$ be representatives of consequent segments $I_1, \ldots, I_l$ in the segmentation $\mathcal{I}(\mathcal{X}'_\triangledown)$ (see Definition 4.6.5).

Analogously, let $\mathcal{Y}_{\triangledown^\star}$, $\mathcal{Y}'_{\triangledown^\star}$ be sets from the definition of the widening $\triangledown^\star$:

$$\mathcal{Y}_{\triangledown^\star} = \{-\infty\} \cup spec_L(\mathcal{S}) \cup spec_{\triangledown^\star}(v_n), \qquad \mathcal{Y}'_{\triangledown^\star} = \mathcal{Y}_{\triangledown^\star} \cup spec_L(\mathcal{S}')$$

and $\mathcal{R}^\star$ be as in the definition of $\triangledown^\star$, that is:

$$\mathcal{R}^* = \big((p_1^\star, v_1^\star), \ldots, (p_m^\star, v_m^\star)\big)$$

for some $m > 0$, and $x_j^\star \in \mathbb{I}$ for $j \in \{1, \ldots, m\}$ be representatives of consequent segments $I_1^\star, \ldots, I_m^\star$ in the segmentation $\mathcal{I}(\mathcal{Y}'_{\triangledown^\star})$.

Let $j \in \{1, \ldots, m\}$. By Remark 4.6.6 it holds that $I_j^\star \subseteq I_i$ for some $i \in \{1, \ldots, l\}$. Intuitively by eq. (4.38), the second widening operator $\triangledown^\star$ may extend the segmentation from $\triangledown$ by adding some new special points for the current variable. Therefore, the segmentation for $\triangledown^\star$ is more precise than for $\triangledown$, i.e. with these additional special points some segments might be split into smaller subsegments. The segment $I_j^\star$ is a subsegment of some segment I_i (which might have been split to obtain $I_j^\star$). The element $x_j^\star$ is a representative of both segment $I_j^\star$ and I_i, therefore it holds that:

$$\mathcal{S}[x_j^\star] = \mathcal{S}[x_i], \qquad \mathcal{S}'[x_j^\star] = \mathcal{S}'[x_i], \qquad (\mathcal{S}\triangledown\mathcal{S}')[x_j^\star] = (\mathcal{S}\triangledown\mathcal{S}')[x_i]. \tag{4.40}$$

The last equality results directly from $I^\star_j \subseteq I_i$ and the construction of $\triangledown$ — one value is computed for the whole segment I_i.

The rest of the proof is a case analysis of the cases in the definition of the widening operator in eq. (4.37) for the widening $\triangledown^\star$. In every case we show that:

$$v^\star_j = (\mathcal{S}\triangledown^\star_n\mathcal{S}')[x^\star_j] \subseteq_\blacklozenge (\mathcal{S}\triangledown_n\mathcal{S}')[x_i] = v_i, \tag{4.41}$$

which satisfies eq. (4.39).

Case eq. (4.37a) In this case values for the segment $I^\star_j$ are different in sequences $\mathcal{S}$ and $\mathcal{S}'$, i.e. $\mathcal{S}[x^\star_j] \neq \mathcal{S}'[x^\star_j]$. By eq. (4.40) also $\mathcal{S}[x_i] \neq \mathcal{S}'[x_i]$. By the induction hypothesis for arguments $\mathcal{S}[x^\star_j]$ and $\mathcal{S}'[x^\star_j]$, and eq. (4.40) we obtain that:

$$v^\star_j = \mathcal{S}[x^\star_j]\triangledown^\star_n\mathcal{S}'[x^\star_j] \subseteq_\blacklozenge \mathcal{S}[x^\star_j]\triangledown_n\mathcal{S}'[x^\star_j] = \mathcal{S}[x_i]\triangledown_n\mathcal{S}'[x_i] = (\mathcal{S}\triangledown_{n+1}\mathcal{S}')[x_i] = v_i.$$

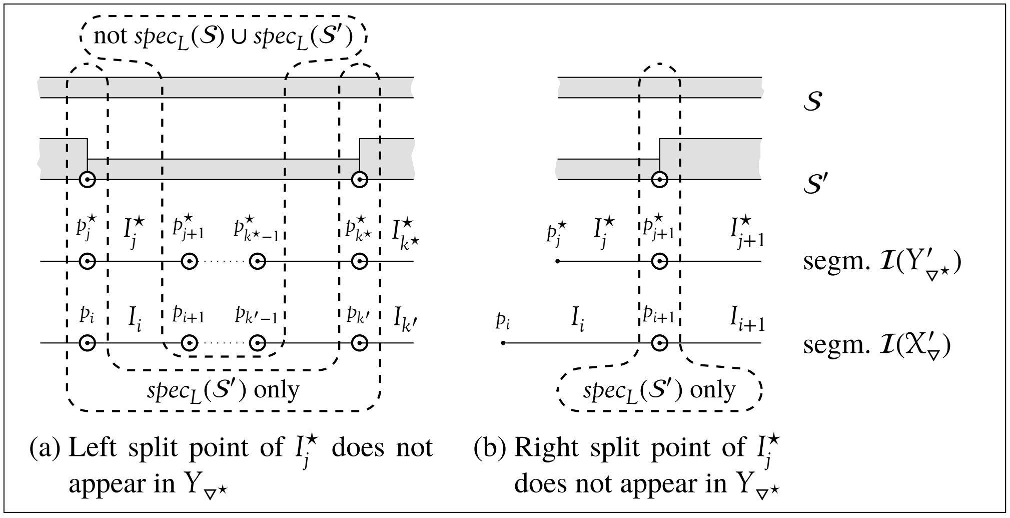
Case eq. (4.37b) This time values for the segment $I^\star_j$ are equal in $\mathcal{S}$ and $\mathcal{S}'$, and $I^\star_j \in \mathcal{I}(Y_{\triangledown^\star})$. Then $v^\star_j = \mathcal{S}[x^\star_j]$. We analyse the subcases of eq. (4.37), in which the value v_i is computed:

- Since $\mathcal{S}[x_i] \neq \mathcal{S}'[x_i]$, the value is not computed with the case eq. (4.37a).
- If v_i is computed with the same case as $v^\star_j$, i.e. eq. (4.37b), then $v_i = \mathcal{S}[x_i]$ and by eq. (4.40) we obtain eq. (4.41).
- If the value v_i is computed with either case eq. (4.37c) or eq. (4.37d) we apply Lemma 4.6.12 to obtain that $v_i = \mathcal{S}[x_i]\triangledown_n W$ for some $W \in \mathbb{S}_n$ such that $\mathcal{S}[x_i] \subset_\blacklozenge W$. Therefore, by the *over-approximation* property of the widening operator $\triangledown$ it holds that:

  $$W \subseteq_\blacklozenge \mathcal{S}[x_i]\triangledown_n W = v_i$$

  and by eq. (4.40) we obtain eq. (4.41).

Case eq. (4.37c) It holds that $\mathcal{S}[x^\star_j] = \mathcal{S}'[x^\star_j]$ and the segment $I^\star_j$ is not in the segmentation $\mathcal{I}(Y_{\triangledown^\star})$, i.e. $I^\star_j \notin \mathcal{I}(Y_{\triangledown^\star})$. What is more, the first element in the set $spec_L(\mathcal{S}) \cup spec_L(\mathcal{S}')$ that is greater than $p^\star_j$ exists: it is $p^\star_{k^\star}$, where $j < k^\star \leq m$ and $p^\star_{k^\star} \in spec_L(\mathcal{S}') \setminus spec_L(\mathcal{S})$. Since $I^\star_j \notin \mathcal{I}(Y_{\triangledown^\star})$ at least one of the split points: $p^\star_j$ or $p^\star_{j+1}$ does not belong to $Y_{\triangledown^\star}$. Both situations are illustrated in Fig. 4.11. We prove that in both situations the current case eq. (4.37c) is used in the computation of v_i, and that the refined value for the second argument is the same for both widening operators — it is a value of a neighbour segment to the right in $\mathcal{S}'$, which is included in the current segment in $\mathcal{S}$. The cases are as follows:



not $spec_L(S) \cup spec_L(S')$
S
S'
segm. $\mathcal{I}(Y'_{\nabla^\star})$
segm. $\mathcal{I}(X'_\nabla)$
$p_j^\star$ $I_j^\star$ $p_{j+1}^\star$ $p_{k^\star-1}^\star$ $p_{k^\star}^\star$ $I_{k^\star}^\star$
p_i I_i p_{i+1} $p_{k'-1}$ $p_{k'}$ $I_{k'}$
$spec_L(S')$ only
$p_j^\star$ $I_j^\star$ $p_{j+1}^\star$ $I_{j+1}^\star$
p_i I_i p_{i+1} I_{i+1}
$spec_L(S')$ only
(a) Left split point of $I_j^\star$ does not appear in $Y_{\nabla^\star}$
(b) Right split point of $I_j^\star$ does not appear in $Y_{\nabla^\star}$

Figure 4.11: Possible situations when $v_j^\star$ is computed with eq. (4.37c). First two rows are graphical representations of arguments S and S'. Last two rows present fragments of two segmentations: $\mathcal{I}(X'_\nabla)$ and $\mathcal{I}(Y'_{\nabla^\star})$.

- The left split point of the segment $I_j^\star$ does not belong to $Y_{\nabla^\star}$, i.e.:

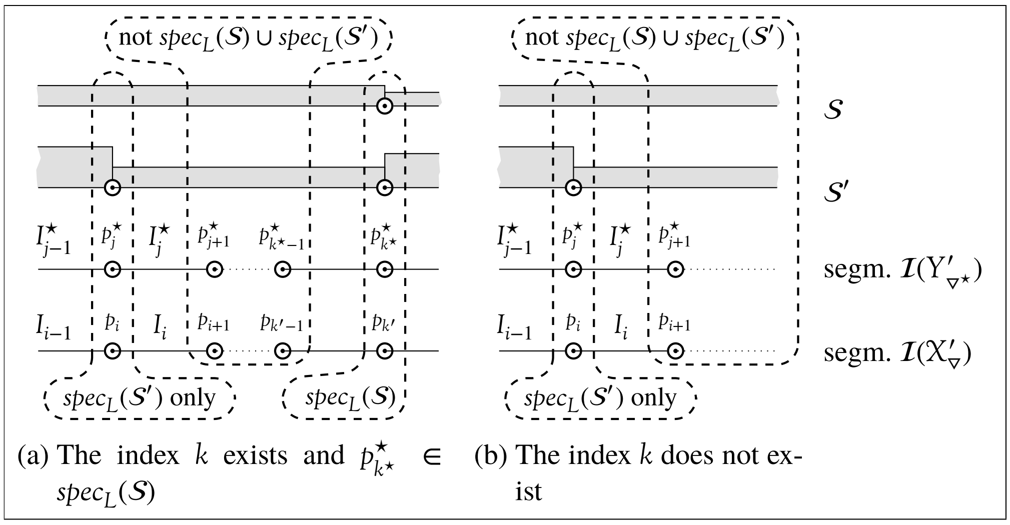
$$p_j^\star \in spec_L(S') \setminus Y_{\nabla^\star}.$$

  The situation is illustrated in Fig. 4.11a. The element $p_j^\star$ is also used as a split point in the segmentation $\mathcal{I}(X'_\nabla)$, which is less precise than $\mathcal{I}(Y'_{\nabla^\star})$. Therefore, it holds that $p_j^\star = p_i$. The element $p_{k^\star}^\star$ is the first element from the set $spec_L(S) \cup spec_L(S')$ that is to the right of the segment $I_j^\star$. The element is also to the right of the segment I_i, therefore index k exists also when v_i is computed. Let k' denote the index. It holds that $p_{k^\star}^\star = p_{k'}$. The same case eq. (4.37c) is used to compute the value v_i. Since $I_{k^\star}^\star \subseteq I_{k'}$ it holds that $S'[x_{k^\star}^\star] = S'[x_{k'}]$, i.e. the refined value for the right argument is the same for both widenings. Finally, by eq. (4.40) and the induction hypothesis we obtain eq. (4.41).

- Otherwise, the right split point of $I_j^\star$ does not belong to $Y_{\nabla^\star}$, i.e.:

$$j < m \text{ and } p_{j+1}^\star \in spec_L(S') \setminus Y_{\nabla^\star}.$$

  Then $k^\star$, the index k used in the computation of $v_j^\star$, is $j+1$. The situation is illustrated in Fig. 4.11b. Since $p_{j+1}^\star \in spec_L(S')$, by the construction of X'_∇, it holds that $p_{j+1}^\star \in X'_\nabla$. Also, it holds that $p_{i+1} = p_{j+1}^\star$, i.e. the right split



not $spec_L(\mathcal{S}) \cup spec_L(\mathcal{S}')$
not $spec_L(\mathcal{S}) \cup spec_L(\mathcal{S}')$
$\mathcal{S}$
$\mathcal{S}'$
$I^\star_{j-1}$
$p^\star_j$
$I^\star_j$
$p^\star_{j+1}$
$p^\star_{k^\star-1}$
$p^\star_{k^\star}$
segm. $\mathcal{I}(Y'_{\nabla^\star})$
I_{i-1}
p_i
I_i
p_{i+1}
$p_{k'-1}$
$p_{k'}$
segm. $\mathcal{I}(\mathcal{X}'_\nabla)$
$spec_L(\mathcal{S}')$ only
$spec_L(\mathcal{S})$
$spec_L(\mathcal{S}')$ only
(a) The index k exists and $p^\star_{k^\star} \in spec_L(\mathcal{S})$
(b) The index k does not exist

Figure 4.12: Possible situations when $v^\star_j$ is computed with eq. (4.37d). First two rows are graphical representations of arguments $\mathcal{S}$ and $\mathcal{S}'$. Last two rows present fragments of two segmentations: $\mathcal{I}(\mathcal{X}'_\nabla)$ and $\mathcal{I}(Y'_{\nabla^\star})$.

point for segments $I^\star_j$ and I_i is the same element. It is a consequence of the segment inclusion $I^\star_j \subseteq I_i$ and the fact that the right split point of $I^\star_j$ appears as a split point in $\mathcal{I}(\mathcal{X}'_\nabla)$. The index $i+1$ matches conditions of eq. (4.37c) as the index k in the computation of v_i, therefore the same case is used to compute v_i.

The value $\mathcal{S}'[x^\star_{j+1}]$ is the refined value of the second argument in the computation of $v^\star_j$ and $\mathcal{S}'[x_{i+1}]$ is the refined value of the second argument in the computation of v_i. Since it holds that $I^\star_{j+1} \subseteq I_{i+1}$ (the left split point is the same element and $\mathcal{X}'_\nabla \subseteq Y'_{\nabla^\star}$), we obtain $\mathcal{S}'[x^\star_{j+1}] = \mathcal{S}'[x_{i+1}]$, thus refined values are equal. Finally, by eq. (4.40) and the induction hypothesis we obtain eq. (4.41).

Case eq. (4.37d) It holds that $\mathcal{S}[x^\star_j] = \mathcal{S}'[x^\star_j]$ and the segment $I^\star_j$ is not in the segmentation $\mathcal{I}(Y_{\nabla^\star})$, i.e. $I^\star_j \notin \mathcal{I}(Y_{\nabla^\star})$. We have two possibilities depending on the existence of the index $k^\star = \min\{q \mid j < q \leq m \text{ and } p_q \in spec_L(\mathcal{S}) \cup spec_L(\mathcal{S}')\}$ (both situations are illustrated in Fig. 4.12):

- The index exists. This situation is illustrated in Fig. 4.12a. Then, by the definition of the current case it holds that $p^\star_{k^\star} \in spec_L(\mathcal{S})$. It means that the value of the sequence $\mathcal{S}$ changes at the split point $p^\star_{k^\star}$, i.e. $\mathcal{S}[x^\star_{k^\star-1}] \neq \mathcal{S}[x^\star_{k^\star}]$.

By the definition of the current case it holds that $I_j^\star \notin \mathcal{I}(\mathcal{Y}_{\triangledown^\star})$, therefore the left split point does not belong to $\mathcal{Y}_{\triangledown^\star}$, i.e. $p_j^\star \in spec_L(\mathcal{S}') \setminus \mathcal{Y}_{\triangledown^\star}$. But then, also $p_j^\star \in spec_L(\mathcal{S}') \setminus \mathcal{X}_{\triangledown}$, therefore as $I_j^\star \subseteq I_i$ we have $p_i = p_j^\star$.

To calculate v_i, we observe that the following conditions hold:

- $\mathcal{S}[x_i] = \mathcal{S}'[x_i]$, as $\mathcal{S}[x_i] = \mathcal{S}[x_j^\star] = \mathcal{S}'[x_j^\star] = \mathcal{S}'[x_i]$.
- $I_i \notin \mathcal{I}(\mathcal{X}_\triangledown)$, as $p_i \in spec_L(\mathcal{S}') \setminus \mathcal{X}_\triangledown$.
- The index $k' = \min\{q \mid i < q \leq l \text{ and } p_q \in spec_L(\mathcal{S}) \cup spec_L(\mathcal{S}')\}$ is defined since $i < k^\star$ and $p_{k^\star}^\star \in spec_L(\mathcal{S})$. Moreover $p_{k'} = p_{k^\star}^\star$.

As a result, we must use the case eq. (4.37d) to compute $(\mathcal{S} \triangledown \mathcal{S}')[x_i]$. The result is $\mathcal{S}[x_i] \triangledown_n \mathcal{S}'[x_{i-1}]$. Since $p_i = p_j^\star \notin \mathcal{Y}_{\triangledown^\star}$ it holds that $p_j^\star \neq -\infty$, and this implies $j > 1$ and $i > 1$. As $\mathcal{Y}_{\triangledown^\star}$ is a refinement of $\mathcal{X}_\triangledown$ and the right-hand bounds of $I_{j-1}^\star$ and I_{i-1} are equal, we obtain $I_{j-1}^\star \subseteq I_{i-1}$. This implies $\mathcal{S}'[x_{j-1}^\star] = \mathcal{S}'[x_{i-1}]$. We can now compute:

$$v_i = (\mathcal{S} \triangledown_n \mathcal{S}')[x_i] = \mathcal{S}[x_i] \triangledown_n \mathcal{S}'[x_{i-1}] = \mathcal{S}[x_j^\star] \triangledown_n \mathcal{S}'[x_{j-1}^\star].$$

By the induction hypothesis to obtain $\mathcal{S}[x_j^\star] \triangledown_n^\star \mathcal{S}'[x_{j-1}^\star] \subseteq_\blacklozenge \mathcal{S}[x_j^\star] \triangledown_n \mathcal{S}'[x_{j-1}^\star]$. Finally, $\mathcal{S}[x_j^\star] \triangledown_n^\star \mathcal{S}'[x_{j-1}^\star] = (\mathcal{S} \triangledown_n^\star \mathcal{S}')[x_j^\star] = v_j^\star$, thus $v_j^\star \subseteq_\blacklozenge v_i$.

- Otherwise, the index does not exist. The situation is illustrated in Fig. 4.12b. By the definition of the current case it holds that $I_j^\star \notin \mathcal{I}(\mathcal{Y}_{\triangledown^\star})$. As a result $p_j^\star \in spec_L(\mathcal{S}') \setminus \mathcal{Y}_{\triangledown^\star}$. But then, also $p_j^\star \in spec_L(\mathcal{S}') \setminus \mathcal{X}_\triangledown$. Since $I_j^\star \subseteq I_i$, it holds that $p_i = p_j^\star$. Further reasoning is analogical to the previous case. □

Theorem 4.6.13 only states that when we take a greater *widening step thresholds* (pointwise ordering, for every variable) then the result of application of the widening step is not worse. But it might happen that the result is strictly greater. A simple example for the two-dimensional case is presented in Fig. 4.13. The variable x is the variable of the first dimension and y is the variable of the second dimension. We focus on segments in the second dimension (of variable y). The segment $[-\infty, 0)$ is handled exactly the same way by both widenings. For the segment $[0, 1)$ both widening operators apply one-dimensional version. The first widening $\triangledown$ widens straight to ∞ while the proposed widening $\triangledown'$ stops at the threshold $2 \in spec_\triangledown(x)$. Note that the second widening operator $\triangledown'$ is able to share information about special points for the variable x between different blocks for the variable y. That is why in the block $y \in [0, 1)$ we use the special point $x = 2$ as a threshold and obtain a more precise result.

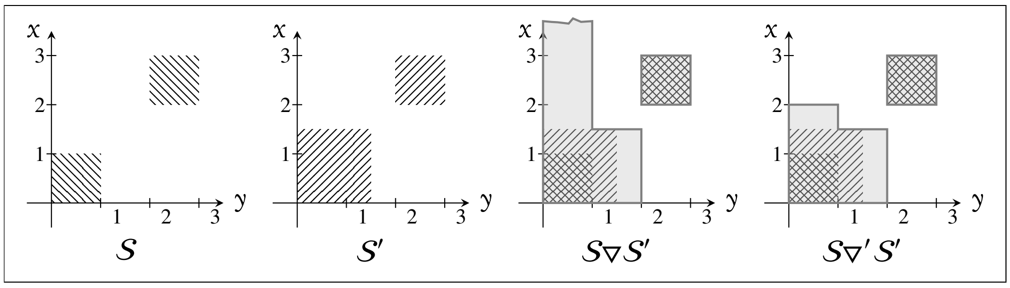

x
y
1
2
3
$\mathcal{S}$
$\mathcal{S}'$
$\mathcal{S}\nabla\mathcal{S}'$
$\mathcal{S}\nabla'\mathcal{S}'$

Figure 4.13: Comparison of a single-step precision of widening operators for different *widening step thresholds* functions. The widening operator ∇ uses $spec_\nabla = \lambda v \,.\, \emptyset$ and ∇' uses $spec'_\nabla = spec_G(\mathcal{S})$, therefore $spec'_\nabla(x) = \{0, 1, 2, 3\}$.

Comparison with the widening operator based on LDDs

The proposed widening operator is similar to ∇_{LDD} — the widening operator based on LDDs. The construction of the widening ∇_{LDD} expressed in terms of the presented sweeping line representation of domain elements is an instance of the presented generic widening operator with proposed refinement strategy for empty *widening sequence thresholds*, i.e. in every step *widening step thresholds* function is $spec_{\nabla,LDD}(v) = \emptyset$. As a consequence, single step application of the introduced widening is more precise than ∇_{LDD}. The fact is stated by Theorem 4.6.14.

Theorem 4.6.14. *For any $n \geq 0$ and $\mathcal{S}, \mathcal{S}' \in \mathbb{S}_n$ it holds that:*

$$\mathcal{S}\nabla\mathcal{S}' \subseteq_\blacklozenge \mathcal{S}\nabla_{LDD}\mathcal{S}'. \tag{4.42}$$

Proof. For any *widening step thresholds* function $spec_\nabla$ it holds that:

$$\forall v \in \mathit{Var} : \ \emptyset = spec_{\nabla,LDD}(v) \subseteq spec_\nabla(v),$$

and therefore, by Theorem 4.6.13 we obtain eq. (4.42). □

For the comparison of ∇ and ∇_{LDD} consider an example from Fig. 4.13. In the example, we compute as *widening step thresholds global special points* (defined in eq. (4.36)).

Dependence on the variable ordering

One drawback of the proposed widening operator is that the result depends on the ordering of variables. Lemma 4.6.15 states what accuracy of the widening operator can be expected. An operator obtained by intersection of results obtained for all possible variable orderings does not create a proper widening operator. By introduction of special points $spec_\nabla$ to the widening sequence we try to get closer to the result of the intersection and *depend less* on the variable ordering. The result of the proposed widening in the example from Fig. 4.13 is pretty close to the intersection of widenings based on LDDs for both variable orderings.

