University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

JĘDRZEJ FULARA

ABSTRACT ANALYSIS OF NUMERICAL AND CONTAINER VARIABLES

PhD Dissertation

under supervision of
Prof. Andrzej Tarlecki

October 2012

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.

Warsaw, October 2012

Jędrzej Fulara

SUPERVISOR'S DECLARATION

This dissertation is ready to be reviewed.

Warsaw, October 2012

Prof. Andrzej Tarlecki

ABSTRACT

This thesis presents new techniques of static program analysis by abstract interpretation.

In the first part of the thesis we focus on *numerical abstract domains* that can be used to automatically discover certain numerical properties of programs. We introduce two new abstract domains: the domain of *weighted hexagons*, and its enriched version, the domain of *strict weighted hexagons*. These domains capture constraints of the form $x \leqslant a \cdot y$ and $x < a \cdot y$, where x and y are program variables and a is a non-negative constant. They lie between the existing domains of pentagons and TVPI in terms of both expressiveness and efficiency.

The second part of the thesis concerns analysis of programs that use containers of scalar values. Existing techniques can be used only to analyse arrays of numerical values. We propose a novel technique that can be used to reason about the content of dictionaries and arrays of arbitrary scalar types. The flexibility of our approach is illustrated on various examples, including analysis of numerical arrays and string-keyed dictionaries.

ACM classification D.2.4, D.3.1, F.3.1
Keywords Static analysis, abstract interpretation, abstract domains

STRESZCZENIE

Niniejsza rozprawa poświęcona jest nowym technikom statycznej analizy programów, z wykorzystaniem *abstrakcyjnej interpretacji*.

W pierwszej części rozprawy, skupiamy się na *numerycznych dziedzinach abstrakcyjnych*, które służą do automatycznego wykrywania zależności między zmiennymi numerycznymi. Przedstawiamy dwie nowe dziedziny abstrakcyjne *ważonych sześciokątów* i *ścisłych ważonych sześciokątów*, służące do wykrywania powiązań postaci $x \leqslant a \cdot y$ oraz $x < a \cdot y$, gdzie x oraz y oznaczają zmienne w programie, a a jest pewną nieujemną stałą. Dziedziny te plasują się pomiędzy dziedziną pięciokątów i dziedziną TVPI, zarówno pod względem siły wyrazu jak i wydajności.

Druga część rozprawy omawia abstrakcyjną analizę programów używających niektórych struktur danych. Istniejące techniki pozwalają jedynie na wnioskowanie na temat zawartości tablic numerycznych. My proponujemy nową technikę, która może być użyta do modelowania zawartości tablic oraz słowników zawierających elementy dowolnych typów. Elastyczność naszego rozwiązania została zilustrowana na przykładach analizy tablic numerycznych oraz słowników z kluczami napisowymi.

Klasyfikacja ACM D.2.4, D.3.1, F.3.1
Słowa kluczowe Analiza statyczna, abstrakcyjna interpretacja, dziedziny abstrakcyjne

ACKNOWLEDGMENTS

I would like to thank my supervisor, professor Andrzej Tarlecki for his invaluable help in preparing this thesis, for his time spent on numerous meetings with me and for motivating me to complete this work. I am also grateful to Aleksy Schubert for pointing me to the area of abstract interpretation and for all the inspiring discussions that we have lead over the past five years.

Last but not least, I would like to thank my wife, for her support and patience.

CONTENTS

1 INTRODUCTION

Computer software, its quality and reliability is of a great importance for every sector of modern economy. A shop, a bank, a factory or public administration could not work properly without reliable computer systems. A crash of a computer game causes some inconvenience for the player. An insecure banking system exposes the client to possible loss of money. An error in an automatic flight control system of an aircraft may even endanger human lives.

Software companies understand how important the reliability of their products is. The quality assurance in the software development process is considered to be as important as the design and implementation. Many companies have dedicated teams of testers, whose goal is to find all possible errors and vulnerabilities in an application before delivering it to the customer. The approach to quality assurance based on testing is very popular, as it can reveal many problems that are most likely to happen in practice. However, it has one major drawback. Testing is just observing the behaviour of a system in some (possibly tricky) scenarios, but it does not give any information about scenarios that were not tested. The better the tests are, the bigger scope of scenarios they cover, but is is usually impossible to check *all* possible executions. Thus, testing cannot ensure that all problems are detected and that the system will not fail in any case: *Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.* — Edsger W. Dijkstra [20].

There exist methods that can be used to ensure certain correctness properties of an application, that is to prove that no error may occur in any execution. These techniques are, unlike testing, performed *statically*, i.e. they do not rely on observing results of a program execution. They reason about properties of the program using its text and structure. Such methods can be applied to prove the absence of some types of errors or, more general, to ensure that a program complies with some specification. It is important that these techniques can guarantee that a program is correct, that is they can reason about every program execution that could potentially happen. On the other hand, such analysis may signal *false alarms*. It can report a potential error even when the program is correct. These techniques are known as *static program analysis* and are part of *formal methods*.

Static program analysis is most widely used for systems that concern human security. It was for instance successfully applied to verify avionics software in the Airbus corporation [11, 63] and by the European Space Agency, to automatically prove the absence of some type of programming errors in control software of an Automated Transport Vehicle [7]. Such formal techniques were also used to verify control software in nuclear power plants [66]. Currently there are attempts to use formal methods also in ordinary applications. The European research project MOBIUS [4] aimed to create a framework for generating correctness certificates for software for mobile devices. Microsoft Research develops a general-purpose static analyser for programs written in languages from the .NET platform [2].

Static analysis is a very challenging research area. In general it is not possible to automatically determine whether a given program may or may not exhibit a runtime error (this question is undecidable). One can only attempt to find some approximate solutions. The static analysis should be *sound*, i.e. if it reports that a program is correct, the program must not raise an error in any possible execution. However, the analysis can claim that a correct program may contain some error. It is a great challenge to design a precise analysis, which signals only few such false alarms.

In this thesis we focus on one particular approach to static analysis, called *abstract interpretation*.

1.1 ORDERS, FUNCTIONS AND FIXPOINTS

Before we introduce the abstract interpretation, we shall briefly recall basic facts about orders, functions and fixpoints.

ORDERS Given a set X, a binary relation $\sqsubseteq \subseteq X \times X$ is a *partial order*, when for all $x, y, z \in X$:

1. $x \sqsubseteq x$ (reflexivity),

2. if $x \sqsubseteq y$ and $y \sqsubseteq x$ then $x = y$ (antisymmetry),

3. if $x \sqsubseteq y$ and $y \sqsubseteq z$ then $x \sqsubseteq z$ (transitivity).

Such a pair $\langle X, \sqsubseteq \rangle$ is called a *partially ordered set* (or a *poset*). A subset $Y \subseteq X$ is *directed*, if for every $x, y \in Y$ there exists $z \in Y$ such that $x \sqsubseteq z$

and $y \sqsubseteq z$. A poset $\langle X, \sqsubseteq \rangle$ is *complete*, if each of its directed subsets $Y \subseteq X$ has a supremum (denoted $\sqcup Y$).

A set X with two binary operations $\sqcup, \sqcap \colon X \times X \to X$ (referred to as *join* and *meet*) is called a *lattice*, if for every $x, y, z \in X$ the following conditions hold:

1. $x \sqcup y = y \sqcup x$ and $x \sqcap y = y \sqcap x$ (commutativity),
2. $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$ and $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$ (associativity),
3. $(x \sqcup y) \sqcap x = x$ and $(x \sqcap y) \sqcup x = x$ (absorption).

The binary meet and join operations induce a partial order $\sqsubseteq$ over X given by $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$.

In a lattice, each non-empty finite subset of X has a supremum and infimum. A lattice is *complete*, if *every* subset of X has a supremum and infimum. A complete lattice has always the least and the greatest element that are usually denoted as $\bot$ (bottom) and $\top$ (top), respectively.

If $\langle D, \sqcup, \sqcap \rangle$ is a (complete) lattice and X is a set, then $\langle X \to D, \curlyvee, \curlywedge \rangle$ (where $X \to D$ denotes the set of functions from X to D), such that

$$f_1 \curlyvee f_2 \triangleq \lambda x. f_1(x) \sqcup f_2(x) \qquad f_1 \curlywedge f_2 \triangleq \lambda x. f_1(x) \sqcap f_2(x)$$

is also a (complete) lattice and is called a *point-wise lifting* of $\langle D, \sqcup, \sqcap \rangle$.

FUNCTIONS We start with introducing some notation that we use throughout this thesis. We write $f \colon X \to Y$ and $g \colon X \rightharpoonup Y$ to denote a total and partial function from X to Y, respectively.

Given two functions $f \colon X \to Y$ and $g \colon Y \to Z$, $g \circ f \colon X \to Z$ denotes the composition of f and g, i.e. for each $x \in X$, $(g \circ f)(x) \triangleq g(f(x))$.

Given a function $f \colon X \to Y$ and a set $Z \subseteq X$, $f|_Z \colon Z \to Y$ denotes a function $g \colon Z \to Y$ such that for each $z \in Z$, $g(z) \triangleq f(z)$.

For a function $f \colon X \to Y$ we denote its *image function* by $f^{\to} \colon \mathcal{P}(X) \to \mathcal{P}(Y)$, where $f^{\to}(P) \triangleq \{y \mid \exists_{x \in P}\, f(x) = y\}$.

We define functions either using lambda notation, or by explicit enumeration of their arguments and values. For instance a (partial) mapping in $X \to Y$ would be written as $[x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots]$, where $x_i \in X$ and $y_i \in Y$ (and all $x_1, \ldots, x_n$ are mutually distinct). A similar notation will be used to define a function that is equal to some given $f \colon X \to Y$ except for a finite set of arguments, e.g. $f[z \mapsto y]$ will denote function $g \colon X \to Y$ such that $g(x) \triangleq f(x)$ for all $x \in X$ other than z, and $g(z) \triangleq y$.

Given two posets $\langle X, \sqsubseteq \rangle$ and $\langle Y, \preceq \rangle$, a function $f \colon X \to Y$ is *monotone* (or *order preserving*), if $x \sqsubseteq y \Rightarrow f(x) \preceq f(y)$ for all $x, y \in X$. It is *anti-monotone*, if $x \sqsubseteq y \Rightarrow f(y) \preceq f(x)$.

A function $g \colon X \to Y$ is *continuous*, if it preserves all directed suprema, in other words for each directed subset $P \subseteq X$, $\sqcup g^{\to}(P) = g(\sqcup P)$. Each continuous function is monotone.

GALOIS CONNECTIONS Let $\langle X, \sqsubseteq \rangle$ and $\langle Y, \preceq \rangle$ be two posets and $\alpha \colon X \to Y$ and $\gamma \colon Y \to X$ be monotone functions. We say that α and γ form a *Galois connection*:

$$\langle X, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle Y, \preceq \rangle$$

if for all $x \in X$ and $y \in Y$ it holds that $\alpha(x) \preceq y \Leftrightarrow x \sqsubseteq \gamma(y)$. This condition is sometimes (equivalently) formulated as $x \sqsubseteq (\gamma \circ \alpha)(x)$ and $(\alpha \circ \gamma)(y) \preceq y$ for all $x \in X$ and $y \in Y$ [52]. The functions α and γ are called *lower* and *upper adjoints*, respectively.

Given two complete lattices $\langle X, \sqcup, \sqcap \rangle$ and $\langle Y, \vee, \wedge \rangle$, a monotone function $\alpha \colon X \to Y$ that preserves all least upper bounds always uniquely defines $\gamma \colon Y \to X$ such that $\langle X, \sqcup, \sqcap \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle Y, \vee, \wedge \rangle$ is a Galois connection; such γ is given by:

$$\gamma(y) \triangleq \bigsqcup \{x \mid \alpha(x) \preceq y\} \, .$$

Likewise, given the monotone upper adjoint γ that preserves all greatest lower bounds, there always exist and is uniquely defined a lower adjoint α:

$$\alpha(x) \triangleq \bigwedge \{y \mid x \sqsubseteq \gamma(y)\} \, .$$

Given a Galois connection $\langle X, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle Y, \preceq \rangle$ and a function $f \colon X \to X$, a function $g \colon Y \to Y$ is called a *sound abstraction of* f if

$$\forall_{x \in X} \; f(x) \sqsubseteq (\gamma \circ g \circ \alpha)(x) \, . \tag{1.1}$$

Galois connections can be composed. Given two Galois connections $\langle X, \sqsubseteq \rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle Y, \preceq \rangle$ and $\langle Y, \preceq \rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle Z, \trianglelefteq \rangle$, $(\alpha_2 \circ \alpha_1, \gamma_1 \circ \gamma_2)$ is a Galois connection as well:

$$\langle X, \sqsubseteq \rangle \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} \langle Z, \trianglelefteq \rangle \, .$$

The notion of sound abstraction is preserved by the composition of Galois connections, i.e. for Galois connections as above and functions $f \colon X \to X$, $g \colon Y \to Y$ and $h \colon Z \to Z$, if g is a sound abstraction of f and h is a sound abstraction of g, then h is a sound abstraction of f.

FIXPOINTS An element $x \in X$ is called a *fixpoint* of a function $f\colon X \to X$ if $f(x) = x$.

Theorem 1.2 (Tarski). *If $\langle X, \sqcup, \sqcap \rangle$ is a complete lattice and $f\colon X \to X$ is a monotone function, then the set of fixpoints of f is also a complete lattice [65].*

This theorem immediately implies that f has at least one fixpoint. The *least fixpoint* of f is denoted as $\mathtt{lfp}(f)$, while the *greatest fixpoint* of f is written as $\mathtt{gfp}(f)$.

Theorem 1.3 (Kleene). *If $\langle X, \sqsubseteq \rangle$ is a complete partial order and $f\colon X \to X$ is a continuous (and therefore monotone) function, then the least fixpoint of f is equal to $\sqcup\{f^n(\bot) \mid n \in \mathbb{N}\}$, where $f^0(\bot) \triangleq \bot$ and $f^{i+1}(\bot) \triangleq f(f^i(\bot))$ for $i \in \mathbb{N}$.*

The sequence $\bot, f(\bot), f(f(\bot)), \ldots$ is called *Kleene's sequence*. Kleene's theorem requires the function f to be continuous, while in program analysis we often deal with functions that are only monotone. We present now an alternative version of Tarski's theorem that gives a characterisation of the least fixpoint of a monotone function.

Let $\langle X, \sqcup, \sqcap \rangle$ be a complete lattice and $f\colon X \to X$ be a monotone function. Additionally, let μ be the smallest ordinal of cardinality greater than the cardinality of X. For an element $d \in X$, we define the *iteration sequence* of f starting from d $\langle f^\delta(d)\colon \delta \in \mu \rangle$ by transfinite induction:

1. $f^0(d) \triangleq d$,
2. $f^{\delta+1}(d) \triangleq f(f^\delta(d))$ for $\delta \in \mu$,
3. $f^\delta(d) \triangleq \bigsqcup_{\alpha<\delta} f^\alpha(d)$ for any limit ordinal $\delta \in \mu$.

We say that $f^\epsilon(d)$ is a *limit* of $\langle f^\delta(d)\colon \delta \in \mu \rangle$, if for each $\beta \in \mu$, $\beta \geqslant \epsilon$ implies $f^\beta(d) = f^\epsilon(d)$. This limit is denoted by $\sqcup\langle f^\delta(d)\colon \delta \in \mu \rangle$.

We present now the following version of Tarski's theorem that gives a characterisation of the least fixpoint of a monotone function [14]:

Theorem 1.4 (Tarski, "constructive" version). *If $\langle X, \sqcup, \sqcap \rangle$ is a complete lattice of cardinality smaller than the cardinality of an ordinal μ and $f\colon X \to X$ is a monotone function, then the set of fixpoints of f is also a complete lattice and the least fixpoint of f is equal to $\sqcup\langle f^\delta(\bot)\colon \delta \in \mu \rangle$.*

Given two complete lattices linked by a Galois connection, there exists a correspondence between least fixpoints of a monotone function and of its monotone sound abstraction [10]:

Theorem 1.5 (Tarski's fixpoint transfer). *Let* $\langle X, \sqcup, \sqcap \rangle$ *and* $\langle Y, \vee, \wedge \rangle$ *be complete lattices linked with a Galois connection* $\langle X, \sqcup, \sqcap \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle Y, \vee, \wedge \rangle$. *If* $f \colon X \to X$ *is a monotone function and* $g \colon Y \to Y$ *is monotone function that is a sound abstraction of* f *(with respect to the Galois connection as above), then*

$$\mathrm{lfp}(f) \sqsubseteq \gamma(\mathrm{lfp}(g)) \, .$$

Least fixpoints of monotone functions play an important role in program analysis. We will see that the set of all possible states reached by any execution of a program can be characterised as the least fixpoint of a (monotone) transfer function, that is of a function that gives meaning to all types of instructions in the program. The abstract interpretation will compute the least fixpoint of some sound abstraction of the transfer function.

1.2 ABSTRACT INTERPRETATION

Abstract interpretation is a static analysis technique introduced by Patrick and Radhia Cousot [12] in late seventies. As we have already mentioned, it is in general not possible to automatically decide if any execution of a given program may exhibit some non-trivial property. The idea of abstract interpretation is to find some over-approximation of all possible program executions, that is to compute at each program point some superset of the set of program states that may occur at this point. If the over-approximation does not contain any forbidden state (such as e.g. a state in which a null variable was de-referenced), then the program is *for sure* safe.

1.2.1 *Language Definition*

Abstract Interpretation has been most thoroughly studied for programs that manipulate some scalar variables, thus we formalise it first for a tiny language that operates only over scalars. We do not elaborate here on the details of the programming language. We fix only the most important notation. Thus, we introduce a finite set of variables $\mathcal{V}ar$ and a set of their values $\mathbb{V}$. The set of booleans will be denoted by $\mathcal{B}ool = \{\mathsf{True}, \mathsf{False}\}$. The language includes at least the following:

1. simple statements $\mathcal{S}tmt$, such as:
   - empty instruction `pass`,
   - constant to variable assignment: $v \leftarrow c$, where $v \in \mathcal{V}ar$ and $c \in \mathbb{V}$,

- variable to variable assignment: $v_r \leftarrow v$, where $v_r, v \in \mathit{Var}$,
- assignment of a result of a unary operation `unop`:
  $v_r \leftarrow$ `unop` v,
- assignment of a result of a binary operation `binop`:
  $v_r \leftarrow v_1$ `binop` v_2,

2. boolean predicates *Pred* that include constants `true` and `false`, unary and binary predicates $\phi(v_1)$ and $\psi(v_1, v_2)$, for $v_1, v_2 \in \mathit{Var}$,
3. control statements *Ctrl*, including:
   - conditional statement: `test P Label1 Label2`, where `P` is a boolean predicate and `Label1` and `Label2` are labels of simple statements,
   - unconditional jump statement: `goto Label` that can be used to implement a loop or sequence of simple statements,

In this language, unary and binary operations may occur only in an assignment, thus, slightly abusing the terminology, we call the whole assignment (e.g. $v_r \leftarrow v_1$ `binop` v_2) a binary (or unary) operation.

We assume that all simple statements are uniquely labelled with labels from a set *Label*. The entry point of the program is labelled with a special label Start, while the (unique) exit point is labelled with End. We assume that Start never occurs as a target of any control statement (in other words, there is no cycle in the program that contains Start). We also assume that the entry and exit points are distinct. A program `prog` $\in \mathit{Prog}$ is a collection of triples (label, simple statement, control statement). At the exit point End there are some special, empty simple and control statements (the execution does not move beyond the exit point). The set *State* of program states consists of valuations $\rho\colon \mathit{Var} \to \mathbb{V}$ of the scalar variables. The set $\mathit{InitState} \subseteq \mathit{State}$ denotes possible input states.

The transfer function $t\colon \mathit{Stmt} \times \mathit{State} \to \mathit{State}$ for the simple statements, $p\colon \mathit{Pred} \times \mathit{State} \to \mathit{Bool}$ for boolean predicates and $t_c\colon \mathit{Ctrl} \times \mathit{State} \to \mathit{Label}$ for the control statements are given in a standard way (see Figure 1.1). We assume here that we are given a function $f_{\mathtt{unop}}\colon \mathbb{V} \to \mathbb{V}$ for each unary operator `unop` and $f_{\mathtt{binop}}\colon \mathbb{V} \times \mathbb{V} \to \mathbb{V}$ for each binary operator `binop`. Likewise, we require $\varphi_{\mathtt{U}}\colon \mathbb{V} \to \mathit{Bool}$ for each unary predicate `U` and $\varphi_{\mathtt{B}}\colon \mathbb{V} \times \mathbb{V} \to \mathit{Bool}$ for each binary predicate `B`.

We define the semantics $t_s\colon \mathit{Prog} \times \mathit{Label} \times \mathit{State} \to \mathit{Label} \times \mathit{State}$ of a single step of the execution of a program, a relation $t_s^* \subseteq \mathit{Prog} \times \mathit{Label} \times \mathit{State} \times \mathit{Label} \times \mathit{State}$ (which is a transitive closure of t_s) and

$$t(\mathtt{pass},\rho)=\rho \quad t(v\leftarrow \mathtt{c},\rho)=\rho[v\mapsto \mathtt{c}] \quad t(v_r\leftarrow v,\rho)=\rho[v_r\mapsto\rho(v)]$$
$$t(v_r\leftarrow \mathtt{unop}\ v,\rho)=\rho[v_r\mapsto f_{\mathtt{unop}}(\rho(v))]$$
$$t(v_r\leftarrow v_1\ \mathtt{binop}\ v_2,\rho)=\rho[v_r\mapsto f_{\mathtt{binop}}(\rho(v_1),\rho(v_2))]$$

$$p(\mathtt{true},\rho)=\mathsf{True} \qquad p(\mathtt{false},\rho)=\mathsf{False}$$
$$p(\mathsf{U}(v),\rho)=\varphi_{\mathsf{U}}(\rho(v)) \quad p(\mathsf{B}(v_1,v_2),\rho)=\varphi_{\mathsf{B}}(\rho(v_1),\rho(v_2))$$

$$t_c(\mathtt{goto}\ \mathsf{L},\rho)=\mathsf{L} \quad t_c(\mathtt{test}\ \mathsf{P}\,\mathsf{L}_1\,\mathsf{L}_2,\rho)=\begin{cases}\mathsf{L}_1 & p(\mathsf{P},\rho)=\mathsf{True}\\ \mathsf{L}_2 & p(\mathsf{P},\rho)=\mathsf{False}\end{cases}$$

Figure 1.1: Semantics of statements and predicates

$$\frac{(\mathsf{L},\mathsf{stmt},\mathsf{ctrl})\in\mathsf{Prog} \quad t(\mathsf{stmt},\rho)=\rho' \quad t_c(\mathsf{ctrl},\rho')=\mathsf{L}'}{t_s(\mathsf{Prog},\mathsf{L},\rho)=(\mathsf{L}',\rho')}$$

$$\frac{t_s(\mathsf{Prog},\mathsf{L},\rho)=(\mathsf{L}',\rho')}{t_s^*(\mathsf{Prog},\mathsf{L},\rho,\mathsf{L}',\rho')} \quad \frac{t_s^*(\mathsf{Prog},\mathsf{L}_1,\rho,\mathsf{L}',\rho') \quad t_s^*(\mathsf{Prog},\mathsf{L}',\rho',\mathsf{L}_2,\rho'')}{t_s^*(\mathsf{Prog},\mathsf{L}_1,\rho,\mathsf{L}_2,\rho'')}$$

$$\frac{t_s^*(\mathsf{Prog},\mathsf{Start},\rho,\mathsf{End},\rho')}{t_p(\mathsf{Prog},\rho)=\rho'}$$

Figure 1.2: Semantics of a single step of program execution (t_s), its closure t_s^* and program semantics t_p

finally the semantics of a program $t_p\colon \mathit{Prog}\times\mathit{State}\rightharpoonup\mathit{State}$ as shown in Figure 1.2 (since for no ρ',L', $t_s^*(\mathsf{Prog},\mathsf{End},\rho,\mathsf{L}',\rho')$, it is easy to see that t_p is a partial function indeed).

We will extend this simple language in further chapters, for instance adding arrays and dictionaries.

1.2.2 *Static Semantics*

To prove some properties of programs, it is often sufficient to consider sets of states which may occur at a program point in any execution, instead of analysing each execution separately [25]. Instead of individual states $\rho\colon \mathit{Var}\to\mathbb{V}$ we consider now *contexts* (sets of states) $c\in\mathit{Ctx}=\mathcal{P}(\mathit{State})$. We are particularly interested in contexts that consist of states reachable by the program. We define a *program sum-*

mary context $s_L \in \mathcal{Ctx}$ associated with a program point labelled with $L \in \mathcal{Label}$ as:

$$s_L \triangleq \{\rho \mid \exists_{\rho_s \in \mathit{InitState}}\ t_s^*(\mathsf{Start}, \rho_s, L, \rho)\}\ .$$

A *context vector* $\vec{c} \in \overrightarrow{\mathcal{Ctx}}$ is a mapping $\mathcal{Label} \to \mathcal{Ctx}$ that assigns a context to each program point. The context vector that consists of all program summary contexts, i.e.

$$\vec{s} \triangleq \lambda L.s_L \tag{1.6}$$

describes all states that may occur in any program execution at any program point and will be called the *static program summary*.

The set $\overrightarrow{\mathcal{Ctx}}$ of all context vectors forms a complete lattice under the join $\vec{\sqcup}$ such that $\vec{c}_1 \vec{\sqcup} \vec{c}_2 \triangleq \lambda L.\vec{c}_1(L) \cup \vec{c}_2(L)$. The transfer function t_s can be extended to contexts. We define $\mathcal{T}\colon \mathcal{Label} \times \overrightarrow{\mathcal{Ctx}} \to \mathcal{Ctx}$ by:

$$\mathcal{T}(L, \vec{c}) \triangleq \begin{cases} \mathit{InitState} & L = \mathsf{Start}, \\ \bigcup\{\rho \mid L' \in \mathcal{Label},\ \rho' \in \vec{c}(L'),\ t_s(L', \rho') = (L, \rho)\} & \text{otherwise.} \end{cases}$$

Intuitively, $\mathcal{T}(L, \vec{c})$ is a context that consists of all states that may be reached at the program point L in a single step of the execution, starting from some state in $\vec{c}$.

For each label $L \in \mathcal{Label}$ the static program summary $\vec{s}$ given by (1.6) satisfies $\vec{s}(L) = \mathcal{T}(L, \vec{s})$. We may define the *static transfer function* $\vec{\mathcal{T}} : \overrightarrow{\mathcal{Ctx}} \to \overrightarrow{\mathcal{Ctx}}$ by $\vec{\mathcal{T}}(\vec{c}) = \lambda L.\mathcal{T}(L, \vec{c})$. It is easy to see that the static transfer function $\vec{\mathcal{T}}$ for a given program is order preserving (with respect to the order on context vectors given by $\vec{c}_1 \sqsubseteq \vec{c}_2 \Leftrightarrow \forall_{L \in \mathcal{Label}} \vec{c}_1(L) \vec{\sqsubseteq} \vec{c}_2(L)$), thus it has fixpoints [65] and the static program summary $\vec{s} \in \overrightarrow{\mathcal{Ctx}}$ is it's least fixpoint.

The static program summary (the context vector $\vec{s}$) may be not feasibly computable. The idea of abstract interpretation is to (soundly) over-approximate this vector.

1.2.3 *Framework Definition*

The core of the abstract interpretation is an *abstract domain* that is formalised as a tuple

$$A = \langle \mathcal{A}, \sqcup_a, \sqcap_a, \top_a, \bot_a, \gamma_a, \alpha_a, \delta_a, \pi_a \rangle\ .$$

In this setting $\mathcal{A}$ denotes some set. We refer to its elements as *abstract states* (or, sometimes, as *abstract contexts*). It forms a lattice under the join ($\sqcup_a$) and meet ($\sqcap_a$) operators. The lattice $\langle \mathcal{A}, \sqcup_a, \sqcap_a \rangle$ is called the *carrier* of the domain. Constants $\top_a \in \mathcal{A}$ and $\bot_a \in \mathcal{A}$ denote the top and bottom in this lattice, i.e. for each $a \in \mathcal{A}$, $a \sqcup_a \top_a = \top_a$ and $a \sqcap_a \bot_a = \bot_a$. Sometimes we use also the order $\sqsubseteq_a$ on $\mathcal{A}$ uniquely defined as $a \sqsubseteq_a b \Leftrightarrow a \sqcup_a b = b$.

The *concretisation* $\gamma_a \colon \mathcal{A} \to \mathcal{Ctx}$ and *abstraction* $\alpha_a \colon \mathcal{Ctx} \to \mathcal{A}$ functions must be order-preserving and fulfil the following consistency conditions:

- $\forall_{c \in \mathcal{Ctx}}\, c \subseteq \gamma_a(\alpha_a(c))$,
- $\forall_{a \in \mathcal{A}}, \alpha_a(\gamma_a(a)) \sqsubseteq_a a$.

It is now easy to see that γ_a and α_a form a Galois connection

$$\langle \mathcal{Ctx}, \cup, \cap \rangle \underset{\alpha_a}{\overset{\gamma_a}{\leftrightarrows}} \langle \mathcal{A}, \sqcup_a, \sqcap_a \rangle$$

thus, if $\langle \mathcal{Ctx}, \cup, \cap \rangle$ and $\langle \mathcal{A}, \sqcup_a, \sqcap_a \rangle$ are complete lattices, then it is sufficient to define explicitly only one of the concretisation and abstraction functions.

We also additionally assume that $\gamma_a(\bot_a) = \emptyset$. This requirement is easy to fulfil, as one can always add an "artificial" bottom element to $\mathcal{A}$ so that $\langle \mathcal{A}, \sqcap_a, \sqcup_a \rangle$ remains complete.

The function $\delta_a \colon \mathcal{Stmt} \times \mathcal{A} \to \mathcal{A}$ is an *abstract transfer function*, while $\pi_a \colon \mathcal{Pred} \times \mathcal{A} \to \mathcal{A} \times \mathcal{A}$ is the *abstract predicate semantics*. Unlike in a concrete execution, a boolean predicate does not evaluate in the abstract domain to a single boolean value. Instead, it returns a pair of abstract states. Intuitively the first one describes a part of the input abstract state for which the predicate holds, while the second — the part of the abstract state for which it does not hold. Both δ_a and π_a are monotone, i.e. for each statement $\mathtt{stmt} \in \mathcal{Stmt}$, predicate $P \in \mathcal{Pred}$ and for every two abstract contexts $a_1, a_2 \in \mathcal{A}$ if $a_1 \sqsubseteq_a a_2$ then:

1. $\delta_a(\mathtt{stmt}, a_1) \sqsubseteq_a \delta_a(\mathtt{stmt}, a_2)$,
2. if $\pi_a(P, a_1) = (a_{t1}, a_{f1})$ and $\pi_a(P, a_2) = (a_{t2}, a_{f2})$ then $a_{t1} \sqsubseteq_a a_{t2}$ and $a_{f1} \sqsubseteq_a a_{f2}$.