Lemma 4.6.15. *For empty* widening sequence thresholds, $\mathbb{I} \in \{\mathbb{Q}, \mathbb{R}\}$ *and* $n \geq 2$ *the operator defined as the intersection of results for all possible variable orderings is not a widening operator.*

Proof. Let S_n be the set of permutations of $\{1, \dots, n\}$, i.e. $S_n = \{\pi : \{1, \dots, n\} \to \{1, \dots, n\} \mid \pi \text{ is bijective}\}$. Let $\mathit{Var} = \{v_1, \dots, v_n\}$ be the set of variables. The operator $\heartsuit : \mathbb{S}_n \to \mathbb{S}_n$ for every $n \geq 0$ is defined as follows:

$$\mathcal{S} \heartsuit \mathcal{S}' \stackrel{\text{def}}{=} \bigcap_{\pi \in S_n} \mathcal{S} \triangledown_\pi \mathcal{S}',$$

where $\triangledown_\pi$ is the widening operator as defined in the current section that uses the following variable ordering: $v_{\pi(1)}, v_{\pi(2)}, \dots, v_{\pi(n)}$ and $\bigcap$ is the generalised intersection operator (greatest lower bound). The operator can be easily implemented for $\mathbb{S}_n$ since meet is precise and the set of permutations S_n is finite.

For $n = 2$ we show an infinite, strictly increasing sequence $\mathcal{S}_0 \subsetneq \mathcal{S}_1 \subsetneq \dots$ such that the sequence:

$$\begin{aligned} \mathcal{R}_0 &= \mathcal{S}_0, \\ \mathcal{R}_{i+1} &= \mathcal{R}_i \heartsuit (\mathcal{R}_i \cup \mathcal{S}_{i+1}) \end{aligned} \tag{4.43}$$

is strictly increasing. Now we explain the construction of the sequence. It is illustrated in Fig. 4.14. Arguments and results of consequent steps are presented in rows. The first argument of the operator (which is also the result of previous step) is displayed in the column (i). The second argument is displayed in the column (ii). The result, which is the intersection of two possible widenings, is presented in the column (iii). The input sequence is constructed as follows (first 3 steps are described in detail):

1. The first element $\mathcal{S}_0$ is presented in step 0, col. (ii). It consists of two disjoint square boxes. Every square box has edge of length 1.

2. The second element $\mathcal{S}_1$ is presented in step 1, col. (ii). It is created from the previous element by extending both disjoint polygons from the previous step by adding a rectangle next to each of them. We place a rectangle of size (width $\times$ height) $\frac{1}{2} \times (1 + \frac{1}{2})$ on the right of the upper box and a rectangle of size $(1 + \frac{1}{2}) \times \frac{1}{2}$ above the lower box.

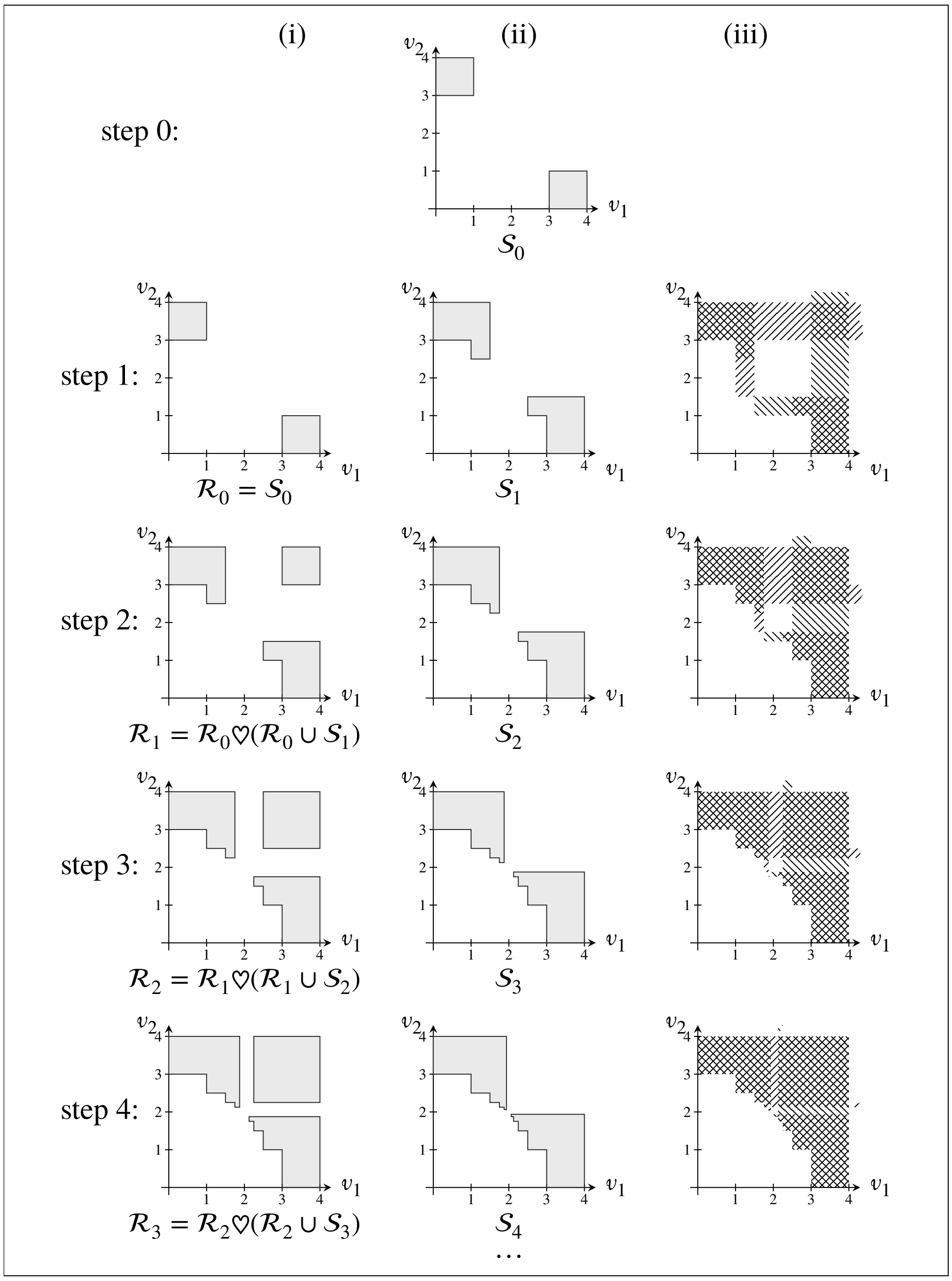

(i)
(ii)
(iii)
step 0:
step 1:
step 2:
step 3:
step 4:
v_1
v_2
$\mathcal{S}_0$
$\mathcal{R}_0 = \mathcal{S}_0$
$\mathcal{S}_1$
$\mathcal{R}_1 = \mathcal{R}_0 \heartsuit (\mathcal{R}_0 \cup \mathcal{S}_1)$
$\mathcal{S}_2$
$\mathcal{R}_2 = \mathcal{R}_1 \heartsuit (\mathcal{R}_1 \cup \mathcal{S}_2)$
$\mathcal{S}_3$
$\mathcal{R}_3 = \mathcal{R}_2 \heartsuit (\mathcal{R}_2 \cup \mathcal{S}_3)$
$\mathcal{S}_4$
...

Figure 4.14: Construction of the divergent sequence, columns describe (i) first argument, (ii) second argument (input sequence), and (iii) results of $\heartsuit$ (intersection of results of ∇ for two possible variable orderings v_1, v_2 and v_2, v_1)

3. The third element $\mathcal{S}_2$ is presented in step 2, col. (ii). First, similarly to the previous step, we extend both polygons from the previous step. Next, we add rectangle of size $\frac{1}{4} \times (1 + \frac{1}{2} + \frac{1}{4})$ on the right of the upper polygon and of size $(1 + \frac{1}{2} + \frac{1}{4}) \times \frac{1}{4}$ above the lower one.
4. …

A general rule for creating element $\mathcal{S}_k$ for $k > 0$ from the previous one is as follows:

- add a rectangle of size $\frac{1}{2^k} \times \sum_{i=0}^{k} \frac{1}{2^k}$ on the right of upper-right polygon,
- add a rectangle of size $\sum_{i=0}^{k} \frac{1}{2^k} \times \frac{1}{2^k}$ above the lower-right polygon.

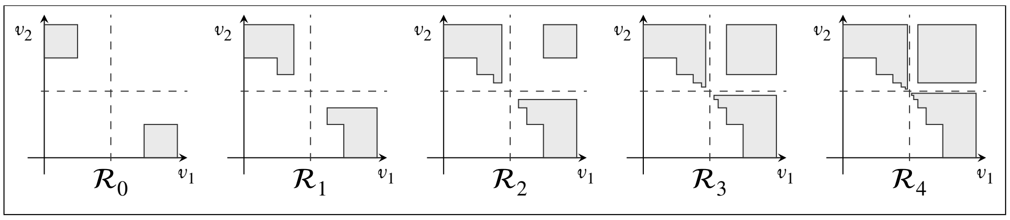


Figure 4.15: Divergent sequence $\mathcal{R}_0, \mathcal{R}_1, \mathcal{R}_2, \ldots$

The sequence $\mathcal{R}_0, \mathcal{R}_1, \ldots$ created with presented algorithm is strictly increasing. A longer fragment of the sequence is presented in Fig. 4.15. As we can see, all three polygons grow and converge to the center point (2,2). We can easily extend the construction to higher dimensions by adding any set of values there. Since variables are independent the sequence would still be strictly increasing. □

Lemma 4.6.15 gives us just a rough estimate of the limitations of the widening operator for the domain of *boxes*. A discussion about precision of the widening operator is a difficult matter since widening operator is not monotone. It may happen that some widening operator $\triangledown'$ at some step may come up with a worse result than $\triangledown$ at the step, but the overall result (the value at which we reach stabilisation) is more precise.

Let us consider the input sequence from Lemma 4.6.15. A widening operator that gives the best (the most precise) result for the input sequence does not exist. A number of possible widening results for the input sequence from Lemma 4.6.15 are presented in Fig. 4.16. In fact, this is a sequence of possible results such that every next element is strictly greater than the previous one. The sequence $\mathcal{R}^0_\triangledown, \mathcal{R}^1_\triangledown, \ldots$ converges to an element that is not a member of the domain of *boxes*, since it consists of infinitely many disjoint boxes.

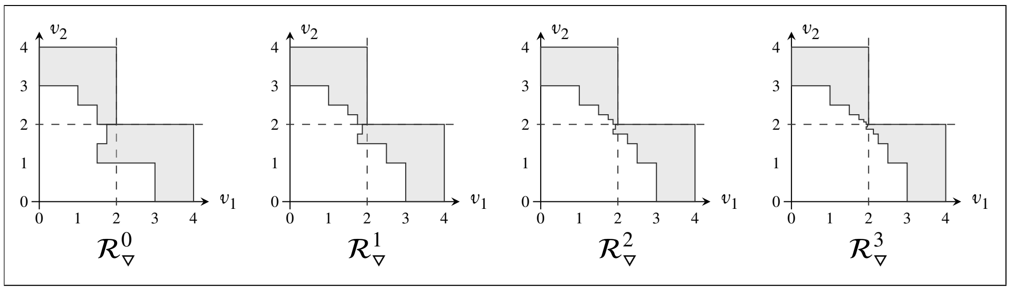


Figure 4.16: Strictly decreasing sequence of possible widening results for the input sequence from Lemma 4.6.15.

The results of widening presented in Fig. 4.16 can be obtained by applying the presented widening operator. What is required is a proper choice of *widening sequence thresholds*. In order to obtain the result $\mathcal{R}^k_\triangledown$ for $k \geq 0$ we choose constant *widening sequence thresholds*, i.e. exactly the same *widening step thresholds* function $spec^k_\triangledown$ is used for every step. These *widening step thresholds* functions are defined as follows:

$$\begin{array}{rcccc}
spec^0_\triangledown(v) & = & \{2\} & = & \{2\}, \\
spec^1_\triangledown(v) & = & spec^0_\triangledown(v) \cup spec_G(\mathcal{S}_1)(v) & = & \{0, 1, 1\frac{1}{2}, 2, 2\frac{1}{2}, 3, 4\}, \\
spec^2_\triangledown(v) & = & spec^1_\triangledown(v) \cup spec_G(\mathcal{S}_2)(v) & = & \{0, 1, 1\frac{1}{2}, 1\frac{3}{4}, 2, 2\frac{1}{4}, 2\frac{1}{2}, 3, 4\}, \\
spec^3_\triangledown(v) & = & spec^2_\triangledown(v) \cup spec_G(\mathcal{S}_3)(v) & = & \{0, 1, 1\frac{1}{2}, 1\frac{3}{4}, 1\frac{7}{8}, 2, 2\frac{1}{8}, 2\frac{1}{4}, 2\frac{1}{2}, 3, 4\}.
\end{array}$$

Note that we have added threshold point 2. This caused that the top-right square box that occurred in elements presented in Fig. 4.15 has disappeared.

4.7 Transfer Functions

In this section we present implementations of *assign* and *test* functions described in Section 3.2 (page 37) for the introduced representation of the domain of boxes. For both function, when the argument $\mathcal{S} \in \mathbb{S}_n$ for some $n > 0$ is ϵ, the result is ϵ. So, from now on we assume $\mathcal{S} \neq \epsilon$.

We recall that function *split* from Section 4.5 (page 88) splits an element $\mathcal{S} \in \mathbb{S}_n$ for some $n > 0$ to corresponding disjoint boxes. The *test* function for the domain of *boxes* can be defined using the $test_v$ function for the domain of *intervals* as follows: first we apply the *split* function to obtain disjoint boxes, next we compute result of $test_v$ for every box in the split, and finally we apply the join operator for these results. Such function *test* is a sound approximation of the concrete test of boolean expressions. This is stated formally by Theorem 4.7.1.

Theorem 4.7.1. *Let $\mathcal{S} \in \mathbb{S}_n$ for some $n > 0$ and $split_n(\mathcal{S}) = \{\mathcal{B}_1, \ldots, \mathcal{B}_k\}$ for some $k > 0$. Furthermore, let $bb \in Bool$ and $t \in BExp$ be a test expression. Then test function for the domain of* boxes *defined as follows:*

$$test(t, bb, \mathcal{S}) = \bigcup_{i=1}^{k} test_v(t, bb, \mathcal{B}_i),$$

is a sound approximation of the boolean test t, i.e. it fulfils the following property:

$$(s \in \gamma(\mathcal{S}) \wedge \mathcal{B}\,[\![t]\!]\; s = bb) \Longrightarrow s \in \gamma(test(b, bb, \mathcal{S})).$$

Proof. Proof left to the reader. □

Analogously, we proceed with the definition of the *assign* function for the domain of *boxes*:

Theorem 4.7.2. *Let $\mathcal{S} \in \mathbb{S}_n$ for some $n > 0$ and $split_n(\mathcal{S}) = \{\mathcal{B}_1, \ldots, \mathcal{B}_k\}$ for some $k > 0$. Furthermore, let $v_i \in Var$ and $e \in Exp$. Then assign function for the domain of* boxes *defined as follows:*

$$assign(v_i, e, \mathcal{S}) = \bigcup_{i=1}^{k} assign_v(v_i, e, \mathcal{B}_i),$$

is a sound approximation of the assignment operation, i.e. it fulfils the following property:

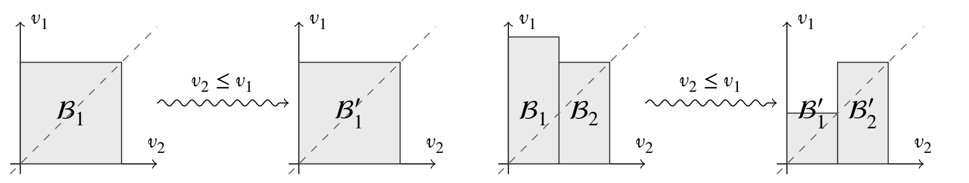
$$(s \in \gamma(\mathcal{S}) \wedge \mathcal{E}\,[\![e]\!]\; s = w) \Longrightarrow s[v_i \leftarrow w] \in \gamma(assign(v_i, e, \mathcal{S})).$$

Proof. Proof left to the reader. □

Note that the transfer function is not monotone. We analyse an example that is presented in Fig. 4.17. In Fig. 4.17(a) the transfer function does not change the input domain value. In Fig. 4.17(b) the input domain value is split into two disjoint boxes and the application of the transfer function modifies only one of them.

In some cases the generic definition is too costly in realization. There are many cases, where it may be done in a more efficient fashion, without the *split* function. We sketch below how a more optimal version can be defined. To do this we start with a definition of a series of helping functions $apply_{j,i} : (\mathbb{S}_i \to \mathbb{S}_i) \times \mathbb{S}_j \to \mathbb{S}_j$, where $i, j \in \{1, \ldots, n\}$ and $i \leq j$. Let $\mathcal{S}_j \in \mathbb{S}_j$ be such that $\mathcal{S}_j = \big((p_1, s_1), \ldots, (p_k, s_k)\big)$ for some $k \geq 0$, then:

$$apply_{j,i}(t, \mathcal{S}_j) = \begin{cases} t(\mathcal{S}_j) & \text{if } i = j \\ \mathcal{R} & \text{otherwise,} \end{cases}$$



(a) No value change

(b) Strictly greater argument gives a strictly smaller result

Figure 4.17: Example that shows the transfer function for *boxes* is not monotone.

where $\mathcal{R}$ is a normalised version of the sequence:

$$\mathcal{R}' = \big((p_1, apply_{j-1,i}(t, s_1)), \dots, (p_k, apply_{j-1,i}(t, s_k))\big)\,.$$

The application of the function $apply_{j,i}(t, \mathcal{S})$ for $\mathcal{S} \in \mathbb{S}_j$ returns a modified sequence $\mathcal{S}'$, where sequences for the variable v_i are replaced by application of the function t. We use the function *apply* in the implementation of both *assign* and *test* functions below.

4.7.1 Test

First, we focus on the *test* function. Let $\mathcal{S} \in \mathbb{S}_n$ for some $n > 0$ be an element of the domain of boxes. Recall first that it is enough to consider atomic boolean expressions only, i.e. ones of the form $e_1 \bowtie e_2$, where $\bowtie \in \{=, \neq, <, \leq\}$ and e_1, e_2 are either constants or variables.

When the test expression does not contain any variables the evaluation process is analogical to the corresponding one in the concrete domain:

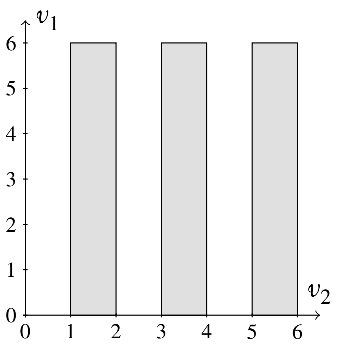
$$test(b, \mathsf{bb}, \mathcal{S}) = \begin{cases} \bot & \text{if } \mathcal{B}\,[\![\mathrm{b}]\!]\ \mathcal{S} = \mathit{ff} \\ \mathcal{S} & \text{otherwise,} \end{cases}$$

where $\mathsf{bb} \in \mathsf{Bool}$. When the test expression contains exactly one variable (exactly one of the atoms a_1, a_2 is a variable) the situation becomes a bit more complex. We use the *apply* function to make the operation more efficient. When $\mathbb{I} \in \{\mathbb{Q}, \mathbb{R}\}$ the implementations are as follows:

- $test(v_i = a, \mathsf{tt}, \mathcal{S}) = apply_{n,i}(\lambda \mathcal{S}_i \,.\, \mathcal{S}_i \cap \big(((a, \oplus), \top_i), ((a, \ominus), \epsilon)\big), \mathcal{S})$,
- $test(v_i = a, \mathsf{ff}, \mathcal{S}) = apply_{n,i}(\lambda \mathcal{S}_i \,.\, \mathcal{S}_i \cap \big((-\infty, \top_i), ((a, \oplus), \epsilon), ((a, \ominus), \top_i)\big), \mathcal{S})$,
- $test(v_i \neq a, \mathsf{bb}, \mathcal{S}) = test(v_i = a, \textbf{not}\ \mathsf{bb}, \mathcal{S})$,

- $test(v_i < a, tt, \mathcal{S}) = apply_{n,i}(\lambda \mathcal{S}_i \,.\, \mathcal{S}_i \cap \big((-\infty, \top_i), ((a, \oplus), \epsilon)\big), \mathcal{S})$,
- $test(v_i \leq a, tt, \mathcal{S}) = apply_{n,i}(\lambda \mathcal{S}_i \,.\, \mathcal{S}_i \cap \big((-\infty, \top_i), ((a, \ominus), \epsilon)\big), \mathcal{S})$,
- $test(v_i < a, ff, \mathcal{S}) = apply_{n,i}(\lambda \mathcal{S}_i \,.\, \mathcal{S}_i \cap \big(((a, \oplus), \top_0)\big), \mathcal{S})$,
- $test(v_i \leq a, ff, \mathcal{S}) = apply_{n,i}(\lambda \mathcal{S}_i \,.\, \mathcal{S}_i \cap \big(((a, \ominus), \top_0)\big), \mathcal{S})$,
- $test(a \leq v_i, bb, \mathcal{S}) = test(v_i < a, \textbf{not}\ bb, \mathcal{S})$,
- $test(a < v_i, bb, \mathcal{S}) = test(v_i \leq a, \textbf{not}\ bb, \mathcal{S})$,

where $bb \in Bool$ and $\top_i$ denotes the top element in the i-dimensional space. When $\mathbb{I} = \mathbb{Z}$ we have to replace each $(a, \ominus)$ with $(a + 1, \oplus)$.

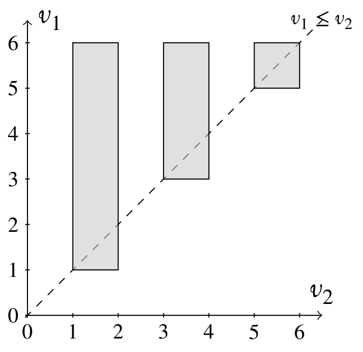


v_2 →

v_1 ↓

$(0, \oplus), (((1, \oplus), \top_0), ((2, \ominus), \epsilon), ((3, \oplus), \top_0),$
$((4, \ominus), \epsilon), ((5, \oplus), \top_0), ((6, \ominus), \epsilon))$
$(8, \ominus), \epsilon$

(a) An original element and its representation



v_2 →

v_1 ↓

$(1, \oplus), (((1, \oplus), \top_0), ((2, \ominus), \epsilon))$
$(3, \oplus), (((1, \oplus), \top_0), ((2, \ominus), \epsilon)), ((3, \oplus), \top_0)), ((4, \ominus), \epsilon))$
$(5, \oplus), (((1, \oplus), \top_0), ((2, \ominus), \epsilon)), ((3, \oplus), \top_0)), ((4, \ominus), \epsilon),$
$((5, \oplus), \top_0), ((6, \ominus), \epsilon))$
$(6, \ominus), \epsilon$

(b) The element and its representation after the test $v_1 \leq v_2$

Figure 4.18: Example application of the *test* function.