The *abstract control function* $\delta_a^c \colon \mathcal{Ctrl} \times \mathcal{A} \to (\mathcal{Label} \rightharpoonup \mathcal{A})$ does not need to be provided as a part of the abstract interpretation A. One can define it in a generic way using only π_a and $\mathcal{A}$. The abstract control function, unlike the concrete one, does not return a single label, but a partial mapping from labels to abstract contexts. This is caused by the definition of the abstract predicate semantics. The abstract control function δ^c_a for the simple language introduced in Section 1.2.1 is given by two rules:

$$\frac{\pi_a(P, a) = (a_t, a_f)}{\delta^c_a(\texttt{test P L1 L2}, a) = [L1 \mapsto a_t, L2 \mapsto a_f]}$$

and

$$\delta^c_a(\texttt{goto L}, a) = [L \mapsto a] \ .$$

We may define now the semantics $\delta^s_a \colon \mathcal{Label} \times \mathcal{A} \to (\mathcal{Label} \rightharpoonup \mathcal{A})$ of a single step of an abstract execution as

$$\frac{(L, \mathrm{stmt}, \mathrm{ctrl}) \in \mathrm{Prog} \quad \delta_a(\mathrm{stmt}, a) = a'}{\delta^s_a(L, a) = \delta^c_a(\mathrm{ctrl}, a')} \ .$$

Similarly as in the static semantics, we introduce *abstract context vectors* $\overrightarrow{\mathcal{A}} = \mathcal{Label} \to \mathcal{A}$ that assign an abstract context to each program point. The lattice operations over $\mathcal{A}$ can be extended to $\overrightarrow{\mathcal{A}}$ point-wise. We denote the order on $\overrightarrow{\mathcal{A}}$ by $\overrightarrow{\sqsubseteq_a}$. Below we identify any partial mapping $f \colon \mathcal{Label} \rightharpoonup \mathcal{A}$ with an abstract context vector $\vec{c}_f \in \overrightarrow{\mathcal{A}}$ that extends f by assigning $\bot_a$ to all labels not in the domain of f. We extend now δ^s_a to $\overrightarrow{\delta_a} \colon \overrightarrow{\mathcal{A}} \to \overrightarrow{\mathcal{A}}$ by defining $\overrightarrow{\delta_a}(\vec{a})$ for $\vec{a} \in \overrightarrow{\mathcal{A}}$ as

$$\lambda L. \begin{cases} \alpha_a(\mathit{InitState}) & L = \mathrm{Start}, \\ \bigsqcup_a \{a \mid L' \in \mathcal{Label},\ a' = \vec{a}(L'),\ a = \delta^s_a(a', L')(L)\} & L \neq \mathrm{Start} \end{cases} \tag{1.7}$$

It is now easy to see that $\overrightarrow{\delta_a}$ is order preserving, hence it has fixpoints. In the abstract interpretation we are interested in finding the *abstract program summary* $\vec{s}_a \in \overrightarrow{\mathcal{A}}$, which is the least fixpoint of $\overrightarrow{\delta_a}$.

We may lift the abstraction and concretisation functions to context vectors:

- $\overrightarrow{\alpha_a} \colon \overrightarrow{\mathit{Ctx}} \to \overrightarrow{\mathcal{A}}$ defined as $\overrightarrow{\alpha_a}(\vec{c}) \triangleq \lambda L.\alpha_a(\vec{c}(L))$,
- $\overrightarrow{\gamma_a} \colon \overrightarrow{\mathcal{A}} \to \overrightarrow{\mathit{Ctx}}$ given by $\overrightarrow{\gamma_a}(\vec{a}) \triangleq \lambda L.\gamma_a(\vec{a}(L))$.

The consistency conditions of α_a and γ_a are also lifted to $\overrightarrow{\alpha_a}$ and $\overrightarrow{\gamma_a}$, hence $\overrightarrow{\alpha_a}$ and $\overrightarrow{\gamma_a}$ form a Galois connection

$$\langle \overrightarrow{\mathit{Ctx}}, \sqsubseteq \rangle \underset{\overrightarrow{\alpha_a}}{\overset{\overrightarrow{\gamma_a}}{\leftrightarrows}} \langle \overrightarrow{\mathcal{A}}, \overrightarrow{\sqsubseteq_a} \rangle \ .$$

The abstract transfer function $\overrightarrow{\delta_a}$ must be a sound abstraction of the static transfer function $\overrightarrow{\mathcal{T}}$ (with respect to the Galois connection as above). Recall that according to (1.1) this is formalised as:

$$\forall_{\vec{c} \in \overrightarrow{\mathit{Ctx}}}\ \overrightarrow{\mathcal{T}}(\vec{c}) \sqsubseteq (\overrightarrow{\gamma_a} \circ \overrightarrow{\delta_a} \circ \overrightarrow{\alpha_a})(\vec{c}) \ .$$

$$\begin{tikzcd}
c \arrow[r, "t"] \arrow[d, "\alpha_a"'] & c_1 \subseteq c_2 \\
a \arrow[r, "\delta_a"'] & a_1 \arrow[u, "\gamma_a"']
\end{tikzcd}$$

Figure 1.3: Abstraction, concretisation and transfer function

This global condition can be restated locally, using t, p, δ_a and π_a. Let $c \in \mathcal{Ctx}$, $\mathtt{stmt} \in \mathcal{Stmt}$ and $\mathtt{pred} \in \mathcal{Pred}$ denote arbitrary context, simple statement and predicate, respectively. Let $(a_{\mathsf{True}}, a_{\mathsf{False}}) = \pi_a(\alpha_a(c), \mathtt{pred})$. The above global sound abstraction condition follows from (and in fact is equivalent to) the conjunction of the following conditions:

1. $\{\sigma \mid \rho \in c,\ t(\mathtt{stmt}, \rho) = \sigma\} \subseteq \gamma_a(\delta_a(\mathtt{stmt}, \alpha_a(c)))$,
2. $\{\rho \mid \rho \in c,\ p(\mathtt{pred}, \rho) = \mathsf{True}\} \subseteq \gamma_a(a_{\mathsf{True}})$,
3. $\{\rho \mid \rho \in c,\ p(\mathtt{pred}, \rho) = \mathsf{False}\} \subseteq \gamma_a(a_{\mathsf{False}})$.

The correlation between the transfer, concretisation and abstraction functions expressed in the first condition above is informally depicted in Figure 1.3.

By Theorem 1.5, the least fixpoint of the abstract transfer function $\overrightarrow{\delta_a}$ (the abstract program summary $\vec{s_a} \in \overrightarrow{\mathcal{A}}$) is a safe over-approximation of the least fixpoint of the static transfer function $\overrightarrow{\mathcal{T}}$ (the static program summary $\vec{s} \in \overrightarrow{\mathcal{Ctx}}$) i.e. $\vec{s} \sqsubseteq \overrightarrow{\gamma_a}(\vec{s_a})$. Moreover, each $\vec{a} \in \overrightarrow{\mathcal{A}}$ such that $\vec{s_a} \mathrel{\overrightarrow{\sqsubseteq_a}} \vec{a}$ is also a safe over-approximation of $\vec{s}$ (however, $\vec{s_a}$ is its best over-approximation).

1.2.4 *Computing the Abstract Interpretation*

We have defined the result of the abstract interpretation (the abstract program summary $\vec{s_a} \in \overrightarrow{\mathcal{A}}$) as the least fixpoint of $\overrightarrow{\delta_a}$, which (by Theorem 1.4) is equal to $\overrightarrow{\sqcup_a}\langle \overrightarrow{\delta_a}^{i}(\lambda L.\bot_a), i \in \mu\rangle$, where μ is an ordinal of cardinality greater than the cardinality of $\overrightarrow{\mathcal{A}}$. If the height of the lattice $\langle \mathcal{A}, \sqcup_a, \sqcap_a\rangle$ is finite, then the fixpoint is feasibly computable. However, when the lattice is of infinite height, then the iteration sequence

$\langle \overrightarrow{\delta_a}^{i}(\lambda L.\bot_a), i \in \mu\rangle$ does not need to stabilise in finitely many steps. In this case, one has to find some efficiently computable $\vec{a} \in \overrightarrow{\mathcal{A}}$ such that $\vec{s_a} \overrightarrow{\sqsubseteq_a} \vec{a}$.

WIDENING A common approach to find $\vec{a} \in \overrightarrow{\mathcal{A}}$ such that $\vec{s_a} \overrightarrow{\sqsubseteq_a} \vec{a}$, is to replace the abstract transfer function $\overrightarrow{\delta_a}$ with some $\widehat{\overrightarrow{\delta_a}}$, which:

- has efficiently computable fixpoints,
- each fixpoint of $\widehat{\overrightarrow{\delta_a}}$ is a post-fixpoint of $\overrightarrow{\delta_a}$.

An approach to construct the modified abstract transfer function $\widehat{\overrightarrow{\delta_a}}$, is to equip the abstract domain with a *widening operator* $\nabla_a \colon \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ [12] that obeys the following properties:

1. $\forall_{a,b\in\mathcal{A}}\ a \sqcup_a b \sqsubseteq_a a \nabla_a b$ (the widening over-approximates the join),
2. for every infinite sequence $c_0, c_1, \dots$ of abstract contexts, the infinite sequence $b_0, b_1, \dots$ given by

$$\begin{cases} b_0 = c_0, \\ b_n = b_{n-1} \nabla_a c_n \end{cases}$$

   is not strictly increasing.

If the height of the lattice $\langle \mathcal{A}, \sqcap_a, \sqcup_a\rangle$ is finite, then the widening is presumed to coincide with $\sqcup_a$ (unless otherwise explicitly defined).

The number of program points (labels) in a program is finite. Hence in every infinite strictly increasing sequence of abstract context vectors $\vec{a}_0, \vec{a}_1, \dots$ there must be some label $L \in \mathcal{Label}$ for which the sequence of abstract contexts $\vec{a}_0(L), \vec{a}_1(L), \dots$ never stabilises. In case of sequences obtained as a result of iteration of the abstract transfer function $\overrightarrow{\delta_a}$ (which are considered in the abstract analysis), this means that in the control flow graph of the program there must be a cycle and either L belongs to this cycle or the abstract context associated with L depends on some $\vec{a}(L')$ such that L' belongs to the cycle and the sequence $\vec{a}_0(L'), \vec{a}_1(L'), \dots$ never stabilises either. Using the widening operator we may ensure that all abstract contexts on each cycle stabilise.

This is achieved by defining a set of widening program points $\mathcal{W}$. It should be chosen as such subset of the labels that each cycle in the control flow graph of the program contains at least one widening point.

Now, the abstract transfer function $\overrightarrow{\delta_a}$ is replaced with $\widehat{\overrightarrow{\delta_a}}$ that is given by

$$\widehat{\overrightarrow{\delta_a}}(\vec{a}) \triangleq \lambda L. \begin{cases} \vec{a}(L) \mathbin{\nabla_a} \overrightarrow{\delta_a}(\vec{a})(L) & \text{if } L \in \mathcal{W}, \\ \overrightarrow{\delta_a}(\vec{a})(L) & \text{otherwise.} \end{cases}$$

Kleene's sequence $\vec{a}_0, \vec{a}_1, \ldots$ such that $\vec{a}_i = (\widehat{\overrightarrow{\delta_a}})^i(\bot_a)$ cannot be strictly increasing.

Suppose that Kleene's sequence is strictly increasing and let $L_w \in \mathcal{W}$ be chosen such that $\vec{a}_0(L_w), \vec{a}_1(L_w), \ldots$ is strictly increasing as well. Consider the sequence of abstract context vectors

$$\begin{cases} \vec{a}'_0 \triangleq \vec{a}_0, \\ \vec{a}'_{i+1} \triangleq \overrightarrow{\delta_a}(\vec{a}_i). \end{cases}$$

The widening sequence $b_0 \triangleq \vec{a}'_0(L_w)$, $b_{i+1} \triangleq b_i \mathbin{\nabla_a} \vec{a}'_i(L_w)$ would be strictly increasing as well (as $b_i = \vec{a}_i(L_w)$ for $i \in \mathbb{N}$), which would contradict the definition of a widening operator.

1.2.5 *Variable Introduction and Elimination*

Sometimes, instead of assuming that the set of variables $\mathcal{V}ar$ is fixed, it is more convenient to allow adding new variables and removing existing ones during program execution. In many programming languages one can define variables in arbitrary places in the code. Variables defined within blocks are not visible and cannot be accessed outside these blocks.

In static analysis we can first identify all variables used by the program and then assume that all of them are present during the whole execution. However, for efficiency reasons it is often better to add new variables when they are used for the first time and remove old ones, when they cannot be accessed.

We parametrise the abstract domain with the set of variables that it models. We write $A(V)$ to indicate that the abstract domain models the variables from set V (the set of abstract elements of $A(V)$ will be denoted as $\mathcal{A}(V)$). We also often say that the domain A is *over the set* V. We omit the set V (and write just A), when it is clear from the context.

We introduce now two additional operations on the abstract domain, namely *variable introduction* $\Uparrow_v \colon \mathcal{A}(V) \to \mathcal{A}(V \cup \{v\})$ and *variable elimination* $\Downarrow_v \colon \mathcal{A}(V) \to \mathcal{A}(V \setminus \{v\})$ that fulfil the following conditions:

1. for each $a \in \mathcal{A}(V)$, $\rho|_V \in \gamma_a(a) \Rightarrow \rho \in \gamma_a(a{\uparrow_v})$. If additionally $\rho \in \gamma_a(a{\uparrow_v}) \Rightarrow \rho|_V \in \gamma_a(a)$, then the introduction is *exact*.

2. for each $a \in \mathcal{A}(V)$, $\rho \in \gamma_a(a) \Rightarrow \rho|_{V\setminus\{v\}} \in \gamma_a(a{\downarrow_v})$. The elimination is exact, if additionally $\sigma \in \gamma_a(a{\downarrow_v}) \Rightarrow \sigma \in \{\rho|_{V\setminus\{v\}} \mid \rho \in \gamma_a(a)\}$.

The variable elimination $\downarrow_v$ and introduction $\uparrow_v$ define a *forget operator* $\updownarrow_v \mathcal{A}(V) \to \mathcal{A}(V)$ such that $a{\updownarrow_v} \triangleq a{\downarrow_v}{\uparrow_v}$. It is easy to see that $\gamma_a(a) \subseteq \gamma_a(a{\updownarrow_v})$.

1.2.6 *Domain Conversion*

Given two abstract domains A and B, a monotone function $\kappa_{A\to B} \colon \mathcal{A} \to \mathcal{B}$ is called a *conversion* between the domains A and B if for each $a \in \mathcal{A}$, $\gamma_a(a) \subseteq \gamma_b(\kappa_{A\to B}(a))$. The conversion $\kappa_{A\to B}$ is *exact* if $\gamma_a(a) = \gamma_b(\kappa_{A\to B}(a))$. Note that $\alpha_b \circ \gamma_a$ is a conversion between A and B.

The domain conversion is often used to define the transfer function. Let us assume that we are given some abstract domain A. For a newly developed domain B the transfer function $\delta_b(i, b)$ can be defined as

$$\delta_b(i, b) \triangleq \kappa_{A\to B}\big(\delta_a(i, \kappa_{B\to A}(b))\big) \ .$$

This definition of δ_b is correct, i.e. it is a sound transfer function for the domain B, whenever δ_a is sound.

The variable introduction $\uparrow_v$ and elimination $\downarrow_v$ defined above are conversions between $A(V)$ and $A(V \cup \{v\})$ and between $A(V)$ and $A(V \setminus \{v\})$, respectively.

1.2.7 *Notation*

We continue using the abstract domain naming convention introduced in this section. And so, an abstract domain is denoted by a capital Latin letter (for example D). The set of abstract elements will be written in italic font (e.g. $\mathcal{D}$) and each domain operation will be subscripted with the corresponding lowercase letter (for instance $\sqcup_d$ for the join and δ_d for the transfer function).

1.2.8 *Alternative Framework Definition*

The definition of the abstract interpretation framework presented above requires existence of the abstraction and concretisation functions that form a Galois connection. In this setting each concrete context has its best abstraction. In some cases the requirements imposed so far, are too strong, e.g. it often happens that the abstract domain forms a lattice that is not complete.

There exist alternative, weaker formalisations of the abstract interpretation framework [15] that can be used when no Galois connection can be established between the concrete and abstract domains. One of the approaches is to work only with the concretisation function γ_a (*concretisation-based abstraction*). In this case, an abstract element $a \in \mathcal{A}$ is an abstraction of a concrete context $c \in \mathcal{Ctx}$, if $c \subseteq \gamma_a(a)$.

The notion of a sound abstraction needs to be reformulated. For two lattices $\langle X, \sqsubseteq\rangle$ and $\langle Y \preceq\rangle$, a monotone function $\gamma\colon Y \to X$ and a function $f\colon X \to X$, we say that $g\colon Y \to Y$ is a sound abstraction of f if for each $y \in Y$:

$$(f \circ \gamma)(y) \sqsubseteq (\gamma \circ g)(y)\ .$$

If the abstract transfer function $\overrightarrow{\delta_a}$ is a sound approximation (in the sense as above) of the static transfer function $\overrightarrow{\mathcal{T}}$, then the least fixpoint of $\widehat{\overrightarrow{\delta_a}}$ (i.e. of the abstract transfer function with widening) over-approximates the static program summary, i.e. $\vec{s} \sqsubseteq \overrightarrow{\gamma_a}(\mathrm{lfp}\ \widehat{\overrightarrow{\delta_a}})$ [16].

1.2.9 *Example*

We illustrate now the formal background of the abstract interpretation using a very simple example. We present an abstraction of numerical variables using only their signs [62].

The domain of signs is a tuple $S = \langle \mathcal{S}, \sqcup_s, \sqcap_s, \top_s, \bot_s, \gamma_s, \alpha_s, \delta_s, \pi_s\rangle$. The set of abstract states is chosen as $\mathcal{S} = \mathcal{Var} \to \mathcal{P}(\{-, 0, +\})$. The join $\sqcup_s$ and meet $\sqcap_s$ are just point-wise set union and intersection, respectively. The top $\top_s$ admits for each variable any value, i.e. $\top_s = \lambda v.\{-, 0, +\}$, while the bottom is given by $\bot_s = \lambda v.\emptyset$.

Let $\mathtt{sign}\colon \mathbb{V} \to \{-, 0, +\}$ be the obvious function that returns the sign of numerical values. The concretisation function γ_s is given by

$$\gamma_s(s) \triangleq \{\rho \mid \forall_{v \in \mathcal{Var}}\ \mathtt{sign}(\rho(v)) \in s(v)\}\ .$$

The abstraction of a context c admits for each variable all signs of this variable in any valuation $\rho \in c$:

$$\alpha_s(c) \triangleq \lambda v.\big\{\mathtt{sign}(\rho(v)) \mid \rho \in c\big\}\ .$$

To give some intuitions about the transfer function we show the transfer rule for an assignment $v \leftarrow x + y$. We start with defining an auxiliary operation PLUS: $\{-,0,+\} \times \{-,0,+\} \to \mathcal{P}(\{-,0,+\})$, given by the following matrix:

PLUS	$-$	0	$+$
$-$	$\{-\}$	$\{-\}$	$\{-,0,+\}$
0	$\{-\}$	$\{0\}$	$\{+\}$
$+$	$\{-,0,+\}$	$\{+\}$	$\{+\}$

The abstract transfer rule for $v \leftarrow x+y$ is now given by:

$$\delta_s(s, v \leftarrow x+y) \triangleq s[v \mapsto \bigcup_{i \in s(x), j \in s(y)} \mathrm{PLUS}(i,j)] \ .$$

The transfer rules for all other basic arithmetic operations can be defined in the same fashion.

We illustrate π_s on a predicate $v < 0$, i.e. we show the definition of $\pi_s(s, v < 0) \triangleq (s_{\mathsf{True}}, s_{\mathsf{False}})$. The two abstract states s_{True} and s_{False} are given by:

$$s_{\mathsf{True}} \triangleq s[v \mapsto s(v) \cap \{-\}] \qquad \text{and} \qquad s_{\mathsf{False}} \triangleq s[v \mapsto s(v) \cap \{0,+\}] \ .$$

The domain is finite, hence no widening is required.

The variable introduction and variable elimination are trivial, i.e. if $s \in \mathcal{S}(V)$, then $s{\downarrow_v} \triangleq s|_{V \setminus \{v\}}$, while $s{\uparrow_v}(x) \triangleq s(x)$ for $x \in V$ and $s{\uparrow_v}(v) \triangleq \{-,0,+\}$ (note that both the elimination and introduction are exact).

We may demonstrate now how the abstract interpretation using signs works on a simple code fragment. The example shown in Figure 1.4 is written in a pseudocode that includes `if` and `while` statements, however it can be rewritten in the standard way in the language from Section 1.2.1. We favour the pseudocode syntax, as it may be more convenient for the reader.

The static context $c \in \mathcal{C}tx$ attached to each program point is a set of variable valuations. For simplicity, we use in the example only two variables x and y. The corresponding abstract context $s \in \mathcal{S}$ maps each variable to the set of its possible signs.

Let us assume that in the program in Figure 1.4 there are two possible input states $\rho_1 = [x \mapsto 0, y \mapsto -4]$ and $\rho_2 = [x \mapsto 2, y \mapsto 7]$ (which gives a context $c_1 = \mathit{InitState} = \{\rho_1, \rho_2\}$). According to the abstraction α_s in the domain of signs, the corresponding initial abstract context is $s_1 = [x \mapsto \{0,+\}, y \mapsto \{-,+\}]$. The subsequent assignment $x \leftarrow x+1$ results in the abstract domain in an abstract context $s_2 = [x \mapsto \{+\}, y \mapsto \{-,+\}]$. The predicate $y < 0$ evaluates in the abstract state s_2 to the pair

$$\pi_s(s_2, y < 0) = (s_{\mathsf{True}} = [x \mapsto \{+\}, y \mapsto \{-\}], s_{\mathsf{False}} = [x \mapsto \{+\}, y \mapsto \{+\}]) \ .$$

```
$c_1 = \{[x \mapsto 0, y \mapsto -4], [x \mapsto 2, y \mapsto 7]\}, s_1 = [x \mapsto \{0,+\}, y \mapsto \{-,+\}]$
$x \leftarrow x + 1$
$c_2 = \{[x \mapsto 1, y \mapsto -4], [x \mapsto 3, y \mapsto 7]\}, s_2 = [x \mapsto \{+\}, y \mapsto \{-,+\}]$
if $y < 0$ then
    $c_3 = \{[x \mapsto 1, y \mapsto -4]\}, s_3 = [x \mapsto \{+\}, y \mapsto \{-\}]$
    $y \leftarrow -y$
    $c_4 = \{[x \mapsto 1, y \mapsto 4]\}, s_4 = [x \mapsto \{+\}, y \mapsto \{+\}]$
end if
$c_5 = \{[x \mapsto 1, y \mapsto 4], [x \mapsto 3, y \mapsto 7]\}, s_5 = [x \mapsto \{+\}, y \mapsto \{+\}]$
while someCondition do
    $c_6 = \{\rho_k \mid k \in \mathbb{N}, \rho_k = [x \mapsto 1 + k, y \mapsto 4 + 2k]$ or $\rho_k = [x \mapsto 3 + k, y \mapsto 7 + 2k]\},$
    $s_6 = [x \mapsto \{+\}, y \mapsto \{+\}]$
    $x \leftarrow x + 1$
    $y \leftarrow y + 2$
end while
$x \leftarrow y - x$
$c_7 = \{\rho_k \mid k \in \mathbb{N}, \rho_k = [x \mapsto 3 + k, y \mapsto 4 + 2k]$ or $\rho_k = [x \mapsto 4 + k, y \mapsto 7 + 2k]\},$
$s_7 = [x \mapsto \{-,0,+\}, y \mapsto \{+\}]$
```

Figure 1.4: Analysis using the domain of signs

Let us now focus on the abstract context s_5, which is right after the `if` statement. According to the definition of $\overrightarrow{\delta_a}$ given by (1.7), it is a join of the outputs of all preceding statements, i.e. it is $s_5 = s_{\mathsf{False}} \sqcup_s s_4$.

The `while` loop iterates arbitrarily many times (we assume that the loop guard $someCondition$ is environment-dependent, thus not statically analysable). However, in the abstract domain the simulation stabilises after the first iteration, with a fixpoint $[x \mapsto \{+\}, y \mapsto \{+\}]$. The corresponding static context at the loop entry

$$c_6 = \{\rho_k \mid k \in \mathbb{N}, \rho_k = [x \mapsto 1 + k, y \mapsto 4 + 2k] \text{ or } \rho_k = [x \mapsto 3 + k, y \mapsto 7 + 2k]\}$$

cannot be computed using a simple iteration. Finally, at the end of the procedure, the abstract context $s_7 = [x \mapsto \{-,0,+\}, y \mapsto \{+\}]$ correctly captures that y must be positive in all possible executions. However, the property that x must also be positive (which holds in every state from the output context c_7), is lost (but it is safely over-approximated).

1.3 STATE OF THE ART

Abstract interpretation has been applied to verify various program properties and multiple abstract domain have been proposed. The most thoroughly studied area is the analysis of numerical variables. One of the most often mentioned numerical domains is the basic, yet very efficient domain of intervals [13]. It is non-relational, which means that it does not capture any relationships among variables (just as the above domain of signs). The most popular relational domains include the domain of pentagons [49], octagons [54] and TVPI [61]. These numerical domains are presented in more detail (as a background for our approach) in Chapter 2.

Although there is already a wide range of numerical abstract domains, certain program properties still cannot be efficiently represented. In Chapter 3 we present our new numerical domain of weighted hexagons that was designed to efficiently model relationships between numerical variables of the form $x \leqslant a \cdot y$ and $x \in [b, c]$, where $x, y \in \mathcal{V}\!ar$ are variables and $a \in \mathbb{V}$ is a non-negative and $b, c \in \mathbb{V}$ are arbitrary constants. We extend this domain to strict variants of the constraints (e.g. $x < a \cdot y$) in Chapter 4.

The abstract interpretation may be also used to model content of data containers such as arrays or dictionaries. There exist techniques for array analysis [17, 21, 32] (which are surveyed in Chapter 5 to give the background for our technique), but as far as we are aware, the problem of modelling arbitrary dictionaries has not been studied so far. To fill this gap, we have defined a generic abstract domain for modelling arrays and dictionaries. It is presented in Chapter 6.

Abstract interpretation can be also applied in many other contexts, such as for instance analysis of string variables [44, 48, 67], heap pointer structures [19, 57, 58], or program transformation [42].

Part I

ABSTRACTION OF NUMERICAL VARIABLES

2 OVERVIEW

Abstract interpretation was originally introduced to analyse properties of programs that use numerical variables. One of the most important tasks is to check the safety properties, i.e. that no error may occur during any program execution. For example, abstract interpretation can be used to verify the correctness of array accesses (that no array may be accessed out of its bounds) or to ensure that no numerical overflows may happen. To successfully reason about such properties, one needs an abstract domain in which it is possible to approximate the set of possible values of numerical expressions.

Multiple abstract domains capable of representing numerical values have been proposed. They differ in types of constraints that they discover as well as in the computational complexity. In this section we present the domains used most often and discuss their advantages and limitations.

We are interested in analysing properties of programs that use only numerical values, thus we impose the following restrictions on the simple language from the previous chapter :

1. each domain is formalised for $\mathbb{V} = \mathbb{R}$ (real numbers). At the end of each section we make it clear whether the domain can be used also to track properties within $\mathbb{Z}$ (integers) and $\mathbb{Q}$ (rational numbers),
2. the binary operations are just the arithmetic operators $-, +, \cdot$,
3. we admit one unary operator — the unary minus,
4. the only boolean predicate is the inequality test $x \leqslant y$.

We use letters v, x, y, z to denote variables and $a, b, c, \ldots$ as numerical values.

Note that in the language introduced in Section 1.2.1 all operations are total, thus we have restricted the binary operations to addition, subtraction and multiplication, deliberately skipping the division, as it is a partial function that does not fit in our language. In the second part of this thesis we show to extend both the language and abstract interpretation framework to support also partial functions.

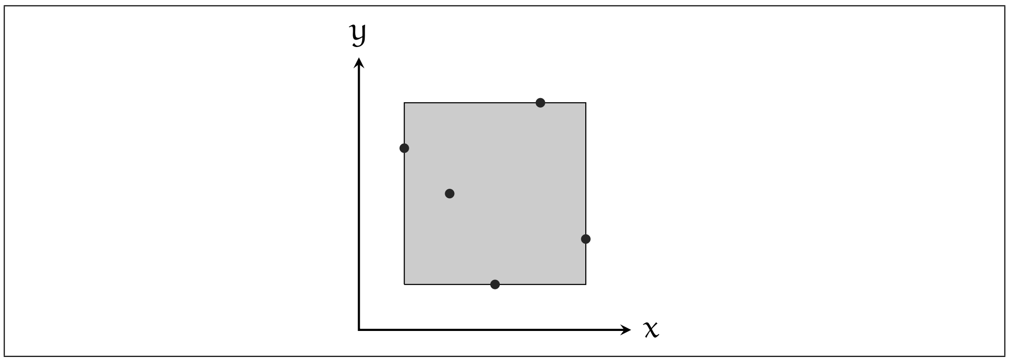

Figure 2.1: An element of the domain of intervals

For each domain presented below we precisely define the set of abstract states, the join and meet operators, as well as the widening (whenever applicable) and concretisation function. For readability, we usually only sketch the abstract transfer function, showing a couple of examples.

2.1 INTERVALS

One of the most basic and most often used numerical abstract domain is the *domain of intervals* [13]. In this domain each variable is represented by an interval that over-approximates the set of possible values of this variable at a given program point. When only two variables are considered, their values can be approximated by a rectangle (Figure 2.1).

The abstract domain of intervals is a tuple:

$$\mathrm{I} = \langle \mathcal{I}, \sqcup_i, \sqcap_i, \top_i, \bot_i, \gamma_i, \alpha_i, \delta_i, \pi_i, \triangledown_i \rangle \ .$$

Let $i\colon \mathcal{V}ar \to \mathbb{V} \cup \{-\infty\} \times \mathbb{V} \cup \{+\infty\}$ be a function that assigns to each variable a pair of numerical (or infinite) values. The order in $\mathbb{V}$ is extended so that the negative (respectively positive) infinity is smaller (resp. greater) than any element in $\mathbb{V}$.