The last and the most difficult case is when both atoms in the test are variables. First, consider a simple example for the domain of intervals: $test_v(v_1 \leq v_2, tt, \mathcal{I})$ and $\mathcal{I}$ is such that $\mathcal{I}(v_1) = (a_1, b_1)$, $\mathcal{I}(v_2) = (a_2, b_2)$. After the test operation it must hold that $a_1 \leq v_1 \leq v_2 \leq b_2$, therefore the test introduces two constraints: $a_1 \leq v_2$ and $v_1 \leq b_2$. These constraints may narrow down intervals for v_1 or v_2, and when the interval for any of the variables becomes empty, the result of the test is $\perp_v$. For the domain of boxes the situation is quite similar. The only difference is that we deal with multiple boxes at once and such constraints have to be applied to every one separately. The problem is that with our representation of domain elements, the result of the test operation may become much bigger in size than the input argument. An example of such a situation is presented in Fig. 4.18.

The simplest solution is to apply generic solution with the *split* function. In some cases a more optimal solution may be applied but it is much more complicated. The idea is to traverse the tree describing the input element:

- When we reach the variable with a bigger index (the one that appears higher in the tree) we use the constraint for the variable and go down to sequences of the second variable and apply the constraint. This part is easy, it is similar to the function *apply* with the difference that the interval is chosen when we analyse the first variable.
- When we have reached sequence for variable with a smaller index (the one that appears deeper in the tree) we take constraints for the variable and propagate it upper to the sequences of the first variable. This part is much more difficult. We have to propagate constraints up the tree and this may highly increase the size of the result (as in Fig. 4.18).

4.7.2 Assignment

First we focus on simple cases of the *assign* function. The simplest case is when the assignment expression is a numeric constant, i.e. $assign(v_i, a, \mathcal{S}) = \mathcal{S}'$, where $\mathcal{S} \in \mathbb{S}_n$ and $a \in \mathbb{I}$. It is implemented as follows:

$$assign(v_i, a, \mathcal{S}) = test(v_i = a, tt, \mathcal{S}).$$

The next case is a simple shift $v_i \leftarrow v_i + a$ for $a \in \mathbb{I}$ and $v_i \in \mathsf{Var}$. This can be implemented with *apply* as follows:

$$assign(v_i, v_i + a, \mathcal{S}) = apply_{n,i}(inc_i(a), \mathcal{S}),$$

where the function $inc_i : \mathbb{I} \to \mathbb{S}_i \to \mathbb{S}_i$ is defined as:

$$inc_i(a)\big(\big((p_1, s_1), \dots, (p_k, s_k)\big)\big) = \big((p_1 + a, s_1), \dots, (p_k + a, s_k)\big),$$

where for simplicity we assume that $-\infty + a = -\infty$.

The case of assignment $v_i \leftarrow v_i * a$ for $a \in \mathbb{I}$ and $v_i \in \mathsf{Var}$ depends on the value of the numeric constant a. There are three possibilities:

- The case when $a = 0$ is already covered by the numeric constant assignment.

- The case when $a > 0$ is implemented using the function *apply* as follows:

  $$assign(v_i, v_i * a, \mathcal{S}) = apply_{n,i}(mul_i(a), \mathcal{S}),$$

  where the function $mul_i : \mathbb{I} \to \mathbb{S}_i \to \mathbb{S}_i$ is defined as:

  $$mul_i(a)\big(((p_1, s_1), \ldots, (p_k, s_k))\big) = \big((p_1 * a, s_1), \ldots, (p_k * a, s_k)\big),$$

  where for simplicity we assume that $-\infty * a = -\infty$.

- The case when $a < 0$ is a little bit more difficult, since additionally we have to reverse every sequence for the variable v_i. Let $\mathcal{S} \in \mathbb{S}_n$ such that $\mathcal{S} = \big((p_1, s_1), \ldots (p_k, s_k)\big)$ for some $k > 0$ be the input argument of the assignment. We define a function $inv : \mathbb{P} \to \mathbb{P}$ as:

  $$inv(p) = \begin{cases} (a, \oplus) & \text{if } p = (-a, \ominus) \\ (a, \ominus) & \text{if } p = (-a, \oplus). \end{cases}$$

  Next, we compute the unnormalised reversed version of $\mathcal{S}$ — the sequence $\mathcal{R}'$, as follows:

  $$\mathcal{R}' = \begin{cases} \big((-\infty, s_k), (inv(p_k), s_{k-1}), \ldots, \\ \qquad (inv(p_2), s_1), (inv(p_1), \epsilon)\big) & \text{if } p_1 \neq -\infty \\ \big((-\infty, s_k), (inv(p_k), s_{k-1}), \ldots, (inv(p_2), s_1)\big) & \text{otherwise.} \end{cases}$$

  Finally, the assignment is implemented using the function *apply* as follows:

  $$assign(v_i, v_i * a, \mathcal{S}) = apply_{n,i}(mul_i(-a)), \mathcal{R}),$$

  where $\mathcal{R}$ is the sequence $\mathcal{R}'$ after normalisation.

More complex assignments that use different variables such as $assign(v_i, v_j, \mathcal{S})$ or $assign(v_i, v_i + v_j, \mathcal{S})$ can be computed using the generic implementation.

4.8 Conclusions

In this chapter we have introduced a sweeping line technique to the abstract interpretation. We have used it to create a representation for the domain of *boxes*, which is a disjunctive refinement of the domain of *intervals*. Our construction generalises the construction of the domain that uses LDDs. We have introduced a generic construction of a widening operator for the domain that uses *threshold* points. We have presented two widening operators: the first one with a condition in the construction and the second one that has a single-step precision property depending on the *threshold* points. We have shown examples, where the new widening operator is more precise than the one based on LDDs. Also, we have described generic construction for transfer functions for the presented construction and proposed a number of optimisations.

Chapter 5

Developing Formal Techniques for Typical Code

5.1 Introduction

In this chapter, we present an approach to develop formal techniques, which may be useful in typical, commercial code. This approach relies on existing tools that check if the code realises given specifications, like ESC/Java2 [23] or JmlRAC [24], but it broadens their applicability by automatic generation of specifications based on the existing code. Specifications can be split into two groups:

- Those describing contracts [81] — these are high level specifications that are used to specify behaviour of classes or methods. Program verification usually checks if the code meets these contracts.
- Assisting specifications, which are low level specifications that describe assertions, loop invariants or variant functions. These specifications are used to assist verification of the program code. Assistance is especially desirable in case of loops verification.

If the code is mature and its functionality was well tested using common testing techniques, generated specifications may still be helpful when someone wants to turn the code into a library. The specifications may also act as documentation for maintenance. What is more, specifications describe different aspects of the code than unit tests. If the code is immature, automatic generation of specifications describing correctness properties of the program (such as loop termination, absence of *NullPointerExceptions* or *ArrayIndexOutOfBoundsExceptions*, etc.) combined with static checking can give feedback on what errors are present in the code. With these scenarios in mind we have developed a tool, which helps to build annotations generators and reveals code fragments that are not obvious and should be reviewed by a human. In addition, we have conducted an experiment, which shows that our approach, using even very basic techniques, can give promising results on real, large-scale code.

Applicability of formal methods has been widely discussed [18, 65, 92], but the considerations are mostly limited to safety-critical systems. According to Heitmeyer [65], tools associated with formal methods are not user-friendly, therefore their usage is difficult. Also the effort needed to employ formal methods to real computer programs seems too costly compared to the benefits it may provide. There are many myths concerning the use of formal methods [17, 62] and most of them can be easily dispelled. Craigen et al. [44] and Woodcock et al. [92] provide various examples of applications of formal methods. Recent publications [61] expose benefits of employing formal methods, but there is still a lot of room for improvement. Also, automatic generation of annotations has already been explored and several generators were presented, e.g. CANAPA [25] for JML, Houdini [51] for ESC/Java or Daikon [49] for various programming languages.

5.2 Static Analysis Tools for Java Language

There is a wide variety of tools for Java language that perform static analysis of program code, however the popular ones (and widely accessible) do not employ formal methods. These tools focus on the first three subjects listed in Section 1.2.2, that is checking code conventions, detecting bad programming practices, and calculation of various software metrics. Some of the popular open-source tools for static analysis of Java are[1]:

Eclipse Metrics plugin [91] helps to find unnecessarily complex places in the code. The plugin measures various metrics and performs a dependency analysis of types and packages, which enables to detect cycles. The calculated metrics include simple ones like number of classes, number of children (i.e. the number of direct subclasses of a class), number of interfaces, depth of inheritance tree (DIT), number of overridden methods (NORM), number of methods (NOM), number of fields, lines of code (TLOC, KLOC, MLOC) like specialisation index, or more complex ones like McCabe cyclomatic

[1]A broader list of tools can be found here: `http://java-source.net/open-source/code-analyzers`

complexity, weighted methods per class (WMC), lack of cohesion methods (LCOM*), afferent coupling, efferent coupling, instability, abstractness.

Semmle is a more powerful, commercial code analyser that comes with its own Query Language [85]. It allows to create and run checks to enforce specific architecture rules and coding standards.

FindBugs [7, 66, 67] detects code instances — so called *bug patterns*, that are likely to be errors. It uses a number of ad-hoc techniques, which are calibrated to be precise and efficient. These techniques search for syntactic code patterns as well as perform simple intraprocedural dataflow analysis.

JLint [5, 6] is similar to FindBugs, performs simple syntactic checks and interprocedural dataflow analysis. It searches for synchronization problems by doing data flow analysis and building the lock graph.

PMD [27] is similar to FindBugs or JLint but the checks are limited to syntactic checks. The main difference is that PMD looks for some stylistic conventions that might be suspicious in specific situations.

The *CodeStatistics* tool that is presented here is designed to find any user defined patterns and is able to insert annotations into the code (preliminary version of the tool was developed with technical help of Jędrzej Fulara [54]). In [3], statistics collected on Java libraries are used as a motivation to focus on particular aspect of the code in termination analysis. In this approach statistics are used to estimate coverage of a given formal method.

5.3 Towards Practical Formal Methods

This chapter is an attempt to answer the question what does it mean to create specification-based formal methods suitable for typical large-scale business applications, where code correctness is not considered to be critical but it also may be applied for such software. A formal method designed for the real market cannot increase development costs too much. If one could show that a formal method gives acceptable results on various, existing projects, then the code producers might expect that this method would be applicable also to their software. Thus, to rate the created formal method, a large set of typical projects should be used. In this section, a methodology is sketched that can be used to review usability of a formal method in practice.

5.3.1 The Pareto Principle

The Pareto principle is also known as ”80:20” rule, the *law of the vital few* or the *principle of the factor sparsity*. It states that roughly 80% of the effects come from 20% of causes. The original idea is attributed to Vilfredo Pareto, an Italian economist and sociologist. In the early 1900’s he noticed the unequal distribution of wealth, which he observed and measured in his country. The observation was that in the late 1800’s roughly 80% of the land in Italy was owned by 20% of the population. Later, in 1940’s, J. M. Juran suggested that a small number of causes determine most of the results in any situation [56]. In fact, J. M. Juran is responsible for attributing the *law of the vital few* to Pareto [70] and thus creating the *Pareto principle* as we know today. It turned out that principle can be applied to variety of areas, especially economics. We have to note that the *Pareto principle* is just a concept that solid majority of effects tend to come from a minority of causes and the 80:20 ratio is just a rough estimation, thus not always exact. It may as well be 75:25 or 85:15. Applications of the *Pareto principle* may be also found in the computer software business:

- Microsoft noted that 20% of most reported bugs are respoinsible for 80% of the errors and crashes[2].
- Various studies over the years have shown that 60% to 90% of the defects arise from about 20% of the modules [15]. Also Fenton and Ohlsson bring up lots of examples [50], in which the ratio varies considerably, e.g. 12% of modules responsible for 75% of errors, or 38% responsible for 80% of faults.

An important conclusion that comes from the *Pareto principle* is that most of the code in real projects is simple and does not require much time for understanding or analysis.

5.3.2 Proposed Methodology

To successfully convince managers to use formal methods, one should show them that a chosen formal method covers most of existing code automatically, without any intervention of programmers. Creating practical tools that automatically cover 100% of existing code is impossible [72]. A viable solution is to consider development of verification tools that cover automatically the most common situations

[2]Rooney, Paula (October 3, 2002), Microsoft’s CEO: 80–20 Rule Applies To Bugs, Not Just Features, ChannelWeb `http://www.crn.com/news/security/18821726/microsofts-ceo-80-20-rule-applies-to-bugs-not-just-features.htm`

appearing in real software. These places are usually obvious and not interesting, evidently not sufficiently hard to be handled by expensive programmers. Big parts of systems can even be generated automatically from requirements or specifications [64] and one does not want to waste time on them. All these common situations create a 'bureaucratic burden', very problematic to deal with by the business. A code reviewer should focus on pieces of code, which are by nature complicated or written improperly. These parts may contain programming tricks, hacks or require analysis, or understanding bigger context of the created program. They need to be covered manually.

It is assumed here that solving automatically 80% of cases (for example, proving termination property of 80% of loops in the code) would be acceptable in practical projects. Of course, the 80% threshold is arbitrary and one can choose a different value, but it has to be remembered that solving the remaining 20% may be very difficult and, in some cases, even impossible. Since Pareto principle is well known in business world, it may be accepted by business managers. They can expect that automatic analysis of the vast majority of source code should be performed easily, cheap and fast. To successfully convince managers to use formal methods, one should show them that a chosen formal method covers most of existing code automatically, without any intervention of programmers.

Modern programming languages offer developers big flexibility in creating the code. This may lead to some inconsistencies in the code that can result in errors, which are hard to find. Using tests one can check, if the code meets functional requirements, but to find programming errors (such as null pointer dereferences, not terminating loops, etc.) that occur in rare, but sometimes critical situations, one should employ other techniques. This is especially important when the code changes its original context of use, what often happens in case it is turned to a library or is a subject to maintenance tasks. Generating logical conditions assuring absence of such inconsistencies may be the first step in introducing formal methods in real, large-scale code. Such conditions can be safely generated automatically. However, specifications that describe program semantics (for example, method pre- or postconditions) might be generated only for mature and well tested code. When they would be generated from incorrect code, they could be even misleading.

Building simple verification methods should be done iteratively. This process involves following steps:

1. Select code constructions one wants to handle — e.g. `for` loops.
2. Execute created method on the selected set of projects.
3. Find cases that are not handled yet.

4. From these cases pick up one, for which one is able to provide generic solution.

5. Mark as solved cases covered by the solution.

6. Generate statistics and calculate effectiveness.

7. If the effectiveness does not meet the selected threshold, go to step 2.

To apply this strategy in practice, one has to identify interesting constructions and their frequencies in the real code. Collecting frequencies should be done automatically, based on a big, representative sample of software. In our work we have developed a flexible tool to compute necessary statistics on the code and estimate how effective given annotation generation (and verification) method is.

For example, let us consider classification of Java `for` loops in the context of generating termination conditions. First step is to identify all interesting code constructions – that is all `for` loops that appear in the code. Next, we need to take a look at all loops and try to find patterns, for which we could easily generate the termination condition. The simplest pattern is a `for` loop, which increments a counter up to some constant and the counter is updated only in the update expression of the loop, presented in Fig. 5.1. Creating termination condition for this case is trivial.

```
//@ decreases 15 - i;
for (int i = 0; i < 15; i++) {
    // Loop body that does not modify i
}
```

Figure 5.1: Simple for loop with *decreases* formula.

The next step is to generate statistics — how many loops matching this pattern are present in the code, and discover what part of all interesting `for` loops is covered by this case. After this, we know for how many loops we can generate termination condition. If this coverage does not meet the selected threshold, we should do another iteration — look at the `for` loops that are not yet covered, and produce new patterns.

Each iteration of the presented methodology can take advantage of a different strategy and technique as far as the goal — achieving desired threshold, is accomplished. The main advantage of the presented procedure is a possibility to use different verification techniques, each applied where it performs best.

A formal method created using this approach may give satisfactory results on a large set of typical projects. Such result should convince managers and developers that such solution can be successfully applied in their work.

In this study, it is shown that it is possible to achieve, using simple methods, the expected by the market approximate 80% coverage for a practical problem. The above methodology is applied here to generate annotations for `for` loops termination in Java programs. The results are described in Section 5.5.

5.4 The *CodeStatistics* Tool

To make the above-described approach applicable in practice, a tool that helps to rate the effectiveness of a verification method for a programming language is necessary. For this, *CodeStatistics*[3] was developed. This tool is designed for Java programming language. It can recognise patterns specified by the user, count their frequencies in given Java projects and output all matching code fragments — together with their locations in the project. Together with the methodology proposed in Section 5.3.2, the *CodeStatistics* tool can be used to measure effectiveness of designed formal methods.

5.4.1 Abstract Syntax Tree

Abstract Syntax Tree (AST) is a tree representation of the structure of the source code written in a specific programming language [1], which in our case is Java. Each node in the tree denotes a construct that occurs in the source code. An example of AST representation for a small code fragment is presented in Fig. 5.2. In *CodeStatistics* AST is generated for each analysed source file. The structure of generated tree has some changes and enhancements compared to the regular Java AST. Nodes that occur in our tree may contain pieces of information available at compile time, for example:

- type and scope of variables,
- values that are evaluated at compile time — e.g. access to static final integer fields is translated to the actual value,
- access to the full hierarchy of types, including all superclasses and interfaces.

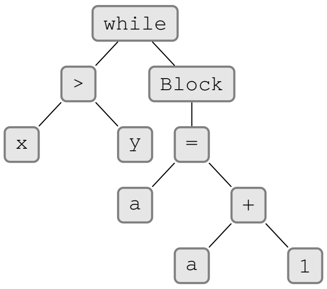
The information is included in the AST since it may be later required to find code patterns in the tree.

Patterns that *CodeStatistics* finds can be expressed as structural conditions for the AST. The tool is generic. The patterns are not hardcoded, but supplied by the

[3]Available at the companion disk and `http://www.mimuw.edu.pl/~kjk/phd`

```
while (x > y) {
  a = a + 1;
}
```

(a) Source code of a simple `while` loop



(b) Abstract Syntax Tree

```
<While>
  <Cond> ... </Cond>
  <Block>
    <Assignment>
      <Left>
        <Var name="a"/>
      </Left>
      <Right> ... </Right>
    </Assignment>
  </Block>
</While>
```

(c) XML representation

Figure 5.2: Sample AST representation (b) of a simple `while` loop (a) and corresponding XML representation (c).

user as a part of the system configuration. They should be written in a simple and expressive language. The *CodeStatistics* tool adapts XML as an AST text representation format and XPath as the language to express patterns. The adaptation is described in Section 5.4.4.

5.4.2 Extensible Markup Language

The Extensible Markup Language (XML) is a general-purpose specification for creating custom markup languages recommended by the World Wide Web Consortium (W3C)[4]. It was designed to transport and store data. The XML format is defined, so that it is both human-readable and machine-readable. An example of an XML file is presented in Fig. 5.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<address>
  <name>
    <title>Mr.</title>
    <first-name>Krzysztof</first-name>
    <last-name>Jakubczyk</last-name>
  </name>
  <city voivodeship="Masovian">Warsaw</city>
  </empty-tag>
</address>
```

Figure 5.3: An example of an XML file that stores simple personal information.

[4]Full specification is available at `http://www.w3.org/TR/REC-xml/`

An XML document is a textual document. It begins with an XML declaration that contains some information about the document itself like version of the XML format (according to W3C specification) and character encoding of the content (first line in Fig. 5.3). The rest of the document consists of a mixture of *tags* and textual content. Tags are fragments that begin with `<` and end with `>`. There are three possible versions of tags:

- starting tag, for example `<address>`,
- finishing tag, for example `</address>`,
- empty element tag, for example `<empty-tag/>`.

An *element* of the document is a fragment that begins with a starting tag and ends with a matching finishing tag (for example, `<title>Mr.</title>`), or an empty element tag. Elements may contain other elements mixed with textual content. These XML elements inside an XML file form a tree structure, e.g. root `name` element from Fig. 5.3 has three child elements: `title`, `first-name` and `last-name`.

Elements may contain a series number of name/value pair elements called *attributes*. For example, the element `city` in Fig. 5.3 has an attribute `voivodeship` with value `Masovian`.

5.4.3 XPath

XPath is a language for selecting nodes in XML documents recommended by W3C. It is based on a tree structure of the XML document. Here we use XPath version 2.0[5]. There are seven kinds of nodes that correspond to different constructs in the syntax of XML. Here we use only four kinds, which are: elements, attributes, text nodes, and document nodes. XPath uses path expressions to navigate in the XML documents. It provides a wide range of functions and operators designed for maximal expressiveness. Here we describe just a fragment of the path syntax of XPath that we have used in our experiment, though *CodeStatistics* makes available a full implementation of XPath 2.0.

Path Expression

A path expression is evaluated with respect to a *context node* and consists of a sequence of steps separated by the `/` operator. It is a binary operator that applies the expression on the right-hand side to every item selected by the expression on the left-hand side. For example, an expression `address/name` selects all elements named `name` that are children of an element named `address` of the context node.

[5]Full specification is available at `http://www.w3.org/TR/xpath20/`

Location Step

Basic step in the path expression of XPath 1.0 is a *location step*. XPath 2.0 is more generic. For example, functions may be used as a right-hand side operand of the operator `/`. The location step consists of an *axis*, *node test*, and zero or more *predicates*:

```
axis :: node-test [predicate]*.
```

The *axis* part indicates navigation direction from the context node within the tree representation of the XML document. Here are some of the possible axes that are available in XPath syntax:

- `ancestor` axis holds ancestor nodes of the context node,
- `attribute` axis holds the attributes of the context node, `@abc` is a shorthand for `attribute::abc`,
- `child` axis holds child nodes of the context node, `xyz` is a shorthand for `child::xyz`,
- `descendant` axis holds all descendant nodes of the context node,
- `descendant-or-self` axis holds the context node and all descendant nodes of the context node, `//` is a shorthand for `/descendant-or-self::node()/`,
- `parent` axis holds the parent of the context node.

Node test may consist of specific node names or more general expressions. Some of the possible node tests are:

- the `*` wildcard character that matches any element or attribute,
- a name that matches a node with the name, for example `child:xyz` matches all children of the context node that have name `xyz`,
- `node()` that matches any type of node,
- `text()` that matches a text node.

The last part of the location step is a *predicate*. It is used to restrict a node-set to those nodes that match the predicate condition. All predicates must be satisfied for a match to occur. Paths that appear in a predicate condition begin at the context of the current step and do not modify the context. When a value of the predicate is numeric it is interpreted as a test on the position of the node, e.g. `xyz[1]` selects the first `xyz` child element while `xyz[last()]` selects the last one. Otherwise, when value is not numeric, it is automatically converted to a boolean. Predicate condition value may evaluate to a node-set. In such case conversion to a boolean returns `true` when the node-set is not empty. Therefore, path expression `xyz[@abc]` returns elements `xyz` that have an attribute `abc`. Predicate expressions may use a variety of operators and functions.

Example

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bibliography>
 <article>
  <title lang="en">Systematic Design of Program Analysis
              Frameworks</title>
  <author>Patrick Cousot</author>
  <author>Radhia Cousot</author>
  <year>1979</year>
 </article>
 <article>
  <title lang="en">Automatic Discovery of Linear Restraints
              among Variables of a Program</title>
  <author>Patrick Cousot</author>
  <author>Nicolas Halbwachs</author>
  <year>1978</year>
 </article>
 <article>
  <title lang="en">Relational Abstract Domain of Weighted
              Hexagons</title>
  <title lang="pl">Abstrakcyjna Dziedzina Numeryczna Wazonych
              Szesciokatow</title>
  <author>Jedrzej Fulara</author>
  <author>Konrad Durnoga</author>
  <author>Krzysztof Jakubczyk</author>
  <author>Aleksy Schubert</author>
  <year>2010</year>
 </article>
 <article>
  <title>The Octagon Abstract Domain</title>
  <author>Antoine Mine</author>
  <year>2006</year>
 </article>
</bibliography>
```

Figure 5.4: Sample XML with a fragment of bibliography of this thesis.