We introduce for i two auxiliary mappings $i^{\downarrow}\colon \mathcal{V}ar \to \mathbb{V} \cup \{-\infty\}$ and $i^{\uparrow}\colon \mathcal{V}ar \to \mathbb{V} \cup \{+\infty\}$ such that

$$\forall_{x \in \mathcal{V}ar}\ (i^{\downarrow}(x), i^{\uparrow}(x)) \triangleq i(x) \ .$$

We say that a function i as above is a *correct interval environment*, if $\forall_{x \in \mathcal{V}ar} i^{\downarrow}(x) \leqslant i^{\uparrow}(x)$.

The set of abstract states $\mathcal{I}$ consists of all correct interval environments and a special element $\bot_i$. The concretisation function γ_i of an abstract state $i \in \mathcal{I}$ admits for each variable x any value $v \in [i^{\downarrow}(x), i^{\uparrow}(x)]$:

$$\gamma_i(i) \triangleq \begin{cases} \emptyset & \text{if } i = \bot_i, \\ \{\rho : \mathcal{Var} \to \mathbb{V} \mid \forall_{x \in \mathcal{Var}}\, i^{\downarrow}(x) \leqslant \rho(x) \leqslant i^{\uparrow}(x)\} & \text{otherwise.} \end{cases}$$

The abstraction α_i of a context $c \in \mathcal{Ctx}$ admits for each variable $x \in \mathcal{Var}$ the smallest interval containing all values $\rho(x)$ for all $\rho \in c$:

$$\alpha_i(c) \triangleq \begin{cases} \bot_i & \text{if } c = \emptyset, \\ \lambda x.(\inf_{\rho \in c} \rho(x), \sup_{\rho \in c} \rho(x)) & \text{otherwise.} \end{cases}$$

We proceed now with the definition of the meet ($\sqcap_i$) and join ($\sqcup_i$) operators. The join $i_1 \sqcup_i i_2$ admits for each variable x the smallest interval containing the corresponding intervals $i_1(x)$ and $i_2(x)$ from the two arguments:

$$i_1 \sqcup_i i_2 \triangleq \begin{cases} i_1 & \text{if } i_2 = \bot_i \\ i_2 & \text{if } i_1 = \bot_i \\ \lambda x.(\min(i_1^{\downarrow}(x), i_2^{\downarrow}(x)), \max(i_1^{\uparrow}(x), i_2^{\uparrow}(x))) & \text{otherwise .} \end{cases}$$

The join is always well defined, i.e. $i_1 \sqcup_i i_2$ is always a correct interval environment. The meet is defined as the intersection of the corresponding intervals:

$$i_1 \sqcap_i i_2 \triangleq \begin{cases} \tilde{i} & \text{if } i_1 \neq \bot_i,\ i_2 \neq \bot_i \text{ and } \tilde{i} \in \mathcal{I} \\ \bot_i & \text{otherwise,} \end{cases}$$

where $\tilde{i}(x) = (\max(i_1^{\downarrow}(x), i_2^{\downarrow}(x)), \min(i_1^{\uparrow}(x), i_2^{\uparrow}(x)))$.

The set of abstract states $\mathcal{I}$, together with $\sqcup_i$ and $\sqcap_i$ forms a lattice (with a bottom $\bot_i$ and a top $\top_i = \lambda x.(-\infty, +\infty)$). The lattice is of infinite height (it is easy to see that e.g. a sequence of abstract states $\langle i_0, i_i, ..., i_n, ...\rangle$ defined as $i_n = \lambda x.(0, n)$ is strictly increasing), thus we define a widening operator $\triangledown_i$. Roughly, whenever the interval for some variable $x \in \mathcal{Var}$ in the second argument of $\triangledown_i$ extends the corresponding interval in the first argument, then the left and/or right endpoint is set to $\pm\infty$:

$$i_1 \triangledown_i i_2 \triangleq \begin{cases} \tilde{i} & \text{if } i_1 \neq \bot_i,\ i_2 \neq \bot_i, \\ i_1 & \text{if } i_2 = \bot_i, \\ i_2 & \text{otherwise} \end{cases}$$

```
$i_1 = [x \mapsto (-5,5)]$
$y \leftarrow x$
$i_2 = [x \mapsto (-5,5), y \mapsto (-5,5)]$
if $x \geqslant 0$ then
    $i_3 = [x \mapsto (0,5), y \mapsto (-5,5)]$
    ...
end if
```

Figure 2.2: Weakness of the domain of intervals

where

$$\tilde{i}^{\downarrow}(x) = \begin{cases} i_1^{\downarrow}(x) & \text{if } i_1^{\downarrow}(x) \leqslant i_2^{\downarrow}(x), \\ -\infty & \text{otherwise} \end{cases} \text{and } \tilde{i}^{\uparrow}(x) = \begin{cases} i_1^{\uparrow}(x) & \text{if } i_1^{\uparrow}(x) \geqslant i_2^{\uparrow}(x), \\ +\infty & \text{otherwise .} \end{cases}$$

The transfer function δ_i is quite straightforward. We present it for the case of a binary plus $v \leftarrow y + z$:

$$\delta_i(v \leftarrow y + z, i) \triangleq i[v \mapsto (i^{\downarrow}(y) + i^{\downarrow}(z), i^{\uparrow}(y) + i^{\uparrow}(z))] \ .$$

Finally, we demonstrate the abstract predicate semantics π_i of the inequality test $x \leqslant y$: $\pi_i(x \leqslant y, i) = (i_{\mathsf{True}}, i_{\mathsf{False}})$ where i_{True} is given by

$$i_{\mathsf{True}} \triangleq i\big[x \mapsto \big(i^{\downarrow}(x), \min(i^{\uparrow}(x), i^{\uparrow}(y))\big), y \mapsto \big(\max(i^{\downarrow}(x), i^{\downarrow}(y)), i^{\uparrow}(y)\big)\big] \ .$$

Dually, i_{False} is defined as

$$i_{\mathsf{False}} \triangleq i\big[x \mapsto \big(\max(i^{\downarrow}(x), i^{\downarrow}(y)), i^{\uparrow}(x)\big), y \mapsto \big(i^{\downarrow}(y), \min(i^{\uparrow}(x), i^{\uparrow}(y))\big)\big] \ .$$

If i_{True} (or i_{False}) defined as above is not a correct interval environment, it is replaced by $\bot_i$.

The main advantage of this domain is its simplicity and efficiency. The meet, join and widening operations can be performed in a linear (with respect to the number of variables) time and memory cost. It can be also used for real numbers, rationals or integers. On the other hand, it is not very precise—it cannot express any relationships among variables. In the code snippet presented in Figure 2.2, it will not detect that the variable y is non-negative inside the True branch of the **if** statement. To address this problem one needs a more powerful, relational domain.

2.2 OCTAGONS

The most popular relational abstract domain is the *domain of octagons* [3, 54]. It can represent sets of inequalities between pairs of variables x and

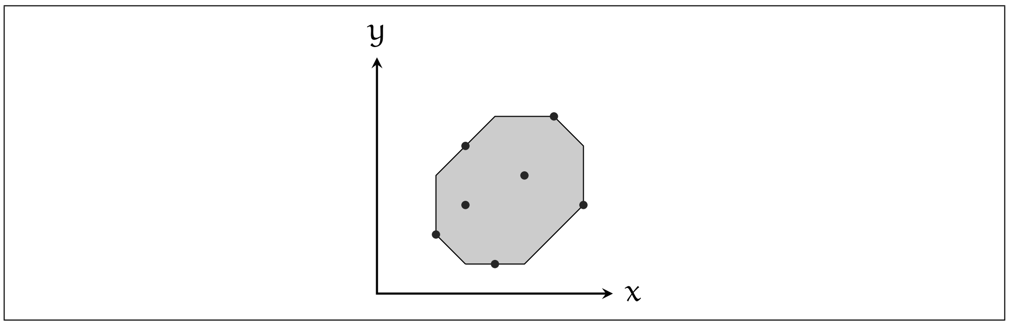

Figure 2.3: An octagon in the two-dimensional case

y of the form $\pm x \pm y \leqslant c$ (where $c \in \mathbb{V}$ is a constant). This allows one to encode also numerical intervals, e.g. $x \in [a, b]$ can be represented as $\{-x - x \leqslant -2a, x + x \leqslant 2b\}$. Any inequality of the form $\pm x \pm y \leqslant c$ is called an *octagonal constraint*. In the two-dimensional case, octagonal constraints describe a polygon with at most eight edges (Figure 2.3).

PRELIMINARIES The design of the domain of octagons follows the ideas of *Difference Bound Matrices* (DBMs) [53]. A DBM is a mapping $m\colon \mathcal{V}ar \times \mathcal{V}ar \to \mathbb{V} \cup \{+\infty\}$ that encodes a system of inequalities $\mathcal{I}$ of the form $x - y \leqslant c$ (called *potential constraints*) so that for all variables $x, y \in \mathcal{V}ar$, $m(x, y) = c \in \mathbb{V}$ (consequently, when $m(x, y) = +\infty$, then no constraint of the form $x - y \leqslant c$ is in $\mathcal{I}$).

We say that a valuation $\rho\colon \mathcal{V}ar \to \mathbb{V}$ *satisfies* a DBM, if for all $x, y \in \mathcal{V}ar$, $\rho(x) - \rho(y) \leqslant m(x, y)$.

The fundamental concept of the octagons is that each octagonal constraint $\pm x \pm y \leqslant c$ can be encoded as some potential constraint $x' - y' \leqslant c$, where x' is either x or $-x$ (and similarly for y). For example $x + y \leqslant c$ is equivalent to $x - (-y) \leqslant c$. To encode the octagonal constraints as potential constraints, the set of variables $\mathcal{V}ar$ is replaced by $\mathcal{V}ar^{\pm}$, such that for each $x \in \mathcal{V}ar$ two variables x^{+} and x^{-} are put into $\mathcal{V}ar^{\pm}$. Intuitively, x^{+} and x^{-} represent x and $-x$, respectively.

We call a DBM over $\mathcal{V}ar^{\pm}$ an *octagon*. We say that a valuation $\rho\colon \mathcal{V}ar \to \mathbb{V}$ satisfies the octagon m, if there exists an extended valuation $\rho^{\pm}\colon \mathcal{V}ar^{\pm} \to \mathbb{V}$ that fulfils the following conditions:

1. for $x \in \mathcal{V}ar$, $\rho(x) = \rho^{\pm}(x^{+})$ and $\rho(x) = -\rho^{\pm}(x^{-})$,
2. for $u, v \in \mathcal{V}ar^{\pm}$, $\rho^{\pm}(u) - \rho^{\pm}(v) \leqslant m(u, v)$.

An octagon is *satisfiable* if there exists at least one valuation that satisfies it.

THE DOMAIN OF OCTAGONS We may define now the domain of octagons $O = \langle \mathcal{O}, \sqcup_o, \sqcap_o, \top_o, \bot_o, \gamma_o, \alpha_o, \delta_o, \pi_o, \triangledown_o \rangle$. The set of abstract states $\mathcal{O}$ consists of all satisfiable octagons and one special element $\bot_o$. The concretisation function γ_o is given by

$$\gamma_o(a) \triangleq \begin{cases} \emptyset & \text{if } a = \bot_o, \\ \{\rho \mid \rho \text{ satisfies } a\} & \text{otherwise.} \end{cases}$$

The least upper bound $\sqcup_o$ is defined as a point-wise maximum:

- $a \sqcup_o \bot_o = \bot_o \sqcup_o a \triangleq a$,
- $[a \sqcup_o b](x, y) \triangleq \max(a(x, y), b(x, y))$ for all $x, y \in \mathcal{V}ar^{\pm}$.

The join is well defined, i.e. for all $a, b \in \mathcal{O}$, $a \sqcup_o b \in \mathcal{O}$. The meet $\sqcap_o$ is defined as point-wise minimum:

- $a \sqcap_o \bot_o \triangleq \bot_o \sqcap_o a \triangleq \bot_o$,
- Otherwise, let c be defined as $c(x, y) \triangleq \min(a(x, y), b(x, y))$ for all $x, y \in \mathcal{V}ar^{\pm}$. If c is satisfiable, then $a \sqcap_o b \triangleq c$. Otherwise $a \sqcap_o b = \bot_o$.

The set $\mathcal{O}$ together with meet $\sqcap_o$ and join $\sqcup_o$ forms a lattice. The least element in this lattice is $\bot_o$, while the greatest one is $\top_o = \lambda x, y. +\infty$.

Given two octagons $a, b \in \mathcal{O}$, their meet is always exact, i.e. the context abstracted by $a \sqcap_o b$ is equal to the intersection of contexts abstracted by a and b:

$$\gamma_o(a \sqcap_o b) = \gamma_o(a) \cap \gamma_o(b) \ .$$

The join is not exact, i.e. $a \sqcup_o b$ may be an abstraction of a bigger context than just the union of the contexts abstracted by a and b:

$$\gamma_o(a \sqcup_o b) \supseteq \gamma_o(a) \cup \gamma_o(b) \ .$$

A system of inequalities may entail some constraints that are not explicitly given. For example from two inequalities $x - y \leqslant c$ and $y - z \leqslant d$ one can derive $x - z \leqslant c + d$. It may happen that a derived constraint is tighter than one explicitly given in the system. The octagon abstract domain is equipped with a graph-based closure algorithm that finds the tightest possible constraints between all pairs of variables. It follows the idea of the well-known Floyd-Warshall shortest paths algorithm [9] (it treats the octagon as an adjacency matrix of a weighted and directed graph, where the nodes correspond to variables and edges represent inequalities). It can detect whether the corresponding octagon is satisfiable or not. If it is satisfiable, it can be shown that the computed closure $a^\bullet \in \mathcal{O}$ of an octagon $a \in \mathcal{O}$ can be used as a normal form for all abstract states with the same concrete meaning:

$$a^\bullet = \bigsqcap_\circ \left\{ \mathcal{b} \in \mathcal{O} \mid \gamma_\circ(\mathcal{b}) = \gamma_\circ(a) \right\} .$$

The closure can be used while computing the join $a \sqcup_\circ \mathcal{b}$ to achieve the best over-approximation of the union of the contexts $\gamma_\circ(a)$ and $\gamma_\circ(\mathcal{b})$:

$$a^\bullet \sqcup_\circ \mathcal{b}^\bullet = \bigsqcap_\circ \left\{ c \in \mathcal{O} \mid \gamma_\circ(c) \supseteq \gamma_\circ(a) \cup \gamma_\circ(\mathcal{b}) \right\} .$$

Finally, the widening operator $\triangledown_\circ$ just drops all constraints that are weaker in the second argument:

- if $a = \bot_\circ$, then $a \triangledown_\circ \mathcal{b} \triangleq \mathcal{b}$,
- else if $\mathcal{b} = \bot_\circ$, then $a \triangledown_\circ \mathcal{b} \triangleq a$,
- else $[a \triangledown_\circ \mathcal{b}](x, y) \triangleq \begin{cases} a(x,y) & \text{if } \mathcal{b}(x,y) \leqslant a(x,y), \\ +\infty & \text{otherwise.} \end{cases}$

The transfer function $\delta_\circ$ was originally defined using a conversion to the domain of polyhedra [18]. We develop here some sample definitions directly in the domain of octagons. These definitions do not tend to be as precise as possible, they are rather meant to be simple.

We start with the simplest example of an assignment $v \leftarrow c$, where $v \in \mathcal{V}ar$ and $c \in \mathbb{V}$:

$$\delta_\circ(v \leftarrow c, a)(x, y) \triangleq \begin{cases} 2c & \text{if } x = v^+ \text{ and } y = v^-, \\ -2c & \text{if } x = v^- \text{ and } y = v^+, \\ 0 & \text{if } x = y = v^+ \text{ or } x = y = v^-, \\ a(x,y) & \text{if } x, y \notin \{v^+, v^-\}, \\ +\infty & \text{otherwise.} \end{cases}$$

The abstract transfer rule for an assignment $u \leftarrow v + w$, where $u, v, w \in \mathcal{V}ar$ is much more complicated, i.e. $\delta_\circ(u \leftarrow v + w, a)(x, y)$ is given by

$$\begin{cases} a(w^+, w^-)/2 & (x,y) \in \{(u^+, v^+), (v^-, u^-)\}, \\ a(w^-, w^+)/2 & (x,y) \in \{(u^-, v^-), (v^+, u^+)\}, \\ a(v^+, v^-)/2 & (x,y) \in \{(u^+, w^+), (w^-, u^-)\}, \\ a(v^-, v^+)/2 & (x,y) \in \{(u^-, w^-), (w^+, u^+)\}, \\ a(v^+, v^-) + a(w^+, w^-) & (x,y) = (u^+, u^-), \\ a(v^-, v^+) + a(w^-, w^+) & (x,y) = (u^-, u^+), \\ a(x,y) & x, y \notin \{u^+, u^-\}, \\ +\infty & \text{elsewhere} . \end{cases}$$

Let us explain the above definition in the case when $(x, y) = (u^+, u^-)$. Note that $u^+ - u^- = 2 \cdot u = 2 \cdot (v + w) = (v^+ - v^-) + (w^+ - w^-) \leqslant a(v^+, v^-) + a(w^+, w^-)$. On the other hand, this case could be alternatively defined as $2 \cdot a(v^+, w^-)$, because $u^+ - u^- = 2 \cdot u = 2 \cdot (v + w) = 2 \cdot (v^+ - w^-) \leqslant 2 \cdot a(v^+, w^-)$. None of these definitions is in general better, thus to obtain a more precise one, the minimum $\min\big(2 \cdot a(v^+, w^-), a(v^+, v^-) + a(w^+, w^-)\big)$ could be taken. It often happens that the transfer function may be defined in many different ways and the chosen definition is always a trade-off between precision and simplicity.

We illustrate the abstract semantics of boolean predicates only on a test $v \leqslant c$, where $v \in \mathcal{V}ar$ and $c \in \mathbb{V}$, i.e. we define $\pi_\circ(v \leqslant c, a) = (a_t, a_f)$, where

$$\begin{aligned} a_t &\triangleq a[(v^+, v^-) \mapsto \min(2 \cdot c, a(v^+, v^-)], \\ a_f &\triangleq a[(v^-, v^+) \mapsto \min(-2 \cdot c, a(v^-, v^+)] . \end{aligned}$$

When a_t (resp. a_f) is not satisfiable (which can be checked using a graph-based satisfiability checking algorithm), it is replaced by $\bot_\circ$.

The computational complexity of all operations in the domain of octagons is greater than in the case of intervals. The dominating operation is the closure, which can be implemented using Floyd-Warshall algorithm [9] and works in $O(N^3)$ time, with respect to the number of variables. As the number of program variables is usually small, there is usually no need for using more complicated algorithms with slightly better worst-time complexity (such as e.g. algorithms based on fast matrix multiplication [64]). The memory cost is $O(N^2)$. The

```
$x \in [-5,5]$
$y \leftarrow x$
$x \in [-5,5] \land x - y \leqslant 0 \land y - x \leqslant 0 \land y \in [-5,5]$
if $x \geqslant 0$ then
    $x \in [0,5] \land x - y \leqslant 0 \land y - x \leqslant 0 \land y \in [0,5]$
    ...
end if
```

Figure 2.4: Example from Figure 2.2 analysed using the domain of octagons

domain is precise enough to correctly handle the example presenting the weakness of the interval domain, as shown in Figure 2.4. For clarity we show only the important constraints on x and y. The fact that $x = y$ is represented as two octagonal constraints $x - y \leqslant 0$ and $y - x \leqslant 0$. This equality, combined with $x \in [0,5]$ guarantees that $y \in [0,5]$ in the **then** branch of the **if** statement.

The domain of octagons may be used when the set $\mathbb{V}$ is chosen as real or rational numbers. It needs major changes to be applicable when $\mathbb{V}$ is chosen as integers — for instance given a constraint $x^+ - x^- \leqslant 2 \cdot c + 1$, the closure algorithm should deduce $x^+ - x^- \leqslant 2 \cdot c$. Such problem may arise e.g. when computing the closure of an octagon that represents the following system:

$$\begin{cases} y - x = 0, \\ y + x = 1 \, . \end{cases}$$

2.3 PENTAGONS

Another important relational domain is the *domain of pentagons* [49]. It can maintain constraints that combine intervals $x \in [a,b]$ and basic symbolic relationships of the form $x < y$ (where $x, y \in \mathcal{Var}$ and $a, b \in \mathbb{V}$). If only two variables are considered, the possible solutions of such inequalities form a pentagonal shape, as presented in Figure 2.5. The domain of pentagons was primarily designed to be used in an abstract interpreter for the bytecode language for Microsoft .NET platform (MSIL) as a lightweight technique to verify correctness of array accesses.

The domain of pentagons combines two basic analyses: the domain of intervals with the most basic symbolic analysis in the domain of *strict upper bounds* $S = \langle \mathcal{S}, \sqcup_s, \sqcap_s, \top_s, \bot_s, \gamma_s, \alpha_s, \delta_s, \pi_s \rangle$. The strict upper bounds encode relationships between pairs of variables $x, y \in \mathcal{Var}$ of the form $x < y$. An abstract element in this domain is a mapping

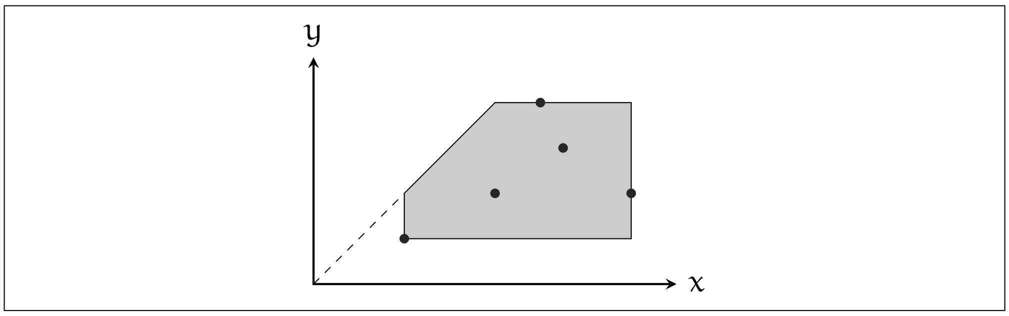

Figure 2.5: A pentagon in the two-dimensional case.

$s\colon \mathcal{V}ar \to \mathcal{P}(\mathcal{V}ar)$, where $s(x) = \{y_1, \ldots, y_k\}$ means $\{x < y_1, \ldots, x < y_k\}$. The join $\sqcup_s$ of two abstract states $s_1, s_2 \in \mathcal{S}$ selects bounds that are present both in s_1 and s_2, i.e.

$$s_1 \sqcup_s s_2 \triangleq \lambda x. s_1(x) \cap s_2(x).$$

Dually the meet $\sqcap_s$ of $s_1, s_2 \in \mathcal{S}$ is defined as the union of the sets of bounds from the two abstract states s_1 and s_2:

$$s_1 \sqcap_s s_2 \triangleq \lambda x. s_1(x) \cup s_2(x)\ .$$

The domain is finite (since the set of variables $\mathcal{V}ar$ is finite), hence no widening is necessary. The concretisation γ_s of an abstract state $s \in \mathcal{S}$ is a context that consists of all variable valuations that respect the constraints from s:

$$\gamma_s(s) \triangleq \{\rho\colon \mathcal{V}ar \to \mathbb{V} \mid \forall_{x \in \mathcal{V}ar} \forall_{y \in s(x)}\ \rho(x) < \rho(y)\}\ .$$

The abstraction α_s is defined in a natural way:

$$\alpha_s(c) = \lambda x. \{y \mid \forall_{\rho \in c}\ \rho(x) < \rho(y)\}\ .$$

The set $\mathcal{S}$ together with the meet and join operators forms a lattice, with the top element $\top_s = \lambda x. \emptyset$ and the bottom $\bot_s = \lambda x. \mathcal{V}ar$. The transfer function δ_s for assignment $x \leftarrow y$ is given by

$$\delta_s(x \leftarrow y, s) \triangleq \lambda v. \begin{cases} s(y) & \text{if } v = x, \\ s(v) \cup \{x\} & \text{if } v \neq x \text{ and } y \in s(v), \\ s(v) \setminus \{x\} & \text{otherwise.} \end{cases}$$

In more complicated cases, e.g. $x \leftarrow y + z$ the information about upper bounds for x is lost.

The domain of pentagons $P = \langle \mathcal{P}, \sqcup_p, \sqcap_p, \top_p, \bot_p, \gamma_p, \alpha_p, \delta_p, \pi_p, \triangledown_p \rangle$ combines the interval analysis I and the strict upper bounds S (however, it is not just a simple product of these two analyses). The set of abstract states $\mathcal{P} = \mathcal{I} \times \mathcal{S}$ consists of pairs of interval environments and strict upper bounds. The concretisation function γ_p of $p = (i, s) \in \mathcal{P}$ is given by $\gamma_p\big((i,s)\big) = \gamma_i(i) \cap \gamma_s(s)$, while the abstraction is performed separately for each component, i.e. $\alpha_p(c) = (\alpha_i(c), \alpha_s(c))$.

The meet $\sqcap_p$ is given point-wise:

$$(i_1, s_1) \sqcap_p (i_2, s_2) \triangleq (i_1 \sqcap_i i_2, s_1 \sqcap_s s_2)\ .$$

The join for the relational part is more complicated. It preserves the constraints that are either present in both operands or are present in one operand and are a consequence of the interval part of the other operand[1]:

$$(i_1, s_1) \sqcup_p (i_2, s_2) \triangleq (i_1 \sqcup_i i_2, \tilde{s}),$$

where $\tilde{s}$ is defined as follows:

$$\begin{aligned}\tilde{s}(x) \triangleq &\big(s_1(x) \cap s_2(x)\big)\\ &\cup \{y \in s_1(x) \mid i_2^{\uparrow}(x) < i_2^{\downarrow}(y)\}\\ &\cup \{y \in s_2(x) \mid i_1^{\uparrow}(y) < i_1^{\downarrow}(y)\}\ .\end{aligned}$$

Thanks to this modification, the strict upper bounds may benefit from the information gathered by the intervals.

In the domain of pentagons there are multiple abstract elements that represent the empty context. To avoid this issue, one would need to define and compute the closure, which would however void the performance advantage of the pentagons over the octagons. Experiments have shown that the loss of precision when no closure is used is negligible in practice [49], hence it is not desired to use this costly operation.

It is easy to check that the bottom element is $\bot_p = (\bot_i, \bot_s)$, while the top is $\top_p = (\top_i, \top_s)$. The widening $\triangledown_p$ is performed point-wise:

$$(i_1, s_1) \triangledown_p (i_2, s_2) \triangleq (i_1 \triangledown_i i_2, s_1 \sqcup_s s_2)\ .$$

We demonstrate the transfer function δ_p on an assignment $stmt = x \leftarrow y - z$:

$$\delta_p\big(stmt, (i,s)\big) \triangleq \lambda v. \begin{cases} \big(\delta_i(stmt, i)(v), \delta_s(stmt, s)(v)\big) & \text{if } v \neq x, \\ \big(\delta_i(stmt, i)(x), \widetilde{s}\big) & \text{if } v = x, \end{cases}$$

1 It is possible to give a more precise definition, but we present here the original one [49]

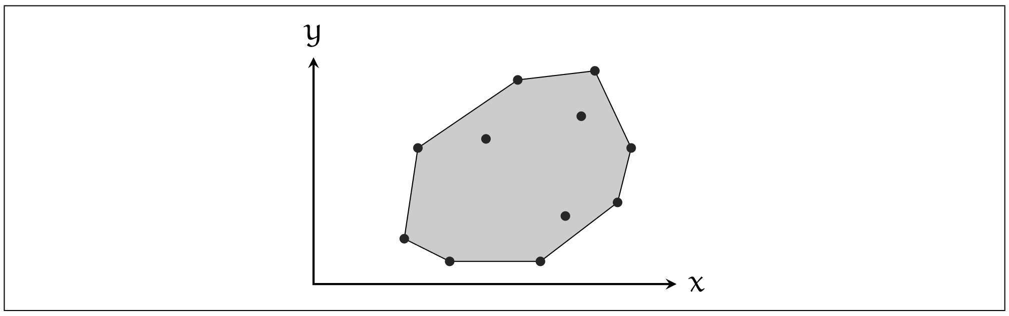

Figure 2.6: An element of the TVPI domain

where $\widetilde{s}$ defines the new upper bounds for x, depending on the numerical interval for the subtracted variable z:

$$\widetilde{s} \triangleq \begin{cases} (\{y\} \cup s(y)) \setminus \{x\} & \text{if } i^{\downarrow}(z) > 0, \\ s(y) \setminus \{x\} & \text{if } i^{\downarrow}(z) = 0, \\ \emptyset & \text{otherwise.} \end{cases}$$

If we subtract a strictly positive value from y, then the result is smaller than y and any of variables greater than y.

The domain of pentagons lies between the domains of intervals and octagons, both in terms of expressiveness and computational complexity. The time and memory cost of domain operations is $O(|\mathcal{V}ar|^2)$. It can be applied when $\mathbb{V}$ is chosen as reals, rationals or integers. The domain behaves very well in practical applications and is used in a commercial static analyser developed by Microsoft Research [49].

2.4 TWO VARIABLES PER INEQUALITY

The domain of octagons puts a serious limitation on the coefficients in the constraints, i.e. only unitary values $+1$ and -1 are allowed. The *domain of two variables per inequality* (TVPI) [61] addresses this problem. It can represent constraints of the form $ax + by \leqslant c$, where $x, y \in Var$ and $a, b, c \in \mathbb{V}$. In a two dimensional case, TVPI describes the convex hull of a given set of concrete points (Fig. 2.6).

Let us recall some basic definitions from the topology. A *metric space* is a pair (M, d), where M is a (non-empty) set and d is a function $d : M \times M \to \mathbb{R}$ that for every $x, y, z \in M$ fulfils the following conditions:

- $d(x, y) \geqslant 0$,
- $d(x, y) = 0$ if and only if $x = y$,

- $d(x, y) = d(y, x)$,
- $d(x, z) \leqslant d(x, y) + d(y, z)$.