An example of XML file that contains a small fragment of bibliography for this thesis is presented in Fig. 5.4. The file carries some information about four selected articles. The information includes the article title stored in textual content of the `title` element. There may be various language versions, where language of the title is determined by the value of the `lang` attribute of the `title` element. The `article` element also brings author information. Article may have many authors, therefore there may be multiple `author` elements. The last information that is provided by the XML file for an article is the year of publication — the `year` element.

Expression	Refers to
`//article`	all articles
`//author`	all authors
`//article[title/@lang = ″pl″]`	all articles that have Polish title
`//article[title/@lang = ″pl″]/author`	authors of articles with Polish title
`//article[year/text() < 1980]`	articles written before 1980
`//article[author[2]]`	articles that have at least two authors
`//article[year/text() > 1980][author[2]]`	articles written after 1980 with at least two authors

Table 5.1: A few examples of XPath queries with their semantics.

Some examples of XPath queries executed for the root `bibliography` element for the XML file in Fig. 5.4 are presented in Table 5.1.

5.4.4 Application of XML and XPath

In *CodeStatistics* the source code of a program is parsed and transformed to an abstract syntax tree. Then the tree is transformed to an XML format. This way, we obtain a different textual format of the source code than the original one. The new format has some additional information that comes from the compile-time analysis of the code. Additionally, we generate a mapping between XML elements and source code fragments, so that we can point exactly the place in the source file that is represented by an XML element. Next, we apply some XPath queries to the generated XML file. These queries are used to find structural patterns in the XML file. These patterns correspond to structural and syntactic patterns that appear in the source code.

An XPath query that finds all elements representing `while` loops that assign something to a variable `a` is as follows:

```
//While[Block//Assignment/Left/Var/attribute::name=″a″].
```

An example of such loop may be found in Fig. 5.2. The example query searches for all `While` elements that have `Block` child with `Assignment` descendant, not necessarily a child. We want the left side of the assignment to be the variable `a`, therefore `Assignment` element must have child element `Left` with child `Var` that has `name` attribute value equal to `″a″`. When we have found XML elements representing `while` loops we may apply our mapping to receive actual code snippets.

5.4.5 *CodeStatistics* in Eclipse

CodeStatistics works as Eclipse plugin. It provides two operations, which can be executed through the Eclipse tool bar — see Fig. 5.5. The first option *Print as XML* can be executed on a Java source file. It generates an XML representation of the AST, which is used by *CodeStatistics*. The output is printed to the Eclipse console. This option is used to make writing XPath expressions easier for the user. One can generate the XML output for a few sample files, and then, knowing what the XML structure is, he can write proper XPath expressions easier.

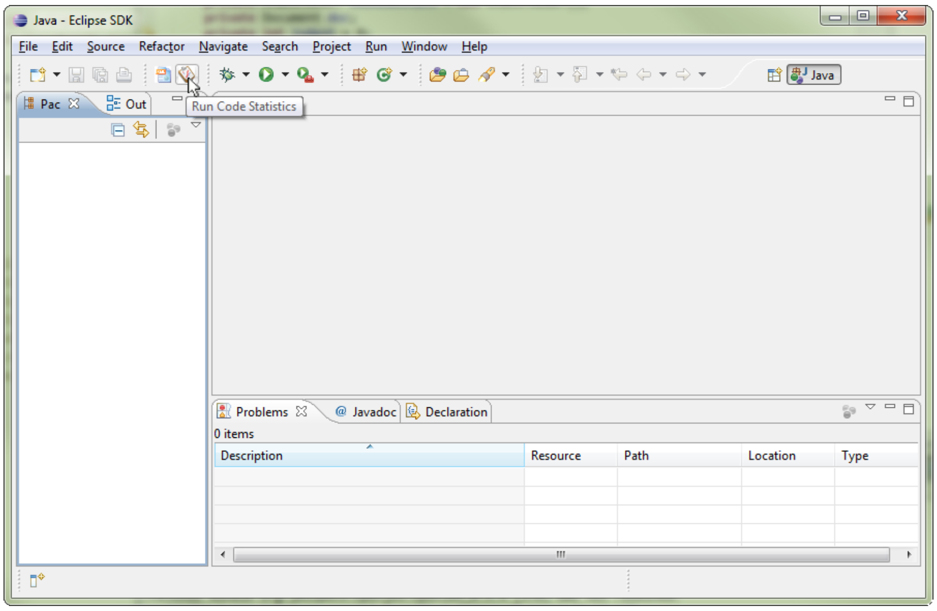


Figure 5.5: *CodeStatistics* plugin options available in Eclipse tool bar.

The second operation provided by *CodeStatistics* that is available in the Eclipse tool bar is *Run Code Statistics*. This option can be executed on a single Java file or on a group of files. The operation executes the *CodeStatistics* on every file in the group as described in Section 5.4.4. It executes XPath expressions on every XML file generated and gathers statistical information about elements matched.

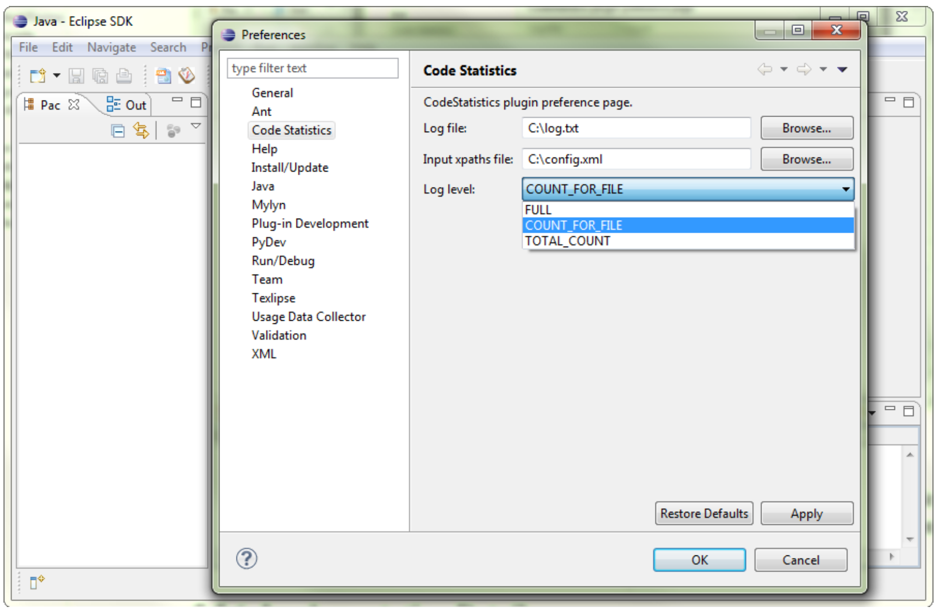
Configuration Options

Configuration options of the *CodeStatistics* Eclipse plugin are available through the Eclipse *Preferences* in *Window* menu. The *CodeStatistics* plugin options page is presented in Fig. 5.6. The entry consists of three options:

- Log file — selection of the output log file. This is the file, to which the output of the *Run Code Statistics* operation is written.

- Input XPath expressions file — which points to an input configuration file that contains the definitions of XPath expressions. This file contains patterns, which *CodeStatistics* will search for in Java source files.

- Log Level — this selection gives user a possibility to choose the logging level. With *FULL* logging level setting *CodeStatistics* outputs exact code fragments that user was eager to find, it includes code snippets corresponding to XML elements found by XPath queries. With *COUNT_FOR_FILE* logging level *CodeStatistics* outputs counts of elements found by every XPath query. These counts are computed on the per file basis. With *TOTAL_COUNT* setting *CodeStatistics* outputs total counts of elements found by every XPath query. For every XPath query it is a sum of matches that were found in all files.


Code Statistics

CodeStatistics plugin preference page.

Log file: C:\log.txt

Input xpaths file: C:\config.xml

Log level: COUNT_FOR_FILE

FULL
COUNT_FOR_FILE
TOTAL_COUNT

Figure 5.6: Configuration options of the *CodeStatistics* plugin.

XPath Expression Configuration File Format

The user specifies a path to the XPath expression configuration file in *CodeStatistics* submenu available from *Preferences* in *Window* menu. These user defined XPath patterns are stored in an XML format file. The file contains the following elements:

- A root element `descriptions`.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<descriptions>
  <all name="AllForLoops" xpath="//ForStatement" />
  <description name="ForWithReturn"
     xpath="//ForStatement[Block//Return]" />
</descriptions>
```

Figure 5.7: Sample configuration file.

- One subelement `all` with two attributes: `name` and `xpath`. This XPath query should describe all interesting code constructions, for example all `for` loops.
- One or more `description` subelements with two attributes: `name` and `xpath`. Each of the elements represents a single pattern that is used to cover a subset of interesting constructs.

A very simple configuration file is presented in Fig. 5.7.

5.4.6 Implementation Details

CodeStatistics is written in the Java language, it uses the following technologies:

Eclipse IDE plugin framework[6] *CodeStatistics* works as Eclipse plugin. It integrates with the editor: a new option is added to the toolbar and a new preferences page is available in the preferences window. *CodeStatistics* is executed on Eclipse project files.

Eclipse Java development tools (JDT)[7] The library provides a Java compiler and a very rich Java abstract syntax tree implementation. With the help of the JDT we have take advantage of some compile-time data that is available during the traversal of the AST. For example, we have access to the class structure or evaluation of compile-time constants.

Java XML DOM API[8] The standard Java library includes support for the creation of documents in the XML format. We use the included XML DOM API to create output XML document from the AST representation.

[6]For details see: `http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html`

[7]For details see: `http://www.eclipse.org/jdt/overview.php`

[8]For details see: `http://www.genedavis.com/library/xml/java_dom_xml_creation.jsp`

[9]For details see: `http://saxon.sourceforge.net/`

Saxon XSLT processor[9] This is an open source library, which provides a complete implementation of XSLT 2.0, XQuery 1.0 and XPath 2.0 recommendations. *CodeStatistics* uses the library to evaluate XPath expressions on generated XML documents.

5.4.7 Usage Example

Here a small example of how *CodeStatistics* can be used is provided. First, we create a new Java project *examples* with only one simple Java source file. The listing of the file is presented in Fig. 5.8. This is a very simple Java class with only one parametrised method that contains a loop, which just prints the iteration number to the standard output.

```java
public class Example {
  public void method(int x) {
        for(int i=0; i<(x+15)*2; i++){
          System.out.println(i);
        }
  }
}
```

Figure 5.8: Listing of an `Example` Java class.

We would like to search how many expressions are, in fact, multiplications. We create a configuration file, which is presented in Fig. 5.9. Our interesting code constructions are all `InfixExpression` nodes. Since we would like to search for all multiplication expressions we search for such `InfixExpression` nodes that have `*` as an operator.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<descriptions>
  <all name="Expressions" xpath='//InfixExpression' />
  <description name="multiplication"
     xpath='//InfixExpression[attribute::operator="*"]' />
</descriptions>
```

Figure 5.9: Example of an XPath configuration of the *CodeStatistics*.

Now we execute *CodeStatistics* with *FULL* logging settings on the file. The output of the execution is presented in Fig. 5.10. The format of the output file is as follows:

- Every XPath expression has a group of matches in the output file. These groups are separated by a line with a series of "`&`" characters. Every group in the file starts with the name of the XPath expression (as specified in the

```
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
TYPE: ALL: Expressions

=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|
FILE : /examples/src/Example.java
[4]
(x + 15) * 2
--------------------------------------------
 [4]
x + 15
--------------------------------------------
 [4]
i < (x + 15) * 2
--------------------------------------------

&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
TYPE: notCategorized

=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|
FILE : /examples/src/Example.java
[4]
x + 15
--------------------------------------------
 [4]
i < (x + 15) * 2
--------------------------------------------

&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
TYPE: multiplication

=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|=|
FILE : /examples/src/Example.java
[4]
(x + 15) * 2
--------------------------------------------
```

Figure 5.10: Output of *CodeStatistics*.

configuration file) with the "`TYPE:`" prefix. A special group, which contains all interesting matches, is prefixed with "`TYPE: ALL:`". Every group name is followed by findings of the XPath expression in subsequent files.

- Findings of the XPath expression in a single file start with a line with a series of "`#`" characters, which is followed by the name of the file with a path to the file within the Eclipse project. Findings in a single file are separated by a line with a series of "`-`" characters.
- Every finding of the XPath expression instance starts with a line number placed in square brackets, e.g. "`[4]`". The line number is followed by a code snippet, which contains the found code fragment.

We can see that *CodeStatistics* found exactly three binary expression items, which are listed in section "`TYPE: ALL: Expressions`". One of them is an expression of type "`multiplication`", which is what we were looking for. The rest are two not categorised expressions that are built with operators "<" and "+".

5.5 Experiment: Loop Termination

5.5.1 Introduction

In this section, an experiment with loop termination conditions generations is presented. It is an application of the methodology proposed in Section 5.3.2, additionally with the *CodeStatistics* tool. In the experiment, a generator of termination conditions of `for` loops in Java programs is developed. First, structural properties of real, production code are analysed. Then, `for` loops are categorised in analysed projects. A subset of loops is chosen, where termination is controlled by a numeric variable and the loop condition contains a numeric variable. The set of test projects is an arbitrary choice of open source Java projects that are popular and widely used in commercial applications. These projects come from various sources like Apache Software Foundation software repository, SourceForge repository, or more commercial ones. The following open source applications are used:

Apache Hadoop[10] is a software platform that lets one easily write and run applications that process vast amounts of data. It allows for the distributed processing of large data sets across clusters of computers using a simple programming framework — MapReduce [46]. It is designed to scale up from single servers to thousands of machines each offering local computation and storage. It does not rely on hardware to deliver high-availability. Instead, it is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures. It is used by large companies such as Adobe, AOL, EBay, Facebook, IBM, or Twitter.

Google App Engine[11] lets to run web applications on Google's infrastructure. App Engine applications are easy to build, easy to maintain, easy to scale as the traffic and data storage needs grow. With App Engine, there are no servers to maintain: one just uploads an application, and it is ready to serve users. Unfortunately, full source code for Google App Engine Java project is publicly unavailable. However, a quite large portion is present in the official distribution, package `org.datanucleus` that is responsible for accessing the data.

[10]For details see `http://hadoop.apache.org/`
[11]For details see `http://code.google.com/appengine/`

JEdit[12] is a cross platform programmer's text editor that is customisable with plugins and macros. There are hundreds of them available. JEdit provides syntax highlighting for more than 200 programming languages and supports the most important character encodings.

Hibernate[13] is a powerful, high performance object/relational persistence and query service. Hibernate lets to develop persistent classes following object-oriented idiom — including polymorphism, association, inheritance, composition, and collections. Hibernate makes it possible to express queries in its own portable SQL extension (HQL), as well as in native SQL, or with an object-oriented Criteria and Example API. It is used by companies, such as AT Labs, Cisco, PriceWaterhouseCoopers, Sony.

Oracle Berkeley DB[14] is an open source, fast, embeddable database that eliminates the overhead of SQL. It stores arbitrary key/value pairs as byte arrays. Berkeley DB can handle multiple threads or concurrent processes accessing the database. It is used by companies, such as Amazon, LinkedIn, AOL, Motorola, or Symantec.

Tomcat[15] is a servlet container developed by the Apache Software Foundation. It implements the Java Servlet and Java Server Pages specifications, and it provides a Web server for Java code to run. It is used by companies, like WalMart or General Motors.

Some details of these projects are presented in Table 5.2. Note that the statistics of occurrences of interesting `for` loops varies between projects — e.g. Hibernate is almost four times bigger than JEdit but the number of `for` loops is a little smaller.

Our goal is to automatically prove termination property of a vast majority (about 80%) of `for` loops in the code. The task is performed by generating *decreases* formula, and then using ESC/Java2 to verify them. The only assumption is that every instruction used in the loop body finishes in finite time, therefore a loop, for which we have generated the *decreases* formula, terminates if all instructions in the loop terminate. The ideas used in this research are very basic. The main advantage of the presented approach is that the effectiveness of such methods is shown on practical, large-scale applications.

[12]For details see `http://www.jedit.org/`
[13]For details see `http://www.hibernate.org/`
[14]For details see `http://www.oracle.com/technology/products/berkeley-db/index.html`
[15]For details see `http://tomcat.apache.org/`

Project Name	**Version**	**Size in KLoC**	**No. of `for`’s**	**`for`’s per 1000 lines**
Oracle Berkeley DB	5.0.34	253	1799	7,11
Google App Engine	1.6.4.1	163	967	5,93
Apache Hadoop	1.0.1	292	1898	6,50
Hibernate	4.1.2	405	855	2,11
JEdit	4.5.1	111	894	8,05
Tomcat	7.0.27	216	1510	6,99
Total		1440	7923	5,50

Table 5.2: Details of analysed projects (sizes are given in Kilo Lines of Code, only interesting `for` loops are considered).

5.5.2 Patterns and Rules

In the current experiment `for` loops are the subject of interest. We would like to find usage patterns of such loops, and possibly for each such pattern generate proper *decreases* formula that could be used to prove termination of the loop. The syntax of `for` loop in Java is similar to the one in C or C++ and is as follows:

```
for (initialisation; termination; increment) {
    S
}
```

where `initialisation` is an expression that initialises the loop and it is executed once, just before the whole loop begins. The `termination` expression is evaluated after every loop iteration, when it evaluates to `false` the loop is terminated. The `increment` expression is also invoked after each iteration. Usually, it increases or decreases some value.

A *control variable* is a numeric variable that controls the number of loop iterations. It is used in the termination expression to verify the loop end. A typical scenario is when the variable is initialised in the initialisation expression and incremented or decremented in the increment expression. Usually, the variable is not changed in the loop body, however during the experiment it turned out that sometimes it happens.

First, one has to define an XPath pattern that describes all the interesting code constructions, which are `for` loops that contain a numeric variable in the termination condition. The basic version of the configuration file for *CodeStatistics* is as follows:

```
<descriptions>
  <all name="For" xpath="//ForStatement[Condition//Name[@numeric]]" />
</descriptions>
```

The element `ForStatement` represents `for` loop and the `Condition` child element stores the termination condition of the loop. The `Name` descendant represents a variable that appears in the condition. The type of the variable is limited to numeric types only, which is represented by the existence of the `numeric` attribute.

Patterns that are used to find subsequent code patterns, for which we are able to create a proper *decreases* condition, are presented in what follows.

Literal

Let us start with the simplest category of `for` loops. It is a loop, where the termination condition is a simple comparison of the control variable with a numeric literal. For loops that match this pattern, we can automatically generate the appropriate JML *decreases* formula. An example of such loop together with a valid *decreases* formula is presented in Fig. 5.11. With *CodeStatistics* one can easily find such loops.

```
//@ decreases 256 - i;
for (int i=1; i <= 256; i++) {
  cftab[i]=m_unzftab[i - 1];
}
```

Figure 5.11: Example of `for` loop to a literal taken from JEdit project (from file `installer.CBZip2InputStream.java`).

```
1   //ForStatement
2     [starts-with(Condition/InfixExpression/@operator, ”<”)]
3     [Condition/InfixExpression/Expr[1]/Name/@scope = ”local”]
4     [Condition/InfixExpression/Expr[1]/Name/@name =
5               Updaters/ChangeVal/LeftOfAssignment/Name/@name]
6     [Updaters/ChangeVal[@kind = ”inc”]
7               [RightOfAssignment[number(Constant/@value) >= 1 or not(node())]]]
8     [Condition/InfixExpression/Expr[2]/Constant/@node = ”NumberLiteral”]
9     [not(Block//ChangeVal/LeftOfAssignment/Name/@name =
10              Condition/InfixExpression/Expr[1]/Name/@name)]
```

Figure 5.12: XPath expression used to find `for` loops to a literal.

A proper XPath expression that is used to find `for` loops that increase the control variable is presented in Fig. 5.12. The XPath may seem complicated but, in fact, it is a conjunction of a series of simple conditions:

line 1 : the search is limited to *ForStatement* elements, which represent `for` loops;

line 2 : makes sure the termination condition (element *Condition*) is a binary expression (element *InfixExpression*), where the operator is either `<` or `<=`;

line 3 : the control variable appears on the left of the condition expression (note that indexing in XPath starts from 1), it must have a local scope — it is either a parameter or a local variable;

lines 4-5 : ensure that the update expression (element *Updaters*) is a value change expression (elements *ChangeVal* represent assignments and prefix or postfix `-`, `++` expressions);

lines 6-7 : the update expression has to increase the value of the control variable — the attribute *kind* of *ChangeVal* element has the value *inc* only for `+=` assignment expressions and `++` prefix/postfix expressions; analogically for `-=` and `--` the attribute has value *dec*; here two cases are accepted:

- when the operator is `++`, then the right hand side of the expression is empty, the corresponding XPath expression fragment is *RightOfAssignment[not(node())]*,
- when the operator is `+=`*<constant>*, where the *<constant>* is a numeric constant with a value greater or equal to 1, then the corresponding XPath expression fragment is *RightOfAssignment[number(Constant/@value) >= 1]*);

line 8 : the right hand side of the condition must be a numeric literal;

lines 9-10 : ensure that the control variable is not modified in the loop body, it means that it does not appear on the left hand side of any assignment expression in the body of the loop — and this is expressed by a formula that makes sure there is no *ChangeVal* descendant of *Block* element, which has the control variable on the left hand side.

Note that the XPath expression presented in Fig. 5.12 handles only loops, where the value of the control variable is increased and the condition is written in the way, where the variable is on the left hand side of the comparison operator, e.g. `x < 2`. In order not to unnecessarily complicate XPath expressions, separate ones are created to handle all four cases (increasing, decreasing the value of the control variable, and two possibilities to write the condition in every case). These cases are analogical to the one from Fig. 5.12.

Statistics of discoveries of `for` loops to a numeric literal found by *CodeStatistics* are presented in Table 5.3. Such loops cover 10,72% of all the numeric `for` loops. Loops, where the numeric literal is some number greater than 0, are usually not good coding practice. Such number literals are called *magic numbers*[16] or *unnamed numerical constants* and they are not desirable, because a modification

[16]For details see `http://en.wikipedia.org/wiki/Magic_number_(programming)`

Project Name	No of `for`'s	To literal	Percentage
Berkley DB	1799	396	22,01 %
Google App Engine	967	17	1,76 %
Hadoop	1898	208	10,96 %
Hibernate	855	78	9,12 %
jEdit	894	84	9,40 %
Tomcat	1510	66	4,37 %
Total	7923	849	10,72 %

Table 5.3: Discoveries of `for` loops to a numeric literal.

of such number in one place might require change in another and these are not automatically synchronised. It would be better to use a `static final` field to store the value.

Constant
In the next step the search is extended to loops that use a constant in the termination comparison expression. An example of such loop is presented in Fig. 5.13. This case is an extension of the previous one. Since a numeric literal has a constant value, all `for` loops found in previous case are also covered by the current case. Thanks to using JDT, all expressions that are evaluated by the JDT compiler into a constant are treated as a constant in the analysis. For example, the expression `CONST+3`, where `CONST` is a `final` field with a value 5, is evaluated by the JDT parser to a constant `8`. This way, only expressions that are built with constants are handled easily.

```
private final int RECORDNUM = 5000;
...
//@ decreases RECORDNUM - i;
for (int i=0; i < RECORDNUM; i++) {
  theData.setKey(i);
  theData.setData("Record " + i);
  da.dataByKey.put(theData);
}
```

Figure 5.13: Example of `for` loop to a constant, taken from BerkleyDB project (from file `android.JECursoAdapter.JECursorAdapterExample.java`).

The XPath for this case is just a slight modification of the one from Fig. 5.12. The new XPath is presented in Fig. 5.14, where the modified part is emphasised. In line 8 the test for the element `Constant` is simply removed. In the case of loops that match this pattern, the *decreases* formula can be easily generated automatically.

```
//ForStatement
  [starts-with(Condition/InfixExpression/@operator, "<")]
  [Condition/InfixExpression/Expr[1]/Name/@scope = "local"]
  [Condition/InfixExpression/Expr[1]/Name/@name =
          Updaters/ChangeVal/LeftOfAssignment/Name/@name]
  [Updaters/ChangeVal[@kind = "inc"]
          [RightOfAssignment[number(Constant/@value) >= 1 or not(node())]]]
  [Condition/InfixExpression/Expr[2]/Constant]
  [not(Block//ChangeVal/LeftOfAssignment/Name/@name =
          Condition/InfixExpression/Expr[1]/Name/@name)]
```

Figure 5.14: XPath expression used to find `for` loops to a constant.

Analogically to the previous case, multiple XPath expressions must be created, that cover loops, which decrease or increase the value of the control variable, and situations, where the condition is written in the different direction.

Project Name	No of `for`'s	To constant	Percentage
Berkley DB	1799	658	36,58 %
Google App Engine	967	19	1,96 %
Hadoop	1898	322	16,97 %
Hibernate	855	94	10,99 %
jEdit	894	95	10,63 %
Tomcat	1510	99	6,56 %
Total	7923	1287	16,24 %

Table 5.4: Discoveries of `for` loops to a constant.

Statistics concerning discoveries of `for` loops to a constant are presented in Table 5.4. We can see that we have gained additional 5,5% over the previous coverage and thus total coverage is 16,24% of `for` loops.

Local expression

Another simple case that can be solved automatically is when the control variable is compared to an arithmetic expression composed only from constants or local numeric variables that are not modified in the loop body nor in the update expression. An example of such loop, with corresponding *decreases* formula, is presented in Fig. 5.15.

Decreases formula can be easily generated, because the result of the arithmetic expression in the loop condition does not change during the execution of the loop. Here the search is limited to a situation, where the update expression updates only the control variable. Therefore, the rest of variables that appear in the loop condition must not change in the loop update expression. Thus, one only has to ensure

```
private String indent(int indentation) {
  ...
  //@ decreases 4 * indentation - i;
  for (int i = 0; i < 4 * indentation; i++) {
    sb.append(″ ″);
  }
  ...
}
```

Figure 5.15: Example of `for` loop to an expression with local numeric variables taken from Google App Engine project (from file `org.datanucleus.query.node.Node.java`).

that these variables are not modified in the loop body. The new case covers the preceding ones.

```
1   //ForStatement
2     [starts-with(Condition/InfixExpression/@operator, ”<”)]
3     [Condition/InfixExpression/Expr[1]/Name/@scope = ”local”]
4     [Condition/InfixExpression/Expr[1]/Name/@name =
5              Updaters/ChangeVal/LeftOfAssignment/Name/@name]
6     [Updaters/ChangeVal[@kind = ”inc”]
7              [RightOfAssignment[number(Constant/@value) >= 1 or not(node())]]]
8     [count(Condition/InfixExpression/Expr[2]//*) =
9      count(Condition/InfixExpression/Expr[2]//(Constant | Expr | InfixExpression |
10                                 Name[@scope=”local”][@numeric]))]
11    [not(Block//ChangeVal/LeftOfAssignment/Name/@name =
12             Condition/InfixExpression//Name/@name)]
```

Figure 5.16: XPath expression used to find `for` loops to a local expression.

The XPath expression for the case is presented in Fig. 5.16. Some of the conditions are the same as in the XPath expression for the literal case, the difference occurs in lines 8-12. First, lines 8-10 ensure the right hand side expression in the loop condition is composed of binary expressions (*InfixExpression* element), subexpressions of binary expressions (*Expr* element), constants, or local numeric variables. In fact, the last two are interesting, other are just building block elements of the expression. The condition in lines 11-12 states that any variable that appears in the condition of the loop cannot be modified in the loop body. In the previous case, similar constraint involved only the control variable, since the right hand side was constant. Analogically to previous cases, multiple XPath expressions must be created, that cover loops, where the value of the control variable is increased or decreased, and where the control variable appears either on the left or the right hand side in the loop condition expression.

Statistics concerning discoveries of `for` loops to an expression built with con-

Project Name	**No of `for`'s**	**To local expression**	**Percentage**
Berkley DB	1799	1047	58,20 %
Google App Engine	967	100	10,34 %
Hadoop	1898	782	41,20 %
Hibernate	855	336	39,30 %
jEdit	894	269	30,09 %
Tomcat	1510	360	23,84 %
Total	7923	2894	36,53 %

Table 5.5: Discoveries of `for` loops to a local expression.

stants and local numeric variables are presented in Table 5.5. This case covers both previous cases. This time the gain is bigger, it is around 20%, and therefore total coverage is 36,53% of numeric `for` loops.

Final field

Very often programmers iterate over an array. If the array variable is not modified during the interaction then the length of the table does not change. This is because it is a `final` field of the array object. A `final` field in Java is such field that can be assigned only once in a constructor or directly in the field definition fragment. Therefore, the `final length` field of an array object, which is of a primitive type, does not change its value during the life of the array object, and thus we are able to generate proper *decreases* formula. An example of a `for` loop, where a final field is used in the loop condition along with generated *decreases* formula, is presented in Fig. 5.17.

```
//@ decreases params.length - j;
for (int j=0; j < params.length; ++j) {
  if (!params[j].equals(parameterTypes[j])) {
    eq=false;
    break;
  }
}
```

Figure 5.17: Example of a `for` loop to a final field taken from Hibernate project (from file `org.hibernate.bytecode.internal.javassist.FastClass.java`).

The example of iteration over an array is generalised here. A very restrictive ”alias safe” approach is chosen. If the guard (in our simple example `tab.length`) is of the form $o_1.o_2.o_3...o_n$, then:

- o_1 must either be declared as final or be a local variable that is not assigned in the loop body,

- all `o`$_2$, `o`$_3$, ... `o`$_n$ must be declared as final fields.

In this case, the well-known problem of modification using aliases (when the same object is referenced by multiple variables and modifications to one of them induces changes of the others) is eliminated. The *CodeStatistics* tool generates `final` attribute for `Name` elements that represent variable accesses, which meet the presented safety criteria. For example, for an array access `some_array.length`, where `some_array` is a local variable, a following XML element is generated:

```
<Name final="true" fullname="some_array.length" name="some_array" name="local"/>
```

The XPath expression for the analysed case is an extension of the one that appeared in the previous case, it is presented in Fig. 5.18. The only difference is in line 10, where in the loop condition `Name` elements that have `final` attribute are admitted. These are elements that match the restrictive alias-safe approach.

```
1    //ForStatement
2      [starts-with(Condition/InfixExpression/@operator, "<")]
3      [Condition/InfixExpression/Expr[1]/Name/@scope = "local"]
4      [Condition/InfixExpression/Expr[1]/Name/@name =
5                  Updaters/ChangeVal/LeftOfAssignment/Name/@name]
6      [Updaters/ChangeVal[@kind = "inc"]
7                  [RightOfAssignment[number(Constant/@value) >= 1 or not(node())]]]
8      [count(Condition/InfixExpression/Expr[2]//*) =
9       count(Condition/InfixExpression/Expr[2]//(Constant | Expr | InfixExpression |
10                                 Name[@scope="local"][@numeric] | Name[@final]))]
11     [not(Block//ChangeVal/LeftOfAssignment/Name/@name =
12                 Condition/InfixExpression//Name/@name)]
```

Figure 5.18: XPath expression used to find `for` loops to a final expression.

In fact, it may happen that the update expression of the loop is expressed by an assignment in such way that the right hand side is a sum of the variable from the left hand side and some constant value (e.g. `i = i + 3`). This can be expressed by the following condition:

```
Updaters/ChangeVal[@kind="assign" and
    RightOfAssignment[InfixExpression/Expr[1]/Name/@name =
       parent::ChangeVal/LeftOfAssignment/Name/@name]
    /InfixExpression[@operator="+"][number(Expr[2]/Constant/@value) >= 1]]
```

This condition is used to extend the one in Fig. 5.18 in lines 6-7.

As it turns out, qualified names with many access steps are not very frequent in practice. Usually, only one step is used (as in the example from Fig. 5.17). Using

Project Name	**No of `for`'s**	**Local expression with final**	**Percentage**
Berkley DB	1799	1312	72,93 %
Google App Engine	967	638	65,98 %
Hadoop	1898	1324	69,76 %
Hibernate	855	711	83,16 %
jEdit	894	547	61,19 %
Tomcat	1510	903	59,80 %
Total	7923	5435	68,60 %

Table 5.6: Discoveries of `for` loops to a final expression.

CodeStatistics, it occurs that 68,60% of all `for` loops match this pattern, which gives us additional 32% (see Table 5.6). Cases, which have been analysed so far, cover a great majority of the interesting `for` loops. Next cases become more complicated and cover much smaller numbers of `for` loops.

Update in loop body

In all the previous cases the control variable was updated in the update expression of the loop and not changed in the loop body. As it turns out, it happens sometimes that the control variable is updated in the loop body but in the fashion, which does not influence the termination. Two situations are especially interesting:

possible speedup when the control variable is updated in the update expression and possibly in the body of the loop, but when the second happens all updates in the body are in the same direction as in the update expression — all changes increase the variable or all changes decrease it (see Fig. 5.19),

update only in body when the control variable is not updated in the update condition of the loop at all, but instead it is updated in the body of the loop but all these updates are in the same direction.

When the update inside the loop body is not performed in an inner loop then it is possible to generate a *decreases* formula for the case. The inner loop is problematic, because the *decreases* formula must evaluate to a number that is greater or equal to 0. If there are no changes in inner loop one can just include the sum of possible modifications of the variable in the *decreases* formula. An example of the first case (termination speedup) with generated *decreases* formula is presented in Fig. 5.19. In the example, `1` is added to the generated termination condition.

An XPath for the speedup case is an extension of the previous case. The condition in lines 11-12 is updated. An actual XPath expression for the possible termination speedup case is presented in Fig. 5.20. First, lines 11-12 ensure that any

```
public static void main(String[] args) {
  ...
  //@ decreases args.length + 1 - i;
  for (int i=0; i < args.length; i+=1) {
    if (args[i].equals("-h")) {
      i+=1;
      homeDir=args[i];
    } else {
      throw new IllegalArgumentException("Unknown arg: "+args[i]);
    }
  }
  ...
}
```

Figure 5.19: Example of a `for` loop, where control value is additionally increased in the loop body, taken from BerkleyDB project (from file `com.sleepycat.je.cleaner.MakeMigrationLogFiles.java`).

variable that appears in the right hand side of the loop condition is not updated in the loop body. In lines 13-14 it is guaranteed that any assignment to the control variable is not performed in an inner loop. Next, lines 15-18 ensure that every change in loop body that is done to a control variable is performed in the correct direction — it increases the variable by a number greater or equal to 0. The *possible speedup* case is an extension of previous one, it covers all the loops found before.

```
1   //ForStatement
2     [starts-with(Condition/InfixExpression/@operator, "<")]
3     [Condition/InfixExpression/Expr[1]/Name/@scope = "local"]
4     [Condition/InfixExpression/Expr[1]/Name/@name =
5               Updaters/ChangeVal/LeftOfAssignment/Name/@name]
6     [Updaters/ChangeVal[@kind = "inc"]
7               [RightOfAssignment[number(Constant/@value) >= 1 or not(node())]]]
8     [count(Condition/InfixExpression/Expr[2]//*) =
9      count(Condition/InfixExpression/Expr[2]//(Constant | Expr | InfixExpression |
10                                 Name[@scope="local"][@numeric] | Name[@final]))]
11    [not(Block//ChangeVal/LeftOfAssignment/Name/@name =
12              Condition/InfixExpression/Expr[2]//Name/@name)]
13    [not(Block//*[@isLoop]//ChangeVal/LeftOfAssignment/Name/@name =
14              Condition/InfixExpression/Expr[1]/Name/@name)]
15    [not(Block//ChangeVal[@kind != "inc" or
16          not(RightOfAssignment[not(node()) or number(Constant/@value) >= 0])]
17          /LeftOfAssignment/Name/@name =
18              Condition/InfixExpression/Expr[1]/Name/@name)]
```

Figure 5.20: XPath expression used to find `for` loops to a final expression with control variable speedup in the loop body.

The second case, when the update is done only in the body of the loop, is disjoint with previous ones since no update to the control variable is performed in the update expression. An example is presented in Fig. 5.21.

```
private static final String[] DEFAULT_MIME_MAPPINGS = ...
...
//@ decreases DEFAULT_MIME_MAPPINGS.length + 1 - i;
for (int i=0; i < DEFAULT_MIME_MAPPINGS.length; ) {
  ctx.addMimeMapping(DEFAULT_MIME_MAPPINGS[i++],
                     DEFAULT_MIME_MAPPINGS[i++]);
}
```

Figure 5.21: Example of a `for` loop, where the value of the control variable is only increased in the loop body, taken from Tomcat project (from file `org.apache.catalina.startup.Tomcat.java`).

In fact, the XPath expression for the case is quite similar to the one presented in Fig. 5.20, and thus it not present here. First, the requirement from lines 4-5 is changed to ensure the control variable does not appear in the *updaters* at all. Next, it is assured that at least one change to the control variable is always performed in the loop iteration. It is formulated as follows: a number of all assignments to the control variable in the loop body must be greater than number of assignments that appear inside conditions (if, switch, conditional expression), outside inner loops or try-catch blocks.

Project Name	**No of `for`'s**	**Only in body**		**Possible speedup**		**Total**	
Berkley DB	1799	0	0,00 %	1338	74,37 %	1338	74,37 %
Google App Eng.	967	12	1,24 %	640	66,18 %	652	66,18 %
Hadoop	1898	4	0,21 %	1364	71,87 %	1368	71,87 %
Hibernate	855	3	0,35 %	711	83,16 %	714	83,16 %
jEdit	894	2	0,22 %	548	61,30 %	550	61,30 %
Tomcat	1510	1	0,07 %	917	60,73 %	918	60,73 %
Total	7923	22	0,28 %	5518	69,65 %	5540	69,92 %

Table 5.7: Discoveries of `for` loops with possible speedup in loop body and without update in the update condition.

Statistics of both cases are presented in Table 5.7. As we have noted before, the increase of coverage is much smaller now, around 1,3%.

Immutable objects

An immutable object is an object, a state of which cannot be modified after it is created. However, Java does not have a special marking for classes the instances of which are immutable, one may find some in the standard Java library. When a class is defined as `final` then it is not possible to subclass it, and thus to change behaviour of it. An example of such class is `String`, which is present in the standard Java library. The class is defined as `final` and execution of any method of a `String` object does not change the state the object — `String` instances are immutable. It turns out that the method `length()` of the class is quite frequently used in the condition of `for` loops. Since `String` object is immutable, every `length` method call gives the same result. In fact, in standard Java implementation the method returns a private integer `final` field. Here a special case is created to find all `for` loops, where the loop condition contains an execution of the `length` method on a variable of class `String`, which is not assigned in the loop body nor in the update expression. An example of such loop is presented in Fig. 5.22. In fact, all the previous cases are extended to allow also unmodified variables of type `String` in the condition expression.

```
String abbrev = ...
...
for (int i=0; i < abbrev.length(); i++) {
  if (abbrev.charAt(i) == '#') {
    m_pp.addElement(abbrev.substring(lastIndex,i));
    lastIndex=i + 1;
  }
}
```

Figure 5.22: Example of a `for` loop, where a string length is used in the loop condition, taken from JEdit project (from file `org.gjt.sp.jedit.Abbrevs.java`).

In the XML representation in *CodeStatistics*, every method invocation is represented by *MethodInvocation* element. Every such element has *methodName* attribute, which stores the name of the method called. In most cases a method invocation has an expression prefix (eg. in `some_string.length()`, `some_string` is the expression prefix), it may not exist when we call a method on current object. An element representing the prefix expression is a first child of the *MethodInvocation* element. If method invocation has any arguments then *MethodArguments* child is present with arguments as children. When we consider `length` method invocation of a `String` object, the expression prefix child always exists, since we do not analyse the standard Java library. The prefix is the `String` object instance. The *methodName* attribute has *length* value and there is no *MethodArguments* subelement present.

XPath expressions for this case are extensions of the ones from the previous case — *Update in loop body*. An updated XPath expression for the case of possible termination speedup is described here in detail. The expression is presented in

```
1   //ForStatement
2     [starts-with(Condition/InfixExpression/@operator, ”<”)]
3     [Condition/InfixExpression/Expr[1]/Name/@scope = ”local”]
4     [Condition/InfixExpression/Expr[1]/Name/@name =
5               Updaters/ChangeVal/LeftOfAssignment/Name/@name]
6     [Updaters/ChangeVal[@kind = ”inc”]
7               [RightOfAssignment[number(Constant/@value) >= 1 or not(node())]]]
8     [count(Condition/InfixExpression/Expr[2]//*) =
9      count(Condition/InfixExpression/Expr[2]//(Constant | Expr | InfixExpression |
10     Name[@final] | Name[@scope=”local”][@numeric or @type=”String”] |
11     MethodInvocation[*[1][@type=”String”]][@methodName = ”length”]))]
12    [not(Block//ChangeVal/LeftOfAssignment/Name/@name =
13              Condition/InfixExpression/Expr[2]//Name/@name)]
14    [not(Block//*[@isLoop]//ChangeVal/LeftOfAssignment/Name/@name =
15              Condition/InfixExpression/Expr[1]/Name/@name)]
16    [not(Block//ChangeVal[@kind != ”inc” or
17          not(RightOfAssignment[not(node()) or number(Constant/@value) >= 0])]
18          /LeftOfAssignment/Name/@name =
19              Condition/InfixExpression/Expr[1]/Name/@name)]
20    [Condition//MethodInvocation]
```

Figure 5.23: XPath expression used to find `for` loops, which have `length` method call of a `String` object in the loop condition.

Fig. 5.23. Differences from Fig. 5.20 are emphasised in bold. First, in line 10, local variables of type *String* are admitted. Then, in line 11, *MethodInvocation* elements are admitted, which represent invocations of the *length* method on `String` objects (first child is of the correct type and *methodName* attribute is correct). The last part is the introduction of line 20, where it is ensured that any method invocation must appear in loop condition. This way, loops that have *length* method call in the condition with a possibility of termination speedup are covered. The last fragment is to make the case disjoint with previously described ones. Analogically, a case is created, where the control variable is updated only in the loop body.

Project Name	No of `for`'s	String		Total	
Berkley DB	1799	2	0,11 %	1340	74,49 %
Google App Engine	967	10	1,03 %	662	68,46 %
Hadoop	1898	11	0,58 %	1379	72,66 %
Hibernate	855	3	0,35 %	717	83,86 %
jEdit	894	24	2,68 %	574	64,21 %
Tomcat	1510	30	1,99 %	948	62,78 %
Total	7923	80	1,01 %	5620	70,93 %

Table 5.8: Discoveries of `for` loops to the length of string.

Statistics of the findings are presented in Table 5.8. This time 1.01% of the remaining cases is covered. Our total coverage is 70,93%.

Condition with conjunction
When the termination condition of the `for` loop evaluates to false, the loop is terminated. Therefore, if the condition contains a conjunction of conditions, it is enough that one of the conditions evaluates to false to terminate the loop. The current case extends the previous one and applies this observation. The difference from the previous case is that the restrictions on condition expression now apply to either the right or the left conjunct of the expression. The *decreases* formula for the conjunction is the same as for the matching one conjunct only, thus may be generated exactly the same way as in the previous case. An example of a conjunction in the loop condition, together with generated *decreases* formula, is presented in Fig. 5.24.

```
MBeanAttributeInfo attrs[] = minfo.getAttributes();
...
//@ decreases attrs.length - i;
for (int i = 0; mattrType == null && i < attrs.length; i++) {
  if (attribute.equals(attrs[i].getName()))
    mattrType = attrs[i].getType();
}
```

Figure 5.24: Example of a `for` loop with a conjunction, taken from Tomcat (from file `org.apache.catalina.ant.jmx.JMXAccessorSet-Task.java`).

The XPath expression is based on the one described in the previous section. First, the restriction for the existence of method invocation in line 20 in Fig. 5.23 is removed. Also, one has to make sure now that the condition is a conjunction:

```
//ForStatement[Condition/InfixExpression[@operator = "&&"]]
```

and that all variables used in the interesting condition (the one used for generating the *decreases* formula) part are not modified in the other one, e.g. when the left conjunct is the interesting one:

```
[not(Condition/InfixExpression/Expr[2]//ChangeVal/LeftOfAssignment/Name/@name =
        Condition/InfixExpression/Expr[1]//Name/@name)].
```

The rest of the XPath expression is derived from the previous section, with a modification that all constraints on element:

```
Condition/InfixExpression/Expr[1]
```

must be updated in order to apply to exactly one of the conjuncts, e.g. for the left conjunct it should be changed to:

```
Condition/InfixExpression/Expr[1]/InfixExpression/Expr[1].
```

Of course, more patterns must be created, since for every one of the conjuncts we have multiple situations to cover: increasing or decreasing the control variable and two ways to write the condition.

Project Name	No of `for`'s	And		Total	
Berkley DB	1799	13	0,72 %	1353	75,21 %
Google App Engine	967	13	1,34 %	675	69,80 %
Hadoop	1898	33	1,74 %	1412	74,39 %
Hibernate	855	5	0,58 %	722	84,44 %
jEdit	894	10	1,12 %	584	65,32 %
Tomcat	1510	69	4,57 %	1017	67,35 %
Total	7923	143	1,80 %	5763	72,74 %

Table 5.9: Discoveries of `for` loops, where the condition is a conjunction and one of the conjuncts is interesting.

XPath expressions created for the described case are disjoint with the previous ones. Statistics of the findings are presented in Table 5.9. New XPath expressions cover 1,80% of the remaining numeric `for` loops, which gives us total coverage of 72,74%.

No dangerous function call

Note that all previous cases were thread safe. In the current case access to fields that are not final is taken into consideration. Therefore, a single thread execution of the analysed loop is assumed, that no other thread can change values of non-final fields. Let us consider a situation when the loop body, the update expression and the condition expression do not contain any method execution nor class instance creation. This means that no function is called during the execution of the loop and therefore, if a field is not changed explicitly in the loop by an assignment then it is not modified at all. In fact, it is possible to be a little less restrictive. What one would like here is assurance that any function from the body of the loop does not modify any field. There are several widely used `final` classes in the standard Java library that have this property. These classes are, for example: `String`, `StringBuffer`, `StringBuilder`, `Integer`, `Array`. Any call to a method from any of the classes does not modify state of other objects. An example is presented in Fig. 5.25. If the update expression is a standard increase of the control variable and other variables that are used in the loop conditions are not modified in the loop body, one can easily generate *decreases* formula for the loop despite existence of method calls from limited classes.