A *ball* of radius $r > 0$ centred at point $p \in M$ is defined as $B_r(p) \triangleq \{x \in M \mid d(x, p) < r\}$. A subset P of M is called *open*, if for every $x \in P$, there exists $\epsilon > 0$ such that $B_\epsilon(x) \subseteq P$. A set $P \subseteq M$ is *closed*, if $M \setminus P$ is open. Any intersection of closed sets is closed too. A *closure* of a set $P \subseteq M$ is defined as

$$\mathrm{cl}(P) \triangleq \bigcap \{P' \subseteq M \mid P \subseteq P' \text{ and } P' \text{ is closed}\} .$$

In this section the vector space $\mathbb{R}^n$ with the Euclidean metrics $d(x, y) \triangleq \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$ is considered. We say that a set $P \in \mathbb{R}^n$ is *convex*, if for all $x, y \in P$ and each $k \in [0, 1]$ the superposition $kx + (1 - k)y$ is also in P. Any intersection of convex sets is convex. For a given set $P \in \mathbb{R}^n$ we define its *convex hull* as $\mathrm{conv}(P) \triangleq \bigcap \{A \in \mathbb{R}^n \mid P \subseteq A \text{ and } A \text{ is convex}\}$. It can be also alternatively characterised as

$$\mathrm{conv}(P) = \{k \cdot x + (1 - k) \cdot y \mid x, y \in P \wedge 0 \leqslant k \leqslant 1\} .$$

We can now proceed with the definition of the TVPI domain. Let S denote a set of systems of linear inequalities of the form $ax + by \leqslant c$, where $x, y \in \mathcal{Var}$ and $a, b, c \in \mathbb{R}$. Let $\gamma_s \colon S \to \mathcal{P}(\mathbb{R}^{\mathcal{Var}})$ be a mapping that assigns to a system all satisfying valuations, i.e.

$$\gamma_s(s) \triangleq \{\rho \colon \mathbb{R}^{\mathcal{Var}} \mid \forall_{\{ax+by\leqslant c\}\in s}\ a \cdot \rho(x) + b \cdot \rho(y) \leqslant c\} .$$

This mapping induces an ordering over S: $s_1 \sqsubseteq_s s_2$ if and only if $\gamma_s(s_1) \subseteq \gamma_s(s_2)$. It yields also a natural equivalence $s_1 \equiv s_2$ iff $\gamma_s(s_1) = \gamma_s(s_2)$. We may formalise now the TVPI domain as:

$$T = \langle \mathcal{T}, \sqcup_t, \sqcap_t, \top_t, \bot_t, \gamma_t, \alpha_t, \delta_t, \pi_t, \triangledown_t \rangle .$$

The set of abstract elements $\mathcal{T}$ is chosen as the quotient $S/\equiv$. The concretisation function γ_t is given by $\gamma_t([s]_\equiv) \triangleq \gamma_s(s)$. The meet $\sqcap_t$ of $[s_1]_\equiv$ and $[s_2]_\equiv$ is quite straightforward. It is defined as a system that contains all inequalities from s_1 and s_2:

$$[s_1]_\equiv \sqcap_t [s_2]_\equiv \triangleq [s_1 \cup s_2]_\equiv .$$

The join $\sqcup_t$ of $[s_1]_\equiv$ and $[s_2]_\equiv$ computes an equivalence class of a system $s \in S$, such that $\gamma_s(s_1) \cup \gamma_s(s_2) \subseteq \gamma_s(s)$:

$$[s_1]_\equiv \sqcup_t [s_2]_\equiv \triangleq [s]_\equiv \quad \text{where} \quad \gamma_s(s) = \mathrm{cl}\big(\mathrm{conv}(\gamma_s(s_1) \cup \gamma_s(s_2))\big) .$$

The set $\gamma_s(s_1) \cup \gamma_s(s_2)$ does not need to be convex, hence there may be no $s \in S$, such that $\gamma_s(s) = \gamma_s(s_1) \cup \gamma_s(s_2)$. In this case we compute its convex hull $conv(\gamma_s(s_1) \cup \gamma_s(s_2))$. However, such a convex hull may be an open set [61], which cannot be described using only non-strict inequalities. To solve this problem, the closure $cl(conv(\gamma_s(s_1) \cup \gamma_s(s_2)))$ is taken.

The definition of the join seems not very practical, as it relies on the concretisation of an abstract element s. We discuss now how to compute the join (find the system s) performing only symbolic operations on s_1 and s_2. In the symbolic computations, we assume that the inequalities are put in some normal form, e.g. by ordering the variables and transforming the constraints so that the first non-zero constant in each constraint is unitary.

For systems of linear inequalities with arbitrary number of dimensions, only exponential (with respect to the size of the system) convex hull algorithms (e.g. Chernikova's algorithm [8]) are known. In a planar case, one can use the $O(m \log m)$ Graham [34] algorithm (where m denotes the number of constraints). The key idea in the TVPI domain is to compute the convex hull for each planar surface separately, instead of finding one hull. This approach requires an auxiliary *completion* operation [55] for systems of inequalities $s \in S$.

We say that an inequality $i = ax_i + by_i \leqslant c$ is *entailed* by a system $s \in S$, if $\gamma_s(s \cup \{i\}) = \gamma_s(s)$. A system of inequalities $s \in S$ is *complete* if each inequality i of the form $ax_i + by_i \leqslant c$ that is entailed by s is also entailed by these inequalities from s that use only the variables x_i and y_i. The completion function transforms s into $s' \in S$ that is complete and equivalent to s (i.e. $s \equiv s'$). Technically, it performs in a loop two steps:

- remove redundant constraints: if there are $i_1, i_2, i_3 \in s$ such that $\{i_1, i_2\} \sqsubseteq_s \{i_3\}$, then i_3 can be discarded (this step is linear in the number of constraints),
- for each pair of inequalities that have a common variable, generate an inequality that eliminates this variable, e.g. for two constraints $x + 2y \leqslant 3$ and $-x + \frac{3}{2}z \leqslant \frac{1}{2}$, generate $y + \frac{3}{4}z \leqslant \frac{7}{4}$.

If the given system $s \in S$ contains n variables, then it is sufficient to iterate the loop $\log_2(n)$ times to achieve a complete system s'.

We can describe now how to compute the join $\sqcup_t$ avoiding the expensive computation of an n-dimensional convex hull. Let $p_{x,y} \colon S \to S$ be a function such that $p_{x,y}(s)$ contains these inequalities from s that use only the variables x and y. We introduce a binary operator

$\curlyvee\colon S \times S \to S$ that computes a convex hull of $\gamma_s(s_1) \cup \gamma_s(s_2)$ for each surface separately:

$$s_1 \curlyvee s_2 \triangleq \bigcup \{s_{x,y} \mid x, y \in \mathcal{Var}\}$$

where $\gamma_s(s_{x,y}) = cl(conv(\gamma_s(p_{x,y}(s_1)) \cup \gamma_s(p_{x,y}(s_2))))$. If both s_1 and s_2 are complete, then

$$\gamma_s(s_1 \curlyvee s_2) = cl(conv(\gamma_s(s_1) \cup \gamma_s(s_2))) \ .$$

Hence, to efficiently compute the join, one should complete both arguments and then find convex hull for each two-dimensional surface separately.

The crucial part of the algorithm is the completion operation that works in time $O(k^2 n^3 \log n(\log k + \log n))$, where n denotes the number of variables and k is equal to the maximal number of inequalities between any pair of variables. In general k can be arbitrarily large, but experimental results show that one can remove (without significant loss of precision) inequalities that contribute least to the shape, that is, remove the inequality that represents the shortest edge of the polyhedron. Using this technique k can be bounded by a constant and the running time of the completion operation is $O(n^3 (\log n)^2)$.

There is no best abstraction function α_t for the TVPI domain. For example, for the set $\{\langle x, y\rangle \mid x^2 + y^2 \leqslant 1\}$ there is no best abstraction, as one can always construct a regular polygon that contains more edges and better approximates the given set.

It is easy to check that the least $\bot_t$ element in $\mathcal{T}$ is equal to $[s_\emptyset]_\equiv$, where $\gamma_s(s_\emptyset) = \emptyset$, while the top is $\top_t = [s_\top]_\equiv$, where $\gamma_s(s_\top) = \mathbb{R}^{\mathcal{Var}}$.

We show only one example of the abstract transfer function δ_t, namely the transfer rule for an assignment $x \leftarrow y$. One has to discard all inequalities that involve x and add new ones $ax + bz \leqslant c$ for each inequality $ay + bz \leqslant c$:

$$\delta_t(x \leftarrow y, [s]_\equiv) \triangleq [(s \setminus s_x) \cup s_{x \leftarrow y}]_\equiv$$

where s_x consists of all inequalities from s that involve x and $s_{x \leftarrow y} = \{ax + bz \leqslant c \mid \{ay + bz \leqslant c\} \in s \setminus s_x\}$.

The widening ∇_t can be adapted from the convex polyhedra [18]. In this approach in a widening step $[s_1]_\equiv \nabla_t [s_2]_\equiv$ only these inequalities from s_1 are kept that are satisfied by all points from $\gamma_s(s_2)$:

$$[s_1]_\equiv \nabla_t [s_2]_\equiv \triangleq [\{i \mid i \in s_1 \text{ and } \gamma_s(s_2 \cup \{i\}) = \gamma_s(s_2)\}]_\equiv \ .$$

The domain of TVPI can be applied when $\mathbb{V}$ is chosen as $\mathbb{R}$ or $\mathbb{Q}$. In the integer case the problem of satisfiability of a system of constraints of the form $a \cdot x + b \cdot y \leqslant c$, where $x, y \in \mathcal{V}ar$ and $a, b, c \in \mathbb{V}$ is NP-complete [47].

2.5 OTHER DOMAINS

We have described only a few of many existing numerical domains. We mention now some other interesting approaches. Using the *constant propagation* [43] it is possible to identify variables, the value of which must be equal at some program point to the same constant in all possible program executions. The domain of *convex polyhedra* [18] can express arbitrary linear constraints over the variables. It is more powerful than any of the domains presented in this chapter, but the complexity of the domain operations is exponential. The *domain of congruences* [35] can discover systems of linear congruence equations of the form $\Sigma_i a_i \cdot x_i \equiv c \mod n$, where $x_i \in \mathcal{V}ar$ and $a_i, c, n \in \mathbb{Z}$.

The domains described in this chapter can represent only convex sets. This limitation can be dropped by using *powerset domains* [30] that maintain at each program point a set of abstract elements, instead of a single element. This approach was studied mostly for the domain of intervals [37, 41].

3 WEIGHTED HEXAGONS

In this chapter we present our domain of weighted hexagons that was originally sketched in a conference article [27]. It is designed to be more precise than the domain of pentagons (see Section 2.3), but simpler and computationally easier than TVPI (Section 2.4). In this domain one can represent invariants that combine interval constraints $x \in [a, b]$ with *hexagonal constraints* $x \leqslant a \cdot y$, where $x, y \in \mathcal{V}ar$ and a is some non-negative constant. We say that a is a *coefficient* of this constraint. In a two-dimensional case such constraints describe a polygon with at most six edges (Figure 3.1). The angles in the polygon are determined by coefficients of the hexagonal constraints. This motivates the name *weighted hexagons*. In Chapter 4 we extend this domain to strict variants of the constraints such as $x < a \cdot y$ or $x \in (a, b)$.

The domain of weighted hexagons can be used instead of the domain of pentagons to slightly more precisely analyse numerical properties of programs that use multiplication. For instance a program that implements a dynamic array (an array, whose size can be changed) adjusts the capacity of the array (by increasing or decreasing it by a factor of two), depending on the current ratio of used elements. Each time when the capacity is changed, new array is allocated and the array content must be copied. The domain of weighted hexagons can be used to verify the correctness of array accesses during the copying phase.

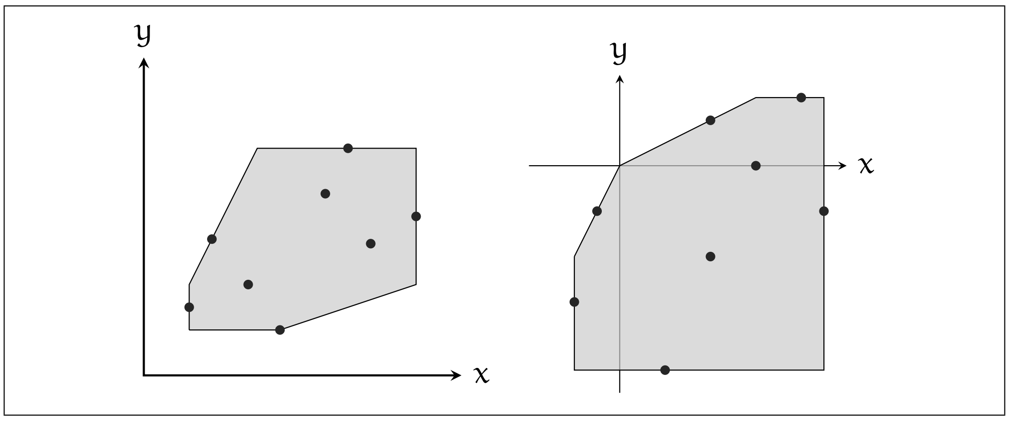

Figure 3.1: Weighted hexagons in a two-dimensional case.

```
Require array In
Out ← new array($2 \cdot \mathrm{In.len}$)
$1 \leqslant \mathrm{Out.len} \leqslant 2 \cdot \mathrm{In.len} \wedge 1 \leqslant \mathrm{In.len} \leqslant \frac{1}{2} \cdot \mathrm{Out.len}$
$i \leftarrow 1$
while $i \leqslant \mathrm{In.len}$ do
    $\ldots 1 \leqslant i \leqslant \frac{1}{2} \cdot \mathrm{Out.len} \ldots$
    $\mathrm{Out}[2 \cdot i - 1] \leftarrow \mathrm{In}[i]$
    $\mathrm{Out}[2 \cdot i] \leftarrow \mathrm{In}[i]$
    $i \leftarrow i + 1$
end while
```

Figure 3.2: Duplicating the content of the array In

In Figure Figure 3.2 we present an even simpler example that demonstrates the advantage of the domain of weighted hexagons over the domain of pentagons. The crucial invariant $i \leqslant \frac{1}{2} \cdot \mathrm{Out.len}$ (that is needed to prove that the array Out is not accessed out of its bounds within the loop) can be expressed (and will be automatically inferred) in our domain, but it can be represented neither in the domain of pentagons nor in the domain of octagons. It can be of course expressed in TVPI.

The elements of the domain of weighted hexagons represent finite systems of hexagonal constraints, thus before we formally develop the domain, we focus on some properties of such systems.

SYSTEMS OF HEXAGONAL CONSTRAINTS Let $\mathcal{I}$ be a finite set of hexagonal constraints. We say that a valuation $\rho\colon \mathcal{V}ar \to \mathbb{V}$ is a *solution* of $\mathcal{I}$, when it satisfies all constraints from $\mathcal{I}$, i.e. $\rho(x) \leqslant a \cdot \rho(y)$ for each $x \leqslant a \cdot y$ in $\mathcal{I}$. We call $\mathcal{I}$ a *system* of hexagonal constraints. Given two systems $\mathcal{I}$ and $\mathcal{J}$, the system $\mathcal{C}$ is a *conjunction* of $\mathcal{I}$ and $\mathcal{J}$, when a solution ρ is a solution of $\mathcal{C}$ if and only if it is a solution of $\mathcal{I}$ and of $\mathcal{J}$. The conjunction of $\mathcal{I}$ and $\mathcal{J}$ always exists (and is equivalent to $\mathcal{I} \cup \mathcal{J}$). Dually, one could try to define an *alternative* $\mathcal{A}$ of $\mathcal{I}$ and $\mathcal{J}$ such that ρ is a solution of $\mathcal{A}$ if and only if it is a solution either of $\mathcal{I}$ or of $\mathcal{J}$. However, the set of such solutions does not need to be convex (as shown in Figure 3.3), thus $\mathcal{A}$ need not exist (as a system of hexagonal constraints always has a convex set of solutions).