An XPath for the case is presented in Fig. 5.26. First of all, restrictions are introduced in lines 13-15, which do not allow any method invocation or object creation

```
private String securityRoles[] ...
...
for (int i=0; i < securityRoles.length; i++) {
  if (role.equals(securityRoles[i])) {
    n=i;
    break;
  }
}
```

Figure 5.25: Example of a `for` loop with a safe `equals` method call on a `String` object, taken from Tomcat (from file `org.apache.catalina.core.StandardContext.java`).

in the condition itself. Creating object instances in the body of the loop is also disallowed. Next, in lines 16-17, method calls on objects from safe classes listed before are admitted while other are forbidden. In lines 18-19 the search is narrowed down to minimise number of loops overlapping with previously covered cases. Similarly to previous cases, multiple patterns are created. Also conjunction cases are included here.

```
1   //ForStatement
2     [starts-with(Condition/InfixExpression/@operator, ”<”)]
3     [Condition/InfixExpression/Expr[1]/Name/@name =
4                 Updaters/ChangeVal/LeftOfAssignment/Name/@name]
5     [Updaters/ChangeVal[@kind = ”inc”]
6                 [RightOfAssignment[number(Constant/@value) >= 1 or not(node())]]]
7     [not(Block//ChangeVal/LeftOfAssignment/Name/@name =
8                 Condition/InfixExpression/Expr[2]//Name/@name)]
9     [not(Block//*[@isLoop]//ChangeVal/LeftOfAssignment/Name/@name =
10                Condition/InfixExpression/Expr[1]/Name/@name)]
11    [not(Block//ChangeVal[@kind != ”inc” or
12          not(RightOfAssignment[not(node()) or number(Constant/@value) >= 0])]
13          /LeftOfAssignment/Name/@name =
14                Condition/InfixExpression/Expr[1]/Name/@name)]
13    [not(Block//ClassInstanceCreation)]
14    [not(Condition//MethodInvocation)]
15    [not(Condition//ClassInstanceCreation)]
16    [not(Block//MethodInvocation[not(*[1][@type=”String” or @type=”Array” or
17       @type=”StringBuffer” or @type=”StringBuilder” or @type=”Integer”])])]
18    [Condition/InfixExpression/Expr[2]//Name[@scope=”field”]
19                [not(@final) or parent::MethodInvocation]]
```

Figure 5.26: XPath expression used to find `for` loops with only safe method invocations.

Statistics of the findings are presented in Table 5.10. Unfortunately, the current case may not be fully disjoint with previous ones. It may happen that a branch of

Project Name	No of `for`'s	No function call		Total	
Berkley DB	1799	18	1,00 %	1371	76,21 %
Google App Engine	967	19	1,96 %	694	71,77 %
Hadoop	1898	36	1,90 %	1448	76,29 %
Hibernate	855	9	1,05 %	731	85,50 %
jEdit	894	30	3,36 %	614	68,68 %
Tomcat	1510	106	7,02 %	1123	74,37 %
Total	7923	218	2,75 %	5981	75,49 %

Table 5.10: Discoveries of `for` loops with only safe method invocations.

a conjunction expression matches the current case but the other branch matches one of the previous cases. But this is a very rare situation. The column *No function* contains number of newly covered loops. A group of XPath expressions made it possible to cover additional 2,75% of cases, thus we are able to generate *decreases* formula for 75,49% of numeric `for` loops.

Expression with private field access
In this section cases are taken into account, where a private field is used in the loop condition. Since the field is private, it cannot be directly accessed by any subclass. When a single thread execution is assumed, in some cases generating proper *decreases* formula is possible. These cases are:

- A private field is assigned directly in the field declaration and never updated. Since the field is private, it cannot be accessed by a subclass, therefore it is not changed during the execution. Such field probably should be declared as final. An example of the situation is presented in Fig. 5.27.

```
public class MBeanUtils {
  ...
  private static String exceptions[][] = ...;
  ...
  for (int i=0; i < exceptions.length; i++) {
    if (className.equals(exceptions[i][0])) {
      return (exceptions[i][1]);
    }
  }
  ...
}
```

Figure 5.27: Example of a `for` loop with a private field in loop condition. The field is assigned only in the declaration. Taken from Tomcat project (from file `org.apache.catalina.mbeans.MBeanUtils.java`).

- A private field is assigned only in a constructor or in the field declaration, and `for` loop, in which it is used, appears outside a constructor. Since the loop is outside a constructor, the value of the private field does not change during the execution of the loop. An example of such situation is presented in Fig. 5.28.

```
public class SQLQueryReturnProcessor {
  ...
  private NativeSQLQueryReturn[] queryReturns;
  ...
  public SQLQueryReturnProcessor(NativeSQLQueryReturn[] queryReturns, ...) {
        this.queryReturns = queryReturns;
        ...
  }
  public ResultAliasContext process() {
    ...
    for (int i=0; i < queryReturns.length; i++) {
      processReturn(queryReturns[i]);
    }
    ...
  }
  ...
}
```

Figure 5.28: Example of a `for` loop with a private field in loop condition. The field is assigned only in a constructor and the loop appears outside a constructor. Taken from Hibernate project (from file `org.hibernate.loader.custom.sql.SQLQueryReturnProcessor.java`).

For the first case, the most important is to find out if the *Name* element that appears in the expression is a private field that is assigned only in the declaration. The XPath fragment responsible for such check is as follows:

```
1  Name[@private][@finalSuffix or @numeric or @type="String"][@name =
2    ancestor::TypeDeclaration/FieldDeclaration
3        [VariableDeclarationFragment/Name[@private]]
4    [not(VariableDeclarationFragment/Name/@name =
5                  parent::*//LeftOfAssignment/Name/@name)]
6    /VariableDeclarationFragment/Name/@name]))]
```

Line 1 narrows down the search to only private fields that are either numeric or have a final suffix (for example, the expression `x.y`, where `x` is private and `y` is not final is not interesting). Lines 2-6 are responsible for the search for a *TypeDeclaration* ancestor (it represents a class declaration), which has a field element that has the same name as the current *Name* element. First, the condition in lines 2-3 ensures that there exists a *TypeDeclaration* ancestor that has a private field declaration. Next, lines 4-5 guarantee that the name of the field does not appear anywhere in the class on the left hand side of an assignment. Line 6 matches the name from line 1 with the name from the declaration. The full XPath expression for the case is a slight modification of the one from Fig. 5.23 with the name filter element included and removed constraint from line 20. Analogically, there is an extended case, where updates of the control variable appears only in the body of the loop.

The case of private field that is assigned in constructors and the `for` loop is used outside a constructor is a little more complex. Similarly to the private field assigned only in the declaration, the fragment responsible for counting descendant elements in the loop condition element is modified. One has to ensure that the *Name* element appears as the field in the class and is not assigned outside a constructor. The check is as follows:

```
1  Name[@private][@finalSuffix or @numeric or @type="String"][@name=
2    ancestor::TypeDeclaration/FieldDeclaration
3        [VariableDeclarationFragment/Name[@private]]
4      [not(VariableDeclarationFragment/Name/@name = parent::*//LeftOfAssignment
5        [not(ancestor::MethodDeclaration/@constructor)]/Name/@name)]
6      [VariableDeclarationFragment/Name/@name = parent::*//LeftOfAssignment
7        [ancestor::MethodDeclaration/@constructor]/Name/@name]
8    /VariableDeclarationFragment/Name/@name]))]
```

Analogically to the previous case, the `name` attribute of the *Name* element is matched with a field declaration, which has a number of filters. Lines 4-5 guarantee that the field is not assigned in methods, which are not constructors. Lines 6-7 ensure that the field is actually assigned in some constructor method. Finally, line 8 finishes the match of the name from line 1 with the field name.

Project Name	**No of `for`'s**	**Declaration only**		**Constructor**		**Total**	
Berkley DB	1799	41	2,28 %	14	0,78 %	1426	79,27 %
Google App Eng.	967	0	0,00 %	1	0,10 %	695	71,87 %
Hadoop	1898	14	0,74 %	37	1,95 %	1499	78,98 %
Hibernate	855	0	0,00 %	3	0,35 %	734	85,85 %
jEdit	894	0	0,00 %	5	0,56 %	619	69,24 %
Tomcat	1510	29	1,92 %	22	1,46 %	1174	77,75 %
Total	7923	84	1,06 %	82	1,03 %	6147	77,58 %

Table 5.11: Discoveries of `for` loops with private field in the loop condition.

Statistics of the findings are presented in Table 5.11. XPath expressions for conjunction cases are also created. A new group of XPath expressions made it possible to cover additional 2.09% of cases, thus total coverage is 77,58%.

Some strange cases
Here a few special cases of `for` loops that can be found in analysed projects are described. They are not very frequent, though they helped to break the barrier of 78%. XPath expressions for these situations are not presented, because they are quite simple, hence not very interesting. These cases are:

Multiplication in the update expression: when a multiplication by a constant is done in the update expression of a positive variable, one can easily generate a *decreases* formula. An example of such situation is presented in Fig. 5.29.

```
//@ decreases WritingServlet.EXPECTED_CONTENT_LENGTH*10 - i;
for (int i=1; i <= WritingServlet.EXPECTED_CONTENT_LENGTH; i*=10) {
  WritingServlet servlet=new WritingServlet(i);
  Tomcat.addServlet(root,"servlet" + i,servlet);
  root.addServletMapping("/servlet" + i,"servlet" + i);
}
```

Figure 5.29: Example of a `for` loop with multiplication in the update expression, taken from Tomcat project (from file `org.apache.catalina.connector.TestOutputBuffer.java`).

Increase the control variable in the loop condition: such situation is possible because of an existence of operators ++ and --. An example of the case together with the generated *decreases* formula is presented in Fig. 5.30.

```
//@ decreases i;
for (int i=MAX_CODE_LEN; --i > 0; ) {
  base[i]=0;
  limit[i]=0;
}
```

Figure 5.30: Example of a `for` loop with update in condition itself, taken from Hadoop project (from file `org.apache.hadoop.io.compress.bzip2.CBZip2InputStream.java`).

In the XPath expression for the case, it is assumed that the update expression is empty, the condition has unary expression with the control variable on one side of the comparison and a final expression on the other side of it.

Increase the control variable by a random integer: in the standard Java library there is a `Random` class that can be used to generate random numbers. According to the documentation of Java[17], the method:

```
public int nextInt(int n)
```

[17]For details see `http://docs.oracle.com/javase/6/docs/api/java/util/Random.html`

generates an integer between 0 (inclusive) and the specified value `n` (exclusive). Therefore, when the result of the method, but increased by a positive integer, is used in the update expression of a loop, then one is able to generate a proper *decreases* formula. An example is presented in Fig. 5.31.

```
private static final int MAX_LENGTH = 15000;
...
//@ decreases Integer.MAX_VALUE + MAX_LENGTH - i;
for (int length=0; length < MAX_LENGTH;
     length += random.nextInt(MAX_LENGTH / 10) + 1) {
  LOG.info("******Number of records: " + length);
  createSequenceFile(length);
  LOG.info("Accepted " + countRecords(0) + " records");
}
```

Figure 5.31: Example of a `for` loop, where the control variable is increased by a positive integer in the update condition, taken from Hadoop project (from file `org.apache.hadoop.mapred.TestSequence-FileInputFilter.java`).

Since `Random` class is not `final`, created XPath expressions need to ensure that the object, which is used in the update expression, is in fact an instance of the `Random` class, not a subclass. XPath expressions created for the case assure that the variable is of `Random` class. The variable may be a `final` field, which has new `Random` object assigned in the field definition fragment. It may also be a local variable that is initialised in the method, where the `for` loop appears. In such situation it is assured that the variable has a new `Random` object instance assigned in the method. The last case is when the variable is a private field, which is not `final`, but is assigned only in the field definition fragment. For the loop condition a "safe" private field approach is used, where expressions build of constants, local variables, final variables, and private fields that are assigned only in the field definition fragment are allowed.

Use of `Math.min` function in the loop condition: there are situations, where the function is used in the loop condition, but one of the parameters does not change. The function:

```
public static int min(int a, int b)
```

is available in a `final` class `Math` and returns a smaller of the two arguments. When one of the arguments does not change during the execution of the loop, i.e. it meets the expression with "safe" private field property, it is possible to generate the *decreases* formula. An example together with the generated *decreases* formula is presented in Fig. 5.32.

```
static final private int NUM_OF_PATHS = 4;
...
//@ decreases NUM_OF_PATHS - i;
for (int i=0; i < Math.min(NUM_OF_PATHS,files.length); i++) {
  path[i]=new Path(files[i]).makeQualified(fs);
  if (!fs.mkdirs(path[i])) {
    throw new IOException("Mkdirs failed to create "+path[i].toString());
  }
}
```

Figure 5.32: Example of a `for` loop, where `Math.min` appears in the loop condition, taken from Hadoop project (from file `org.apache.hadoop.fs.TestGlobPaths.java`).

Project Name	No of `for`'s	Specific cases		Total	
Berkley DB	1799	0	0,00 %	1426	79,27 %
Google App Engine	967	0	0,00 %	695	71,87 %
Hadoop	1898	49	2,58 %	1548	81,56 %
Hibernate	855	0	0,00 %	734	85,85 %
jEdit	894	0	0,00 %	621	69,46 %
Tomcat	1510	6	0,40 %	1180	78,15 %
Total	7923	55	0,69 %	6204	78,30 %

Table 5.12: Discoveries of `for` loops for some specific cases.

Statistics of the findings for all the cases mentioned are presented in Table 5.12. With these cases the boundary of 78% is broken and final coverage 78,30% of numeric `for` loops is achieved.

5.5.3 Final Result

In the current section, an experiment was described, in which *decreases* formula were generated for over 78% of numeric `for` loops in the analysed projects. Note that all the above-described categories were expressed by purely syntactic criteria. A summary of the discoveries is presented in Fig. 5.33. Most of the uncovered cases were using method calls and were iterations over Java collections like `List`. These cases are very hard or even impossible to cover by syntactic only criteria. The problem is that usually one cannot discover what is the actual class of the collection. It might be one of the standard Java collections but it may be user's own implementation as well. Therefore, we may not have any information about possible side effects of method calls on such object.

In order to verify that the obtained results also apply to other projects, the method was checked against a fresh set of applications. Here, also an arbitrary set of very

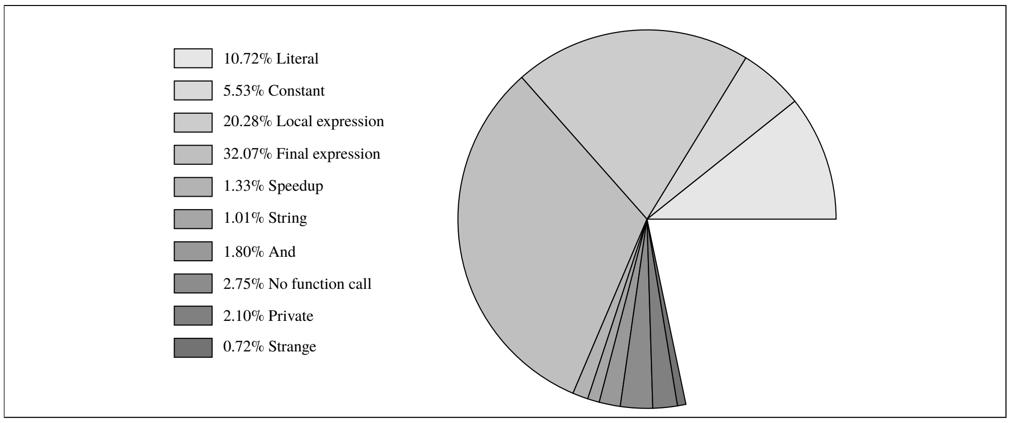


Figure 5.33: Contribution of each loop type.

popular and quite large projects was chosen. These project come from various sources: Sourceforge project repository, Eclipse open-source projects repository and one is commercial. These projects are:

AspectJ[18] is a seamless aspect-oriented extension to the Java programming language that enables clean modularisation of the ’crosscutting concerns’.

Spring[19] is an application framework for Java platform. It includes Inversion of Control container that manages object lifecycles.

Vuze[20] (previously named Azureus) is an application to exchange and distribute data over the Internet. One of the most popular projects in Sourceforge repository[21]. It is also one of the most popular BitTorrent clients.

Details of the projects used for verification of the results along with statistics of `for` loops coverage are presented in Fig. 5.34. For over 78% of numeric `for` loops *decreases* formula was generated, which is consistent with the original findings.

5.5.4 Bad Loops Found

One of the outcomes of the loop termination experiment was that a number of non-trivial loops was found in the analysed projects. They were sometimes written improperly, sometimes could be simplified, may contain bugs, or cause errors after

[18]For details see `http://www.eclipse.org/aspectj/`

[19]For details see `http://www.springsource.org/`

[20]For details see `http://www.vuze.com`

[21]3rd place on May 22, 2012, for details see `http://sourceforge.net/top/`

Project Name	**Version**	**Size in KLoC**	**No. of `for`'s**	**`for`'s per 1000 lines**	**Coverage**	
AspectJ	1.6.12	385	5049	13,11	4461	88,35 %
Spring toolkit	3.1.1	181	457	2,52	349	76,37 %
Vuze	4.7.0.2	496	3689	7,44	2395	64,92 %
Total		1062	9195	8,66	7205	78,36 %

Figure 5.34: Details of projects used for verification and statistics of covered loops (sizes are given in Kilo Lines of Code, only interesting `for` loops are considered).

the maintenance. *CodeStatistics* tool provides them (their code, location in the project etc.), so that a programmer can handle them manually. Using the output generated by *CodeStatistics* it was quite easy to select examples for this section. They were taken from the code snippets that were marked as ”not classified” and were short enough to be included in the thesis.

First, let us consider the example from Fig. 5.35. Note that the control variable is increased in the update expression of the loop, but it may be decreased in the inner `if` condition. In some iterations the value of the control variable may not change. Therefore, termination of the loop is not clear and requires thorough analysis.

```
for (int i=0; i < markers.size(); i++) {
  Marker marker=markers.get(i);
  if (getLineOfOffset(marker.getPosition()) == line) {
    setFlag(MARKERS_CHANGED,true);
    marker.removePosition();
    markers.removeElementAt(i);
    i--;
  }
}
```

Figure 5.35: Bad loop example, taken from JEdit (from file `org.gjt.sp.jedit.Buffer.java`).

Example from Fig. 5.36 is similar to the previous one. Here, further analysis makes it possible to discover that the loop terminates, because the value of `count` is decreased inside the `removeHeader(j--)` function call.

In the code fragment from Fig. 5.37 it is not obvious why the loop should be terminating. At first, it seems correct but one can notice that the inequality direction in the termination condition is against intuition. The loop will terminate only if `nSamples - nRemove` gets smaller than `i`, but this can be done only inside the `deltas.remove` method.

In all the examples presented in this subsection, it is not obvious why loops are

```
for (int i=0; i < count; i++) {
  if (headers[i].getName().equalsIgnoreCase(name)) {
    removeHeader(i--);
  }
}
```

Figure 5.36: Bad loop example, taken from Tomcat (from file `org.apache.tomcat.util.http.MimeHeaders.java`).

```
for (int i=nSamples - 1; i < nSamples - nRemove; i--) {
  deltas.remove(i);
}
```

Figure 5.37: Bad loop example, taken from Vuze (from file `com.aelitis.azureus.core.networkmanager.admin.impl.NetworkAdminSpeedTesterBTImpl.java`).

eventually terminating. It should be at least documented why they are working correctly. It should be explained in the loop and in the place, which is responsible for the termination. Finding such error prone code fragments is crucial also for maintenance.

5.5.5 Verification

In this experiment ESC/Java2 tool was used to verify loop termination property of `for` loops enriched with *decreases* annotations inserted by *CodeStatistics* tool. ESC/Java2 provides `-LoopSafe` option that can be used to verify these annotations. Since ESC/Java2 accepts only Java 1.4 source code and selected projects use features introduced in Java 1.5 (e.g. generics and enums), the source code had to be adapted to the older Java version. This was applied to jEdit project. Unfortunately, ESC/Java2 did not manage to verify all files containing for loops — there were some fatal errors caused by missing or invalid standard library specifications (most of them caused by the `java.awt` package). All files that were successfully processed by ESC/Java2 did not return any errors or warnings concerning *decreases* formulae.

5.6 Summary

In this chapter a simple approach for creating verification methods was presented. Additionally, *CodeStatistics* tool that supports this methodology was described. The approach was applied in an experiment — simple loop termination prover for Java `for` loops. The results obtained show that basic verification techniques can be successfully used in real, production code. Tools similar to *CodeStatistics* can be really helpful in the Quality Assurance process. Code reviewers can pay less attention to simple cases, hence they can focus on difficult aspects that are usually more error prone. Such methods can even guarantee correctness of some aspects of the code (e.g. that `for` loops terminate). They can be used as a first step in introducing formal methods into business applications.

Chapter 6

Extended Pattern Discovery in Evaluation of Abstract Domains

6.1 Introduction

In this chapter an engine is presented that can be used to evaluate abstract domains. First, in Section 6.2, *JavaAI* is introduced — an abstract interpreter for Java language that was implemented as a part of this study. Next, in Section 6.3, a generic extension mechanism for *CodeStatistics* is described that makes it possible to write plugins that perform additional code analysis and extend the query language used by *CodeStatistics* with user-defined functions. The extension was also implemented as a part of this study. In Section 6.4, an adaptation of *JavaAI* as the analysis plugin for *CodeStatistics* is presented. Finally, in Section 6.5, an experiment is described that uses the *JavaAI* analysis plugin for *CodeStatistics* to evaluate abstract domain implementations.

6.2 *JavaAI* — an Abstract Interpreter for Java

In this section an implementation of a static analyser for Java language is presented — *JavaAI*[1]. The analyser works on Java source code. It uses the *abstract interpretation* technique as described in Section 2.4.6. In *JavaAI* the following abstract domains are implemented:

- *Bool*: a non-relational abstract domain of boolean values;

[1]Available at the companion disk and `http://www.mimuw.edu.pl/~kjk/phd`

- *Intv*: a non relational abstract domain of intervals;
- *Bool* × *Intv*: a product domain of booleans and intervals;
- *Boxes*: an abstract domain of *boxes* — together with a widening operator that has configurable *widening sequence thresholds*;
- *Bool* × *Boxes*: product domain of booleans and boxes.

First, in Section 6.2.1 some interesting details about the semantics and syntax of Java are presented. Next, in Section 6.2.3 the assumptions made in the implementation of *JavaAI* are described. In Section 6.2.4 the control flow graph used in *JavaAI* is presented, and finally, Section 6.2.5 contains a short review of tests written for the implementation.

6.2.1 The Java Programming Language

The Java language [57] is much more complex than the language *Simple* that was introduced to present the abstract interpretation framework and the domain of *boxes*. Java is a class-based and object-oriented programming language. Every statement in Java is attached to some class: it may be either a part of a method or in an initializer block.

Types
There are two kinds of types in Java:

- *Primitive types*: boolean type (`boolean`), integer types (`byte`, `short`, `int`, `long` and `char`) and floating point types (`float` and `double`). There are various conversions between these primitive types (details can be found in [57, Chapter 5]).
- *Reference types*: this includes class types, interface types, type variables and array types [57, Section 4.3].

Boxing and Unboxing
The feature was introduced in Java version 5. It simplifies usage of primitive types in situations when objects (reference types) are required, e.q. collections. For every primitive type there is a corresponding reference type, e.g. for `boolean` class `Boolean`. *Boxing* is the operation of converting a primitive type value into a value of the corresponding reference type (a wrapper). *Unboxing* is the reverse operation — it extracts a primitive value from the reference type. Both operations are implicit. For example, the expression `true && new Boolean(false)` is a correct Java code (`&&` is a boolean conjunction). In the expression, when executed, the second argument is automatically converted to the primitive value `false`.

Statements
In the Java language, a method or an initializer block is built from statements. Both statements or statement blocks may be labelled. Here we list the most interesting statement types:

- conditional `if` statement,
- `switch` statement,
- three loop statements: `while`, `do-while`, and standard `for` loop,
- enhanced `for` loop statement, which simplifies iteration over arrays and built-in collections (this is just a syntactic sugar and is translated to a regular `for` loop, see [57, Section 14.14]),
- `break` statement that breaks out the innermost enclosing loop or `switch` statement, there is a also a `break <label>` version of the statement, which causes break out of the block/statement with the label `<label>`,
- `continue` statement that stops the current loop iteration and jumps straight to the next one, (also, label is allowed here: in such case a control jumps to the next iteration of the loop with the chosen label),
- `return` statement that is used to end a method and return a value (the if result type is not `void`),
- `try-catch-finally` statement is used to handle exceptions,
- `throw` statement to throw an exception and end a block or method execution,
- `synchronized` statement that is used to control concurrency,
- `assert` statement used to make assertions in the source code.

6.2.2 Expression Evaluation

At first, we have to note that expression evaluation may cause side-effects, e.g. `(c = 17) > 3` is a boolean expression that evaluates to `true` and as a side-effect it sets the value of the integer variable `c` to `17`. The general rule of expression evaluation in Java is that operands are evaluated before the operation, for example:

- In case of a method call, arguments are evaluated from left to right first and then the call is executed.

- In case of a prefix or a postfix expression, the operand is evaluated first and then the operator is applied.
- In case of a binary expressions, operands are calculated first and then the operator is applied.

It may happen than an exception occurs when some operand is evaluated (e.g. division by 0 or an exception in some method call). When that happens, other operands are not evaluated.

There is one exception from the general rule of expression evaluation in Java, i.e. boolean expressions are evaluated in a lazy fashion. That means that when left argument of a conjunction evaluates to `false`, the right argument is not evaluated (therefore, there are no side-effects from that evaluation). Similarly, when an alternative is calculated and left argument evaluates to `true`, the right one is not evaluated.

Here we explain the semantics of boolean expressions in *JavaAI* in detail. Let $\mathit{BExp}_{\mathit{Java}}$ be the set of Java boolean expressions and $\langle \leq_{\mathcal{D}}, \mathcal{D} \rangle$ be an abstract domain. The abstract boolean semantic function has type:

$$\mathcal{B}_{\mathit{Java}} : \mathit{BExp}_{\mathit{Java}} \to (\mathcal{D} \to \mathcal{D} \times \mathcal{D}).$$

The abstract transfer function for some boolean expression calculates simultaneously the result of both positive and negative evaluation of that boolean expression. The implementation of the lazy conjunction is as follows:

$$\frac{\mathcal{B}_{\mathit{Java}}[\![a]\!]\, D = \langle D'_T, D'_F \rangle \qquad \mathcal{B}_{\mathit{Java}}[\![b]\!]\, D'_T = \langle D''_T, D''_F \rangle}{\mathcal{B}_{\mathit{Java}}[\![a \;\texttt{\&\&}\; b]\!]\, D = \langle D''_T, D'_F \cup_{\mathcal{D}} D''_F \rangle},$$

and similarly for the alternative:

$$\frac{\mathcal{B}_{\mathit{Java}}[\![a]\!]\, D = \langle D'_T, D'_F \rangle \qquad \mathcal{B}_{\mathit{Java}}[\![b]\!]\, D'_F = \langle D''_T, D''_F \rangle}{\mathcal{B}_{\mathit{Java}}[\![a \;\texttt{||}\; b]\!]\, D = \langle D'_T \cup_{\mathcal{D}} D''_T, D''_F \rangle}.$$

The implementation of negation is pretty straightforward:

$$\frac{\mathcal{B}_{\mathit{Java}}[\![a]\!]\, D = \langle D_T, D_F \rangle}{\mathcal{B}_{\mathit{Java}}[\![\texttt{!}a]\!]\, D = \langle D_F, D_T \rangle}$$

as well as evaluation of boolean constants:

$$\mathcal{B}_{\mathit{Java}}[\![\texttt{true}]\!]\, D = \langle D, \perp_{\mathcal{D}} \rangle, \qquad \mathcal{B}_{\mathit{Java}}[\![\texttt{false}]\!]\, D = \langle \perp_{\mathcal{D}}, D \rangle.$$

The evaluation of atomic boolean expressions, such as boolean variable, boolean field access, boolean method call, or an arithmetic comparison depends on the implementation of the domain itself. Every domain that is implemented in *JavaAI* has to provide implementation of such transfer functions.

6.2.3 Implementation Assumptions

The abstract interpreter *JavaAI* focuses only on primitive Java types, it performs intra-method analysis on local variables. The list of the assumptions made is as follows:

- Only primitive types are interesting. All domains that are implemented handle only `boolean` or integer values (all integer types).
- Every method or an initializer block is analysed separately.
- During the analysis of a method or an initializer block we do not go inside any other method that is defined inside (e.g. a part of inner class definition or anonymous class). In Java language, only the method, where a local variable is defined, can modify the variable. In particular, when a class is defined inside a method (also anonymous class) and performs an access to a variable of the method, the variable must be declared as `final`.
- When a primitive value comes from unboxing, method call, field access, or array access we assume that we know nothing about the value.
- No exception analysis is performed: at the entry to `catch` or `finally` block we assume that anything might have happened (the state is cleared). Also we do not analyse, where to a `throw` might lead.
- We assume that we analyse production code — `assert` expression content is ignored.

Besides the assumptions listed above, we have to note that we highly depend on the variable and type binding provided by the Eclipse JDT. Therefore, if the project's classpath settings are incorrect or incomplete, the analysis may give not accurate results, i.e. some control points may not have the domain value calculated at all or the result that is calculated may be imprecise. Regular *CodeStatistics* does not have this limitation since it uses only syntactic matching.

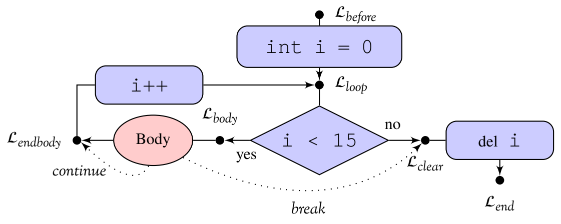
6.2.4 Control Flow Graph

In order to create an abstract interpreter, one needs to build a control flow graph for Java programs. Since in *JavaAI* every method or initializer block is analysed separately, a separate control flow graph is created for every such item. We call such graph a *method control flow graph*.

The *method control flow graph* is constructed by traversing the *abstract syntax tree* of the method or the initializer and every time a statement is encountered, a specific graph fragment for that statement is built. An example of a control flow graph fragment for Java `for` statement is presented in Fig. 6.1. The first step is to

```
for(int i=0; i < 15; i++) {
   //Body
}
```

(a) Source code



(b) Control flow graph

Figure 6.1: Fragment of a *method control flow graph* for the Java `for` statement. Optional `break` and `continue` edges from loop body are present.

create consequent edges representing `for` loop initialization statements (e.g. `int i = 0`), where new variables may be introduced. The last initialization edge leads to the loop vertex, labelled $\mathcal{L}_{loop}$. Next, a conditional edge is created. It has two output edges:

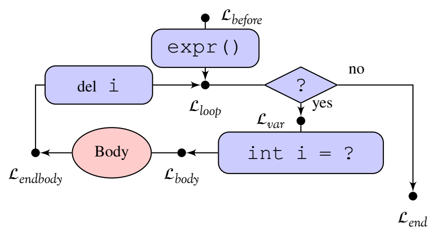
- positive, that leads to the graph fragment for the loop body, and later through the `for` loop update statements back to the loop vertex,
- negative, that goes through the variable removal edge, where variables introduced in `for` loop initialization statements are removed, to the node after the loop (labelled $\mathcal{L}_{end}$).

Specific Modifications

In a few cases, implementation assumptions from Section 6.2.3 are applied that simplify the process of creation of the control flow graph. A first example is the Java enhanced `for` statement. This is just a syntactic sugar — every enhanced `for` can be translated into a regular `for` [57, Section 14.14.2]. The value of the new variable that is introduced in the enhanced `for` may come either from the array access (if we iterate over array items) or from a method call possibly connected with unboxing (if we iterate over a elements of a collection it is a result of a `next` method call). Since we have assumed that we know nothing about such variables, we simplify the construction of *method control flow graph*. An example of a fragment of the *method control flow graph* for the enhanced `for` statement is presented in Fig. 6.2. The first simplification is that the loop test is a “dummy” test that assumes we know nothing. The second simplification is that the variable is not stored

```
for(int i: expr()) {
    //Body
}
```

(a) Source code



(b) Control flow graph

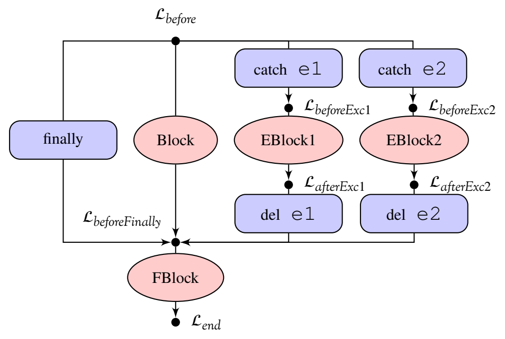
Figure 6.2: A simplified fragment of *method control flow graph* for the Java enhanced `for` statement.

between loop iterations: the introduction of the new variable is moved to the location just before the loop body and the variable is removed just after the loop body. Additionally, the variable is introduced with unknown value.

Another simplification of the *method control flow graph* construction is done for the *try-catch-finally* block. Since we do not handle exceptions at all, we do not analyse when a specific exception may be thrown. In Fig. 6.3 a fragment of the *method control flow graph* that is created for the `try-catch-finally` statement is presented.

```
try {
        // Block
} catch {Exception1 e1} {
        // EBlock1
} catch {Exception1 e2} {
        // EBlock1
} finally {
        //FBlock
}
```

(a) Source code



$\mathcal{L}_{before}$
catch e1
catch e2
$\mathcal{L}_{beforeExc1}$
$\mathcal{L}_{beforeExc2}$
finally
Block
EBlock1
EBlock2
$\mathcal{L}_{afterExc1}$
$\mathcal{L}_{afterExc2}$
$\mathcal{L}_{beforeFinally}$
del e1
del e2
FBlock
$\mathcal{L}_{end}$

(b) Control flow graph

Figure 6.3: A simplified fragment of *method control flow graph* for the Java `try-catch-finally` statement.

For every `catch` statement we create a graph fragment that introduces a new exception variable and after the `catch` block the variable is removed. There is also a `finally` edge that leads from the input edge to the vertex directly before finally block (it bypasses the block). This edge represents an uncaught exception. Every abstract domain in *JavaAI* implements the abstract transfer function in such fashion, that the `finally` edge as well as any `catch` edge clears the knowledge (i.e. it should return the top element of the domain). This way, it is not necessary to create edges to `catch` and `finally` from every instruction of the block.

Variable Scope

In the construction of the control flow graph the scope of variables is taken into consideration: some of the edges introduce new variables and there are special edges that remove them. New variables may be introduced in two situations: at the beginning of a method (method arguments) and in every block with explicit variable declaration statement. Information that a variable is removed is attached to every edge that goes to the target that is outside the scope of the variable, e.g. to the edge that goes out of the variable declaration block. When attaching information about variable removal, besides the regular end of a block, edges corresponding to `break`, `continue`, and `return` statements are considered. It holds that for every vertex in the *method control flow graph* all paths, which lead to the vertex, bring the same set of variables (when one keeps track of the variable set: add and remove them when going through mentioned edges).

6.2.5 Testing of *JavaAI*

The *JavaAI*, the abstract interpreter for Java, is implemented in Java and it works as the Eclipse plugin. The implementation of various aspects of the interpreter were thoroughly tested with unit tests. In the application of unit tests, Eclipse is executed with a special workspace that contains a test project with all the test files. The following aspects of the interpreter are tested:

Method control flow graph — a special text format is used to describe the graph of a method. The description is stored with method's Javadoc, and automatically, read by the test engine and compared with the graph that is built for the method.

Expression evaluation — analogically to the previous case, a special text format is used to describe domain state in vertices of the graph. This is also stored in method's Javadoc. Multiple tests for evaluation of boolean expressions (including lazy evaluation) and numeric expressions were created. The main focus was on testing whether the interpreter correctly implements expression evaluation principles of the Java language.

Statement evaluation — simple tests were created in order to verify if abstract evaluation of statements works correctly.

Domain tests — tests for intervals, booleans, non-relational domain of booleans, non-relational domain of intervals, and the domain of boxes were created. They check if the domain operations, i.e. join, meet, widening, comparison, are properly implemented.

About 240 test cases were created, where some include multiple smaller tests. The coverage obtained by these tests is over 75% of the code.

6.3 Extending *CodeStatistics*

The *CodeStatistics* tool was extended in order to make it possible to extend the default XPath language so that user-defined functions can be used. The extension is very generic. The Eclipse plugin engine mechanism, called *Extension points*, was employed for the purpose. It makes it possible to extend an Eclipse plugin by another Eclispe plugin. With this mechanism one can add new functions according to the needs of a specific task.

CodeStatistics provides a `codeStatistics.analysisPlugin` extension point for *analysis plugins*. An implementation of the extension provides:

- Analysis algorithm that can be applied on a piece of code and return a map from AST nodes to analysis results. An interface of the plugin is presented in Fig. 6.4.

```
public interface IAnalysisPlugin {
    public String getName();
    public void initialiseAnalysis();
    public List<IAnalysisFunction> getExtensionFunctions();
    public Map<ASTNode, Object> analyse(CompilationUnit cu);
    public String getSummary();
}
```

Figure 6.4: Interface of the *CodeStatistics* anlalysis plugin.

- A number of extension functions that can be used in XPath queries, e.g. ask questions about the analysis result. An interface of the analysis plugin extension function is presented in Fig. 6.5.

  The implementation of extension functions is a little bit more complex. In this case, the programmer must define types of arguments and the result of the function. These types are verified at runtime, when XPath is evaluated.

When Eclipse starts, *CodeStatistics* looks for extensions that are analysis plugins, and then loads them in order to extend the XPath language by functions provided

```
public interface IAnalysisFunction {
    public String getName();
    public Object execute(Object[] args, Map<ASTNode, Object> context);
    public Class<?>[] getArgumentType();
    public Class<?> getResultType();
}
```

Figure 6.5: Interface of the *CodeStatistics* analysis plugin extension function.

by the plugins. *CodeStatistics* itself adds one XPath function: `cs:ast(<elem>)` that returns AST node for the XML elements. Extension functions may work on these nodes.

6.4 *JavaAI* Analysis Extension for *CodeStatistics*

JavaAI implementation provides an analysis plugin implementation for *CodeStatistics*. The plugin executes the *JavaAI* analyser on all methods and initializer blocks of the source file. The analysis with multiple domains may be executed and results compared. The execution of the analysis plugin should return a mapping between AST node and analysis result objects, which in case of *JavaAI* are domain values. In the implementation of the analysis plugin, the translation between AST nodes and control points, for which domain values are computed, is simplified. Not every AST node has a corresponding value and some nodes may have multiple values:

- For every statement in the code we actually have two values: one from the control point just before the statement, and another one from the point just after the statement.
- For every loop statement we attach the loop invariant value. This is done in order to unify access to these values, otherwise for every loop kind we would have to look for the invariant differently.
- AST nodes that are not statements do not have any value attached.

6.4.1 XPath Extension Functions

The analysis plugin provides XPath functions to access various abstract domain values as well as functions to evaluate them (e.g. to check whether a value is non-trivial — not the top element in the domain) or compare concrete values corresponding to abstract ones. The plugin provides the following extension functions:

- `cs:isTop(<domain-value)` that returns true if the domain value is top.

- `cs:lt(<domain-value>, <domain-value>)` is a function that can be used to compare concrete domain values corresponding to the abstract ones. It returns true if the concrete value corresponding to the first argument is strictly less (by inclusion) than the concrete value corresponding to the second argument.

- `cs:incomparable(<domain-value>, <domain-value>)` is similar to the previous function. The difference is that it returns true if the concrete value corresponding to the first argument is not directly comparable with the concrete value corresponding to the second argument.

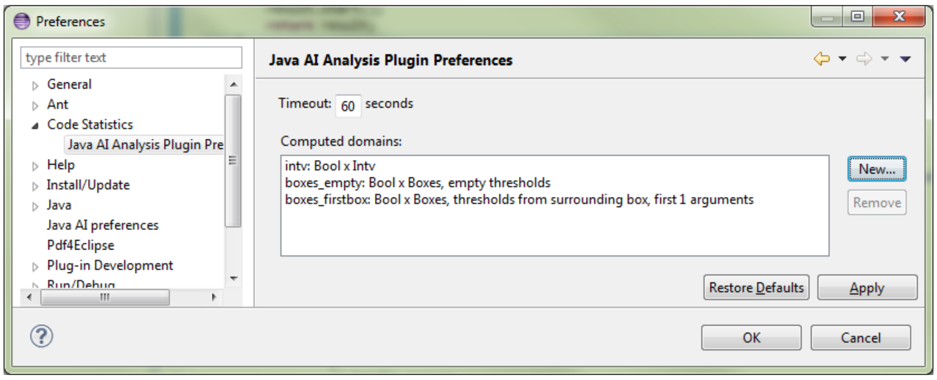
- `cs:dom_before(<domain-name>, <ast-node>)` is a function that returns the value of an abstract domain with given name in the control point before the AST node in the parameter, or empty sequence if there is no value.

- `cs:dom_after(<domain-name>, <ast-node>)` is similar to the previous one, but returns the value of domain in the control point after the AST node in the parameter.

- `cs:dom_invariant(<domain-name>, <ast-node>)` is a function that returns the loop invariant for the domain with given name for the loop node in the parameter, or empty sequence if the parameter is not a loop node.

6.4.2 *JavaAI* Analysis Plugin Configuration

In *JavaAI* analysis plugin, the user may define a number of abstract domains to use in the analysis. To every such domain, the user has to assign a unique name that is used to access domain values in extension functions. An example configuration



Timeout: 60 seconds

Computed domains:

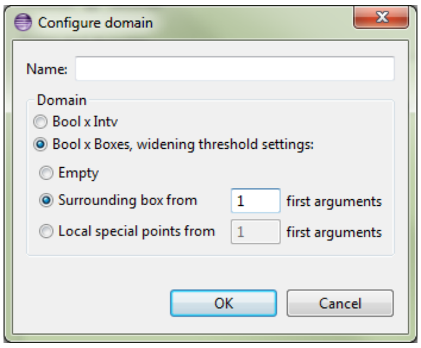
intv: Bool x Intv
boxes_empty: Bool x Boxes, empty thresholds
boxes_firstbox: Bool x Boxes, thresholds from surrounding box, first 1 arguments

Figure 6.6: *JavaAI* analysis plugin configuration.

of domains is presented in Fig. 6.6. In the screen, there are three abstract domains configured:

- $Bool \times Intv$ with the name *intv*,
- $Bool \times Boxes$ with empty thresholds sequence — named *boxes_empty*,
- $Bool \times Boxes$ with simple thresholds sequence created from the box surrounding the first argument in the widening sequence — named *boxes_first-box*.

The domain configuration dialog, which is shown when the user defines new domain, is presented in Fig. 6.7. Currently, only two domains are provided: $Bool{\times}Intv$ and $Bool \times Boxes$, which has three strategies for widening thresholds available.


Configure domain

Name:

Domain
( ) Bool x Intv
(●) Bool x Boxes, widening threshold settings:
( ) Empty
(●) Surrounding box from 1 first arguments
( ) Local special points from 1 first arguments

Figure 6.7: *JavaAI* analysis plugin domain configuration.

The analysis plugin executes *JavaAI* abstract interpreter separately for every domain that is configured, therefore it may affect the waiting time experienced by the user.

Note that there is also a *timeout* setting available (in the top of the preferences window in Fig. 6.6) for the *JavaAI* analysis plugin. The purpose of this setting is to stop the analysis of a single method or initializer block when it is taking longer than the timeout value.

6.4.3 Example *CodeStatistics* XPath Configuration File

Here, an example of XPath configuration file is presented that uses the extension functions provided by *JavaAI* analysis plugin for *CodeStatistics*. We consider *JavaAI* plugin configuration as presented in Fig. 6.6. An example XPath configuration file for *CodeStatistics* is presented in Fig. 6.8.

```
<descriptions>
  <all name="AllLoops" xpath="//*[@isLoop]" />
  <description name="nonTrivialIntv"
    xpath='//*[@isLoop][cs:dom_invariant("intv", cs:ast(.))]
                 [not(cs:isTop(cs:dom_invariant("intv", cs:ast(.))))]'/>
  <description name="nonTrivialFirstBox"
    xpath='//*[@isLoop][cs:dom_invariant("boxes_firstbox", cs:ast(.))]
                 [not(cs:isTop(cs:dom_invariant("boxes_firstbox", cs:ast(.))))]'/>
</descriptions>
```

Figure 6.8: Example of an XPath configuration of the *CodeStatistics*.

In the example, we focus on loop invariants, therefore we search for elements that are loops. The category `nonTrivialIntv` is used to find all non-trivial invariants for the *Bool*×*Intv* domain, which is named *intv* in the *JavaAI* plugin configuration. The XPath expression test for the category applies the following filters for the node:

- `[@isLoop]` to ensure the node is a loop node,
- `[cs:dom_invariant("intv", cs:ast(.))]` to ensure the loop invariant for the domain named *intv* is present for the AST node corresponding to the XML node,
- `[not(cs:isTop(cs:dom_invariant("intv", cs:ast(.))))]` to ensure the invariant is not trivial.

6.5 Experiment: Evaluate Abstract Domain of *Boxes*

In this section, an experiment is described. It uses *CodeStatistics* with the *JavaAI* analysis plugin to evaluate the abstract domain of *boxes* (see Chapter 4) focusing on the widening operator enhancements proposed in Section 4.6. Since we focus on the widening operator, the obvious choice is the analysis and comparison of loop invariants.

In the experiment, the same set of projects was used as in the *CodeStatistics* loop termination experiment presented in Section 5.5. We recall details of the projects in Table 6.1. Note that one of the projects that was present in the first experiment with *CodeStatistics* presented in the thesis — *Spring toolkit*, is missing in the list. This is because the project depends on a large number of external libraries and we did manage to configure the Java classpath for the Eclipse project properly. Therefore, there were lots of variable and type binding errors, which highly disturbed results.

Project Name	**Version**	**Size in KLoC**	**Number of loops**
Oracle Berkeley DB (BDB)	5.0.34	253	3412
Google App Engine (GAE)	1.6.4.1	163	1853
Apache Hadoop	1.0.1	292	4726
Hibernate	4.1.2	405	2875
JEdit	4.5.1	111	1523
Tomcat	7.0.27	216	3023
AspectJ	1.6.12	385	7534
Vuze	4.7.0.2	496	6140
Total		2321	31086

Table 6.1: Details of analysed projects (sizes are given in Kilo Lines of Code).

6.5.1 Configuration

In the experiment, the following domains configuration of *JavaAI* analysis plugin was used:

- $Bool \times Intv$ with the name *intv*,
- $Bool \times Boxes$ with empty thresholds sequence — named *boxes_ldd*, which is actually the implementation of the widening operator for *boxes* based on LDDs adopted to our representation,
- $Bool \times Boxes$ with simple thresholds sequence created from the box surrounding the first argument in the widening sequence — named *boxes_first-box*.

We were interested in finding non-trivial loop invariants. The *CodeStatistics* XPath configuration file was an extension of the one presented in Fig. 6.8. Patterns for computing non-trivial loop invariants for every domain in the experiment and multiple patterns for pairwise comparison of these loop invariants were created.

A timeout of 120 seconds was used for the analysis of a single method or initializer block. After this time, the analysis was stopped and no domain values were present in the analysis results for the method or initializer block. The XPath patterns that were prepared took this into consideration: we say that domain A is better than B if value of the domain A is present and either value of B does not exist or it is strictly less precise than A — a strict inclusion of corresponding concrete states.

6.5.2 Results

Before the actual results of the comparison are presented, we focus on timeouts that appeared during the analysis. Timeouts statistics details are presented in Table 6.2. For the domain *intv* there were no timeouts at all. For both variants of the domain of *boxes* timeouts occurred for analysis of the same methods with one exception: *JEdit* project, where analysis for the variant with simple threshold finished correctly. All timeouts happened for methods that were very large or highly complicated. For example, the only method analysis that exhausted the time limit for the *Berkley DB* project (`accept` from class `com.sleepycat.asm.ClassReader`) has over 800 lines of code, 37 internal loops, and over 20 integer variables in the scope at once in some of analysed control points!

Project Name	***intv***	***boxes_ldd***		***boxes_firstbox***	
		Methods	**Loops**	**Methods**	**Loops**
Berkley DB	0	1	37	1	37
Google App Engine	0	0	0	0	0
Hadoop	0	1	2	1	2
Hibernate	0	0	0	0	0
JEdit	0	1	35	0	0
Tomcat	0	1	3	1	3
AspectJ	0	3	7	3	7
Vuze	0	2	4	2	4
Total	0	9	88	8	53

Table 6.2: Timeout details for the anlaysis using the *boxes* domain.

Project Name	***intv***	***Boxes* with empty thresholds (*boxes_ldd*)**							
		Non-trivial		**Better**		**Worse**		**Incomp.**	
		$\sum$	Δ (%)	$\sum$	%	$\sum$	%	$\sum$	%
BDB	1986	1955	−1,56	378	19,03	39	1,96	2	0,10
GAE	1153	1153	0,00	113	9,80	0	0,00	2	0,17
Hadoop	2420	2428	0,33	656	27,11	6	0,25	23	0,95
Hibernate	1144	1144	0,00	277	24,21	0	0,00	0	0,00
JEdit	971	939	−3,30	206	21,22	37	3,81	1	0,10
Tomcat	1705	1701	−0,23	319	18,71	9	0,53	14	0,82
AspectJ	5366	5366	0,00	1882	35,07	21	0,39	112	2,09
Vuze	3782	3784	0,05	542	14,33	10	0,26	23	0,61
Total	18527	18470	−0,31	4373	23,60	122	0,66	177	0,96

Table 6.3: Comparison of non-trivial invariants for *intv* and *boxes_ldd*.

Now we present results of the comparison of both variants of the domain of *boxes* with the domain of intervals. The results of the comparison of the first variant of the domain of *boxes* — *boxes_ldd* (with empty *widening sequence thresholds*) are presented in Table 6.3. One may notice a slightly smaller number of non-trivial loop invariants found for the domain of *boxes* in some of the projects. This is caused by timeouts that appeared in the analysis. The number of cases, in which we obtain strictly more precise loop invariant, is quite high: almost 1/4 of all cases. Unfortunately, we also can notice a number of cases, in which we obtain a strictly worse result (this includes timeouts), and a few when invariants are incomparable.

Project Name	***intv***	***Boxes* with simple thresholds (*boxes_firstbox*)**							
		Non-trivial		**Better**		**Worse**		**Incomp.**	
		$\sum$	Δ (%)	$\sum$	%	$\sum$	%	$\sum$	%
BDB	1986	1957	−1,46	385	19,39	37	1,86	1	0,05
GAE	1153	1153	0,00	125	10,84	0	0,00	0	0,00
Hadoop	2420	2431	0,45	695	28,72	2	0,08	0	0,00
Hibernate	1144	1144	0,00	279	24,39	0	0,00	0	0,00
JEdit	971	976	0,51	248	25,54	0	0,00	0	0,00
Tomcat	1705	1704	−0,06	335	19,65	6	0,35	5	0,29
AspectJ	5366	5370	0,07	2000	37,27	8	0,15	13	0,24
Vuze	3782	3784	0,05	590	15,60	4	0,11	3	0,08
Total	18527	18519	−0,04	4657	25,14	57	0,31	22	0,12

Table 6.4: Comparison of non-trivial invariants for *intv* and *boxes_firstbox*.

Analogical results for the variant of *boxes* with simple *widening sequence thresholds* — *boxes_firstbox*, is presented in Table 6.4. The analysis with the variant with simple thresholds caused finding a few non-trivial invariants more than with the basic version of the widening operator. The number of strictly better results is also higher: about 1,5% more such cases. With adding very simple thresholds we have reduced the number of incomparable invariants by almost 8 times. Also, when we take into consideration timeouts (subtract numbers from Table 6.2), we obtain only 4 loops, for which analysis with the domain of intervals gives strictly more precise result.

The last part of the results concerns the direct comparison of the two widening strategies for the domain of *boxes*, which is presented in Table 6.5. It is worth noting that we have a marginal number of the cases, in which adding simple thresholds leads to a strictly worse or incomparable result. On the other hand, the number of cases, in which the result is strictly more precise, is noticeable — over 2%.

Project Name	*boxes_ldd*	***Boxes* with simple thresholds (*boxes_firstbox*)**							
		Non-trivial		**Better**		**Worse**		**Incomp.**	
		Total	Δ (%)	∑	%	∑	%	∑	%
BDB	1955	1957	0,10	16	0,82	0	0,00	0	0,00
GAE	1153	1153	0,00	18	1,56	0	0,00	0	0,00
Hadoop	2428	2431	0,12	57	2,35	5	0,21	0	0,00
Hibernate	1144	1144	0,00	3	0,26	0	0,00	0	0,00
JEdit	939	941	0,21	54	5,75	0	0,00	0	0,00
Tomcat	1701	1704	0,18	49	2,88	1	0,06	3	0,18
AspectJ	5366	5364	−0,04	149	2,78	2	0,04	16	0,30
Vuze	3784	3784	0,00	70	1,85	1	0,03	0	0,00
Total	14686	14694	0,05	346	2,36	8	0,05	19	0,13

Table 6.5: Direct comparison of *Boxes* with two widening operators: one with empty thresholds (*boxes_ldd*) and the other with simple widening sequence thresholds (*boxes_firstbox*).

6.6 Summary

In this chapter, we have presented a tool that helps to evaluate and compare abstract domains. We have applied it to evaluate the domain of *boxes* and compare the proposed widening operator with the existing one from the construction based on LDDs. The tool showed that the domain of boxes with a simple variant of the new widening operator is almost never less precise than the domain of intervals, and that in about 1/4 situations the result is strictly more precise. Additionally, it turned out that the domain of *boxes* with the proposed widening operator is almost never less precise than with the previously known widening operator, and that in about 1.5% situations the result is strictly better.

Chapter 7

Conclusions

The main objective of the thesis was to design a technique that could be used to investigate practicality of abstract interpretation domains on real Java code.

The first matter we focused on was the *abstract interpretation* framework. An implementation of the abstract domain of *boxes* was presented that is based on the *sweeping line* technique that was not applied to the *abstract interpretation* before. Two new widening operators for the domain were introduced. Both use *threshold* points to enhance precision. The first operator is a generic one, the second is an instance of the generic operator that has an interesting one-step precision characteristic. Namely, it was proved that adding new threshold points to *widening step threshold* may increase precision of the single-step application of the operator. Examples were presented, in which the proper choice of threshold points increases precision, and also examples, in which the new widening operator is more precise that previously known.

The second matter concerned *CodeStatistics* that had been developed. It is a tool that makes it possible to discover patterns in large Java projects and generate specifications. It was successfully applied to generate JML loop termination conditions. Using it, we generated the JML *decreases* clauses for over 78% of numeric `for` loops in large Java projects.

The last task of the thesis was a combination of the first and the second matter. The *CodeStatistics* tool was extended by implementing a plugin engine that can be used to extend the pattern language with semantic tests. An *abstract interpretation* analyser — *JavaAI* was implemented as a part of this study. It works as the analysis plugin for *CodeStatistics*. By connecting these two tools it was possible to evaluate and compare abstract domains on real Java code. It proved that the domain of *boxes* with a simple variant of the proposed widening operator is more precise than with the existing widening operator on typical code. Note that loop invariants that were generated in the experiment can be transformed into specifications and together with the loop termination conditions these specifications may be used to support verification tools like KeY or ESC/Java2.

Bibliography

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, Aug. 2006.

[2] Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., and Schmitt, P. H. The KeY tool. *Software and System Modeling* 4, 1 (2005), 32–54.

[3] Albert, E., Arenas, P., Genaim, S., and Puebla, G. Dealing with Numeric Fields in Termination Analysis of Java-like Languages. In: *Formal Techniques for Java-like Programs*. 2008, 77–87.

[4] Andersen, M., Barnett, M., Fähndrich, M., Grunkemeyer, B., King, K., and Logozzo, F. *Code Contracts User Manual*. Microsoft Research, 2012.

[5] Artho, C. Finding Faults in Multi-threaded Programs. MA thesis. ETH Zürich, 2001.

[6] Artho, C. and Havelund, K. Applying Jlint to Space Exploration Software. In: *VMCAI*. Ed. by Steffen, B. and Levi, G. Vol. 2937. Lecture Notes in Computer Science. Springer, 2004, 297–308.

[7] Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J., and Zhou, Y. Evaluating static analysis defect warnings on production software. In: *PASTE*. Ed. by Das, M. and Grossman, D. ACM, 2007, 1–8.

[8] Bagnara, R. A Hierarchy of Constraint Systems for Data-Flow Analysis of Constraint Logic-Based Languages. *Science of Computer Programming* 30, 1-2 (1998), 119–155.

[9] Bagnara, R., Hill, P. M., and Zaffanella, E. Exact join detection for convex polyhedra and other numerical abstractions. *Computational Geometry* 43, 5 (2010), 453–473.

[10] Bagnara, R., Hill, P. M., and Zaffanella, E. Widening operators for powerset domains. *International Journal on Software Tools for Technology Transfer* 8, 4-5 (2006), 449–466.

[11] Beck, K. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.

[12] Bentley, J. L. and Ottmann, T. A. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Transactions on Computers* 28 (9 Sept. 1979), 643–647.

[13] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. A Static Analyzer for Large Safety-Critical Software. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*. ACM Press, June 2003, 196–207.

[14] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones. In: ed. by Mogensen, T., Schmidt, D., and Sudborough, I. H. Lecture Notes in Computer Science 2566. Springer-Verlag, 2002. Chap. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, 85–108.

[15] Boehm, B. W. and Basili, V. R. Software Defect Reduction Top 10 List. *IEEE Computer* 34, 1 (2001), 135–137.

[16] Bourdoncle, F. Efficient Chaotic Iteration Strategies With Widenings. In: *In Proceedings of the International Conference on Formal Methods in Programming and their Applications*. Springer-Verlag, 1993, 128–141.

[17] Bowen, J. P. and Hinchey, M. G. Seven More Myths of Formal Methods. *IEEE Software* 12 (4 July 1995), 34–41.

[18] Bowen, J. P. and Hinchey, M. G. The Use of Industrial-Strength Formal Methods. In: *Proceedings of 21st International Computer Software and Application Conference (COMPSAC'97*. IEEE Computer Society Press, 1997, 332–337.

[19] Brown, W. H., Malveau, R. C., McCormick, H. W., and Mowbray, T. J. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998.

[20] Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J. R., Leavens, G. T., Leino, K. R. M., and Poll, E. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer* 7, 3 (2005), 212–232.

[21] Center, N. S. S. D. *Mariner 1*. URL: `http://nssdc.gsfc.nasa.gov/nmc/spacecraftDisplay.do?id=MARIN1`.

[22] Chaki, S., Gurfinkel, A., and Strichman, O. Decision diagrams for linear arithmetic. In: *Formal Methods in Computer-Aided Design*. IEEE, 2009, 53–60.

[23] Chalin, P., Kiniry, J. R., Leavens, G. T., and Poll, E. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In: *Formal Methods for Components and Objects*. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, 342–363.

[24] Chalin, P. and Rioux, F. JML Runtime Assertion Checking: Improved Error Reporting and Efficiency Using Strong Validity. In: *Formal Methods*. Lecture Notes in Computer Science. Springer-Verlag, Turku, Finland, 2008, 246–261.

[25] Cielecki, M., Fulara, J., Jakubczyk, K., and Jancewicz, Ł. Propagation of JML non-null annotations in Java programs. In: *PPPJ*. 2006, 135–140.

[26] Cok, D. R. and Kiniry, J. R. ESC/Java2: Uniting ESC/Java and JML. In: *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Vol. 3362. Lecture Notes in Computer Science. Springer–Verlag, 2004, 108–128.

[27] Copeland, T. *PMD Applied: An Easy to Use Guide for Developers*. An easy-to-use guide for developers. Centennial Books, 2005.

[28] Cousot, P. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM Computing Surveys* 28, 2 (June 1996), 324–328.

[29] Cousot, P. Abstract Interpretation MIT Course. URL: `http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/`. 2005.

[30] Cousot, P. Avionic Software Verification by Abstract Interpretation. In: *ISoLA Workshop On Leveraging Applications of Formal Methods, Verification and Validation. Special Workshop Theme: Formal Methods in Avionics, Space and Transport*. 2007.

[31] Cousot, P. The Calculational Design of a Generic Abstract Interpreter. In: *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[32] Cousot, P. and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 1977, 238–252.

[33] Cousot, P. and Cousot, R. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming* 13, 2–3 (1992). (The editor of Journal of Logic Programming has mistakenly published the unreadable galley proof. For a correct version of this paper, see `http://www.di.ens.fr/~cousot`.), 103–179.

[34] Cousot, P. and Cousot, R. Abstract Interpretation Frameworks. *Journal of Logic and Computation* 2, 4 (Aug. 1992), 511–547.

[35] Cousot, P. and Cousot, R. Basic Concepts of Abstract Interpretation. In: *Building the Information Society*. Ed. by Jacquard, R. Kluwer Academic Publishers, 2004, 359–366.

[36] Cousot, P. and Cousot, R. Constructive Versions of Tarski's Fixed Point Theorems. *Pacific Journal of Mathematics* 81, 1 (1979), 43–57.

[37] Cousot, P. and Cousot, R. 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*. IEEE Computer Society Press, Los Alamitos, California, 1994, 95–112.

[38] Cousot, P. and Cousot, R. Static determination of dynamic properties of programs. In: *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France, 1976, 106–130.

[39] Cousot, P. and Cousot, R. Systematic design of program analysis frameworks. In: *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 1979, 269–282.

[40] Cousot, P. and Halbwachs, N. Automatic discovery of linear restraints among variables of a program. In: *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 1978, 84–97.

[41] Cousot, P. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Electronic Notes in Theoretical Computer Science* 6 (1997), 77–102.

[42] Cousot, P. and Cousot, R. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In: *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*. Springer-Verlag, 1992, 269–295.

[43] Cousot, P. and Cousot, R. Galois Connection Based Abstract Interpretations for Strictness Analysis (Invited Paper). In: *Formal Methods in Programming and Their Applications*. Lecture Notes in Computer Science. Springer, 1993, 98–127.

[44] Craigen, D., Craigen, D., Canada, O., Canada, O., Gerhart, S., Gerhart, S., Ralston, T., and Brown, R. H. *An International Survey of Industrial Applications of Formal Methods Volume 2 Case Studies*. 1993.

[45] Datamonitor. *Software: Global Industry Guide 2010*. report summary: `http://www.datamonitor.com/store/Product/software_global_industry_guide_2010?productid=4F026C5C-EBCC-4193-AD27-77260196E7F5`. Jan. 31, 2011.

[46] Dean, J. and Ghemawat, S. MapReduce: simplified data processing on large clusters. *Communications of the ACM* 51, 1 (Jan. 2008), 107–113.

[47] Delmas, D. and Souyris, J. Astrée: From research to industry. In: *In Static Analysis Symposium (SAS*. Vol. 4634. Lecture Notes in Computer Science. 2007, 437–451.

[48] Dowson, M. The Ariane 5 software failure. *SIGSOFT Software Engineering Notes* 22 (2 Mar. 1997), 84–.

[49] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1–3 (Dec. 2007), 35–45.

[50] Fenton, N. E. and Ohlsson, N. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering* 26, 8 (Aug. 2000), 797–814.

[51] Flanagan, C., Rustan, K., and Leino, M. Houdini, an Annotation Assistant for ESC/Java. In: *In Proceedings of the International Symposium of Formal Methods Europe*. 2001, 500–517.

[52] Fortune, S. A sweepline algorithm for Voronoi diagrams. In: *Proceedings of the Second Annual Symposium on Computational Geometry*. SCG '86. ACM, Yorktown Heights, New York, United States, 1986, 313–322.

[53] Fulara, J., Durnoga, K., Jakubczyk, K., and Schubert, A. Relational Abstract Domain of Weighted Hexagons. In: *Proceedings of the 2nd Workshop on Numerical and Symbolic Abstract Domains*. Vol. 267. Elsevier Science Publishers B. V., Oct. 2010, 59–72.

[54] Fulara, J. and Jakubczyk, K. Practically Applicable Formal Methods. In: *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*. SOFSEM '10. Springer-Verlag, 'pindlerov Ml&#253;n, Czech Republic, 2010, 407–418.

[55] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*. Addison-Wesley, Jan. 1995.

[56] Gittens, M. and Godwin, D. A Closer Look at the Pareto Principle for Software. *Software Quality Provessional* 9, 4 (Sept. 2007), 4–14.

[57] Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A. *The Java Language Specification, Java SE 7 Edition*. 2012.

[58] Granger, P. Static Analyses of Congruence Properties on Rational Numbers (Extended Abstract). In: *Static Analysis Symposium*. Vol. 1302. Lecture Notes in Computer Science. 1997, 278–292.

[59] Granger, P. Static Analysis of Linear Congruence Equalities among Variables of a Program. In: *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Vol.1*. Vol. 493. Lecture Notes in Computer Science. Springer, 1991, 169–192.

[60] Gurfinkel, A. and Chaki, S. BOXES: a symbolic abstract domain of boxes. In: *Static Analysis Symposium*. Lecture Notes in Computer Science. Springer-Verlag, Perpignan, France, 2010, 287–303.

[61] Hall, A. Realising the Benefits of Formal Methods. In: *Formal Methods and Software Engineering*. Vol. 3785. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, 1–4.

[62] Hall, A. Seven Myths of Formal Methods. *IEEE Software* 7 (5 Sept. 1990), 11–19.

[63] Handjieva, M. and Tzolovski, S. Refining Static Analyses by Trace-Based Partitioning Using Control Flow. In: *Static Analysis Symposium*. Vol. 1503. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1998, 200–214.

[64] HAREL, D. and MARELLY, R. Specifying and executing behavioral requirements: the play-in/play-out approach. *Software and System Modeling* 2, 2 (2003), 82–107.

[65] HEITMEYER, C. Formal methods: A panacea or academic poppycock? In: *ZUM '97: The Z Formal Specification Notation*. Vol. 1212. Lecture Notes in Computer Science. 10.1007/BFb0027280. Springer Berlin / Heidelberg, 1997, 1–9.

[66] HOVEMEYER, D. and PUGH, W. Finding bugs is easy. *SIGPLAN Notices* 39, 12 (2004), 92–106.

[67] HOVEMEYER, D. and PUGH, W. Finding more null pointer bugs, but not too many. In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE '07. ACM, 2007, 9–14.

[68] HOWE, J. M., KING, A., and LAWRENCE-JONES, C. Quadtrees as an Abstract Domain. *Electron. Notes Theor. Comput. Sci.* 267 (1 Oct. 2010), 89–100.

[69] JAKUBCZYK, K. Sweeping in Abstract Interpretation. In: *Proceedings of the 3rd Workshop on Numerical and Symbolic Abstract Domains*. Vol. 288. Elsevier Science Publishers B. V., 2012, 25–36.

[70] JURAN, J. M. The non-Pareto principle; mea culpa. *Juran Institute* (2001).

[71] KLEENE, S. *Mathematical Logic*. Dover Books on Mathematics Series. Dover, 2002.

[72] LANDI, W. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems* 1, 4 (1992), 323–337.

[73] LEAVENS, G. and CHEON, Y. *Design by Contract with JML*. 2004.

[74] LEAVENS, G. T., BAKER, A. L., and RUBY, C. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes* 31 (3 May 2006), 1–38.

[75] LEAVENS, G. T., POLL, E., CLIFTON, C., CHEON, Y., RUBY, C., COK, D., MÜLLER, P., KINIRY, J., CHALIN, P., and ZIMMERMAN, D. M. *JML Reference Manual (DRAFT)*. July 2011.

[76] LEVESON, N. G. An Investigation of the Therac-25 Accidents. *IEEE Computer* 26 (1993), 18–41.

[77] LOGOZZO, F. and FÄHNDRICH, M. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In: *Proceedings of the 2008 ACM Symposium on Applied Computing*. SAC '08. ACM, Fortaleza, Ceara, Brazil, 2008, 184–188.

[78] Markets and Markets. *World Mobile Applications Market—Advanced Technologies, Global Forecast*. report summary: `http://www.marketsandmarkets.com/Market-Reports/mobile-applications-228.html`. Aug. 2010.

[79] Mauborgne, L. and Rival, X. Trace partitioning in abstract interpretation based static analyzers. In: *European Symposium on Programming*. Lecture Notes in Computer Science. Springer, 2005, 5–20.

[80] McCabe, T. J. A complexity measure. In: *Proceedings of the 2nd international Conference on Software Engineering*. ICSE '76. IEEE Computer Society Press, San Francisco, California, United States, 1976, 407–.

[81] Meyer, B. *Object-Oriented Software Construction*. Prentice Hall PTR, Mar. 1997.

[82] Microsystems, I. S. *Code Conventions for the Java Programming Language*. 1997.

[83] Miné, A. The octagon abstract domain. *Higher-Order and Symbolic Computation* 19 (1 Mar. 2006), 31–100.

[84] Miné, A. Weakly Relational Numerical Abstract Domains. PhD thesis. l' école Normale Supérieure, 2004.

[85] Moor, O., Sereni, D., Verbaere, M., Hajiyev, E., Avgustinov, P., Ekman, T., Ongkingco, N., and Tibble, J. .QL: Object-Oriented Queries Made Easy. In: *Generative and Transformational Techniques in Software Engineering II*. Vol. 5235. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, 78–133.

[86] Plotkin, G. D. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming* 60-61 (2004), 17–139.

[87] Rival, X. and Mauborgne, L. The Trace Partitioning Abstract Domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 5 (2007).

[88] Sankaranarayanan, S., Ivančić, F., Shlyakhter, I., and Gupta, A. Static Analysis in Disjunctive Numerical Domains. In: *Static Analysis Symposium*. Vol. 4134. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, 3–17.

[89] Simon, A., King, A., and Howe, J. Two Variables per Linear Inequality as an Abstract Domain. English. In: *Logic Based Program Synthesis and Transformation*. Vol. 2664. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, 71–89.

[90] Tarski, A. A Lattice-theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics* 5, 2 (3 1955), 285–309.

[91] Walton, L. *Eclipse Metrics plugin:* `http://metrics.sourceforge.net/`.

[92] Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J. Formal methods: Practice and experience. *ACM Computing Surveys* 41 (4 Oct. 2009), 19:1–19:36.

[93] Yoo, J., Cha, S., and Jee, E. A Verification Framework for FBD Based Software in Nuclear Power Plants. In: *APSEC '08: Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*. IEEE Computer Society, 2008, 385–392.