It may happen that a system $\mathcal{I}$ contains some redundant information, i.e. there exists a system $\mathcal{I}' \subsetneq \mathcal{I}$ with the same set of solutions as $\mathcal{I}$. We are particularly interested in constraints binding the same two variables, e.g. $y \leqslant 2x$ and $y \leqslant \frac{1}{2}x$ for $\mathbb{V} = \mathbb{R}$. Unless we have an extra knowledge about x and y (for instance that x and y are limited to

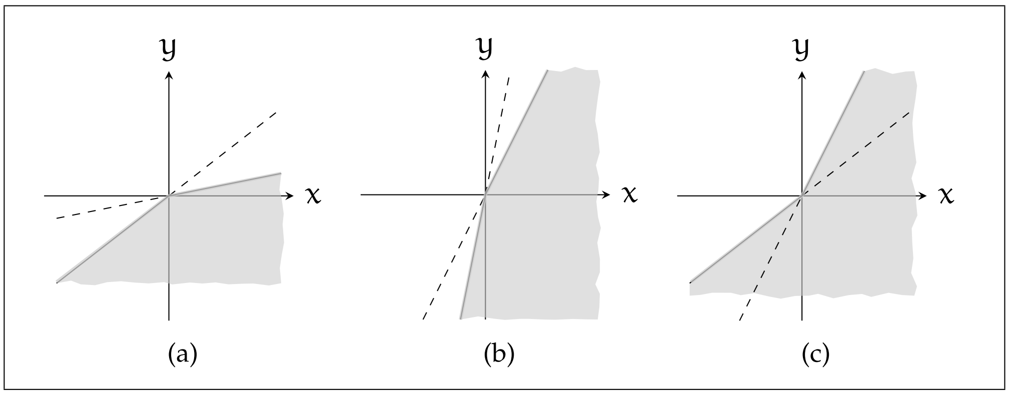

Figure 3.3: Solutions of two two-dimensional systems ((a) and (b)) and solutions of their alternative (c)

positive values) we have to keep track of both inequalities. However, if we additionally had that $y \leqslant x$ then this constraint would be superfluous as can be seen in Figure 3.4, thus it can be safely discarded. This justifies the approach of representing systems of hexagonal constraints using for each pair of variables only the two extreme constraints.

Numerical intervals can be encoded as hexagonal constraints using a simple trick. We extend the set of variables $\mathcal{Var}$ by introducing three artificial variables denoted by c^-, c^0 and c^+ that are equal (in all possible valuations) to -1, 0 and 1, respectively. Each interval $x \in [a, b]$ can now be represented as two hexagonal constraints (that involve x and one of the artificial variables), e.g. $x \in [-5, 3]$ is expressed as a conjunction of $c^- \leqslant \frac{1}{5}x$ and $x \leqslant 3c^+$. In the rest of this chapter we assume that $c^-, c^0, c^+ \in \mathcal{Var}$ and that $\mathcal{I}$ contains the following trivial inequalities relating these artificial variables:

$$\begin{aligned} &\{x \leqslant 0 \cdot y \mid x \in \{c^-, c^0\}, y \in \{c^-, c^0, c^+\}\} \\ \cup &\{x \leqslant +\infty \cdot y \mid x \in \{c^-, c^0\}, y \in \{c^0, c^+\}\} \\ \cup &\{x \leqslant 1 \cdot x \mid x \in \{c^-, c^+\}\} \cup \{c^+ \leqslant +\infty \cdot c^+\} \,. \end{aligned} \tag{3.1}$$

The (obvious) extension of the multiplication in $\mathbb{V}$ to $\mathbb{V} \cup \{+\infty\}$ will be formalised below. The set of non-negative elements of $\mathbb{V}$ is denoted by $\mathbb{V}_{\geqslant 0}$, i.e. $\mathbb{V}_{\geqslant 0} \triangleq \{a \mid a \in \mathbb{V} \wedge 0 \leqslant a\}$. A special constant Nil will be used to indicate that there is no constraint between some pair of variables.

Following the discussion above, we introduce a representation of a system $\mathcal{I}$ as a pair of functions $s, l \colon \mathcal{Var} \times \mathcal{Var} \to \mathbb{V}_{\geqslant 0} \cup \{+\infty, \mathsf{Nil}\}$ (s

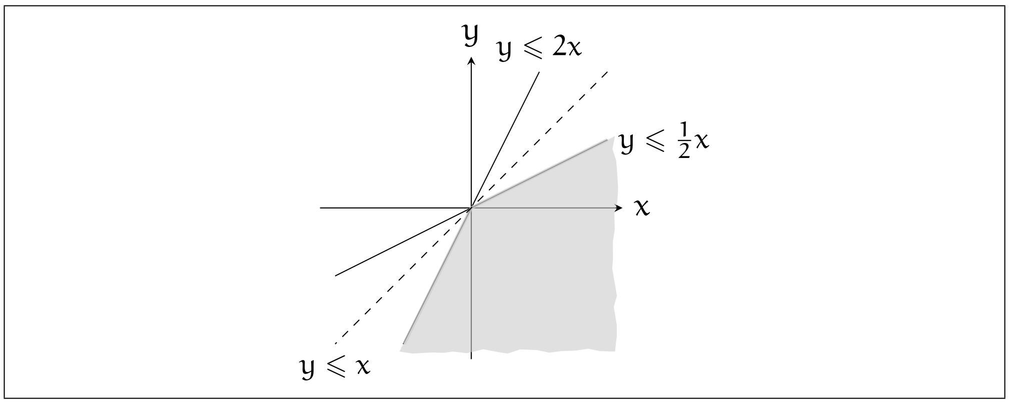

Figure 3.4: Region of possible solutions limited by inequalities $y \leqslant 2x$ and $y \leqslant \frac{1}{2}x$

stores the *smallest* and l the *largest* coefficient in constraints between a pair of variables) defined as follows:

$$\begin{aligned} s(x,y) &\triangleq \min\big\{a \in \mathbb{V}_{\geqslant 0} \cup \{+\infty\} \mid \text{inequality } x \leqslant a \cdot y \text{ is in } \mathcal{I}\big\} \\ l(x,y) &\triangleq \max\big\{a \in \mathbb{V}_{\geqslant 0} \cup \{+\infty\} \mid \text{inequality } x \leqslant a \cdot y \text{ is in } \mathcal{I}\big\} \, . \end{aligned} \tag{3.2}$$

We additionally define $s(x,y) \triangleq l(x,y) \triangleq \mathsf{Nil}$ if and only if $\mathcal{I}$ contains no constraint $x \leqslant a \cdot y$. This definition ensures that either $s(x,y) = l(x,y) = \mathsf{Nil}$ or $s(x,y) \leqslant l(x,y)$, where neither $s(x,y)$ nor $l(x,y)$ is equal to Nil.

We have allowed also $+\infty$ as a valid value of the functions s and l. It will be used in the normalisation algorithm (see Section 3.3) for paired functions (s,l). Roughly, when for some pair of variables $x, y \in \mathcal{V}ar$ the system $\mathcal{I}$ entails a constraint $x \leqslant M \cdot y$ for each M greater than some M_0, we will put $l(x,y) = +\infty$.

The pair of functions (s,l) representing some system $\mathcal{I}$ will be called a *weighted hexagon*.

We formalise now the meaning of s and l, i.e. we say when a variable valuation ρ satisfies (s,l). We start with extending the standard multiplication to $\mathbb{V} \cup \{\mathsf{Nil}, \pm\infty\}$:

- $a \cdot \mathsf{Nil} = \mathsf{Nil} \cdot a = \mathsf{Nil}$,
- $+\infty \cdot a = a \cdot +\infty = +\infty$ if $a > 0$,
- $+\infty \cdot a = a \cdot +\infty = -\infty$ if $a < 0$,
- $0 \cdot +\infty = +\infty \cdot 0 = 0$.
- $-\infty \cdot a = a \cdot -\infty = -\infty$ if $a > 0$,

- $-\infty \cdot a = a \cdot -\infty = +\infty$ if $a < 0$,
- $0 \cdot +\infty = +\infty \cdot 0 = 0$.

We introduce two order relations in $\mathbb{V}_{\geqslant 0} \cup \{\pm\infty, \mathsf{Nil}\}$:

$$a \leqslant^\star b \text{ iff } a \leqslant b \text{ or } b = \mathsf{Nil} \qquad a \leqslant_\star b \text{ iff } a \leqslant b \text{ or } a = \mathsf{Nil}\ .$$

Obviously, we can also introduce operators (that are denoted by $\min_{\leqslant^\star}$ and $\max_{\leqslant^\star}$; similarly for $\leqslant_\star$) for computing minimum and maximum of an arbitrary, yet finite, number of elements with respect to $\leqslant^\star$ and $\leqslant_\star$ orders. We write $a <^\star b$ to denote $a \leqslant^\star b$ and $a \neq b$ (and similarly for $<_\star$).

A valuation $\rho\colon \mathcal{Var} \to \mathbb{V}$ *satisfies* the weighted hexagon (s, l) if it fulfils the following conditions:

1. $\rho(c^-) = -1$, $\rho(c^0) = 0$ and $\rho(c^+) = 1$,
2. $\forall_{x,y \in \mathcal{Var}}\ \rho(x) \leqslant^\star s(x, y) \cdot \rho(y)$,
3. $\forall_{x,y \in \mathcal{Var}}\ \rho(x) \leqslant^\star l(x, y) \cdot \rho(y)$.

A valuation that satisfies (s, l) will be also called a *solution* of (s, l).

Now one can treat $s(x, y) = l(x, y) = \mathsf{Nil}$ as a representation of an artificial constraint $x \leqslant \mathsf{Nil} \cdot y$, which is satisfied by each variable valuation ρ (as $\rho(x) \leqslant^\star \mathsf{Nil} \cdot \rho(y)$ always holds). It is worth mentioning that we could not use $+\infty$ instead of Nil. For example the constraint $c^+ \leqslant \mathsf{Nil} \cdot c^-$ holds in each valuation, while $c^+ \leqslant +\infty \cdot c^-$ is not satisfiable.

Clearly, a valuation ρ satisfies (s, l) if and only if it is a solution of the system $\mathcal{I}$ represented by (s, l) as above.

We say that (s, l) is *satisfiable* if there exists at least one valuation ρ that satisfies it.

3.1 THE DOMAIN

We may formalise now the domain of weighted hexagons as a tuple $H = \langle \mathcal{H}, \sqcup_h, \sqcap_h, \top_h, \bot_h, \gamma_h, \alpha_h, \delta_h, \pi_h, \triangledown_h \rangle$. The set of abstract states $\mathcal{H}$ consists of all satisfiable weighted hexagons and a special element $\bot_h$. The concretisation function γ_h is just given by

$$\gamma_h(a) \triangleq \begin{cases} \emptyset & \text{if } a = \bot_h, \\ \{\rho \mid \rho \text{ satisfies } a\} & \text{otherwise}\ . \end{cases}$$

3.1.1 *Domain Operations*

The meet $a \sqcap_h b$ of two weighted hexagons $a, b \in \mathcal{H}$ corresponds to a conjunction of systems of constraints $\mathcal{I}_a$ and $\mathcal{I}_b$ encoded as a and b, respectively. The system $\mathcal{I}_{a \sqcap_h b}$ entails all inequalities from $\mathcal{I}_a$ and $\mathcal{I}_b$. Thus, as in our representation we keep only constraints with smallest and largest coefficients, for each pair of variables $x, y \in \mathcal{V}ar$, $s_{a \sqcap_h b}(x, y)$ is defined as the minimum of $s_a(x, y)$ and $s_b(x, y)$. Dually, $l_{a \sqcap_h b}(x, y)$ is defined as a maximum of $l_a(x, y)$ and $l_b(x, y)$:

$$\begin{aligned} & s_{a \sqcap_h b}(x, y) \triangleq \min_{\leqslant^\star}\left(s_a(x, y), s_b(x, y)\right) \qquad \text{and} \\ & l_{a \sqcap_h b}(x, y) \triangleq \max_{\leqslant_\star}\left(l_a(x, y), l_b(x, y)\right) . \end{aligned} \tag{3.3}$$

Finally, the meet $a \sqcap_h b$ is defined as $(s_{a \sqcap_h b}, l_{a \sqcap_h b})$, whenever it is satisfiable, and $\perp_h$ otherwise:

$$a \sqcap_h b \triangleq \begin{cases} (s_{a \sqcap_h b}, l_{a \sqcap_h b}) & \text{if } a = (s_a, l_a) \text{ and } b = (s_b, l_b) \\ & \quad \text{and } \gamma_h((s_{a \sqcap_h b}, l_{a \sqcap_h b})) \neq \emptyset \\ \perp_h & \text{otherwise.} \end{cases}$$

It is not evident how to efficiently check whether $\gamma_h((s_{a \sqcap_h b}, l_{a \sqcap_h b}))$ is empty. We discuss this problem in detail in Section 3.3.

JOIN The join $a \sqcup_h b$ should encode an alternative of the systems $\mathcal{I}_a$ and $\mathcal{I}_b$, but, as we have already discussed, the exact alternative may, in general, not exist.

We define the join $a \sqcup_h b$ as a representation of some system $\mathcal{I}'$ such that each solution of any of the systems $\mathcal{I}_a$ or $\mathcal{I}_b$ is also a solution of $\mathcal{I}'$ (but we do not require the converse). If $a = \perp_h$ then $a \sqcup_h b \triangleq b$, symmetrically, if $b \triangleq \perp_h$, then $a \sqcup_h b = a$. In other cases we define:

$$\begin{aligned} & \widehat{s}(x, y) \triangleq \max_{\leqslant^\star}\left(s_a(x, y), s_b(x, y)\right) \qquad \text{and} \\ & \widehat{l}(x, y) \triangleq \min_{\leqslant_\star}\left(l_a(x, y), l_b(x, y)\right) . \end{aligned}$$

The pair $(\widehat{s}, \widehat{l})$ may be not a valid weighted hexagon, as it may violate the property that for every $x, y \in \mathcal{V}ar$, $\widehat{s}(x, y) \leqslant^\star \widehat{l}(x, y)$. We fix this problem by putting $s(x, y) = l(x, y) \triangleq \mathsf{Nil}$ in this case (below we write $(s, l)(x, y)$ to denote $(s(x, y), l(x, y))$:

$$(s_{a \sqcup_h b}, l_{a \sqcup_h b})(x, y) \triangleq \begin{cases} (\widehat{s}, \widehat{l})(x, y) & \text{if } \widehat{s}(x, y) \leqslant^\star \widehat{l}(x, y), \\ (\mathsf{Nil}, \mathsf{Nil}) & \text{otherwise.} \end{cases} \tag{3.4}$$

Note that $(s_{a \sqcup_h b}, l_{a \sqcup_h b})$ is always satisfiable. Summing up:

$$a \sqcup_h b \triangleq \begin{cases} a & \text{if } b = \bot_h, \\ b & \text{if } a = \bot_h, \\ (s_{a \sqcup_h b}, l_{a \sqcup_h b}) & \text{otherwise.} \end{cases}$$

We have already mentioned that $a \sqcap_h b$ exactly represents the conjunction of systems $\mathcal{I}_a$ and $\mathcal{I}_b$, while $a \sqcup_h b$ only over-approximates their alternative. This can be formalised by the following lemma:

Lemma 3.5. *For all $a, b \in \mathcal{H}$ we have that:*

1. $\gamma_h(a \sqcap_h b) = \gamma_h(a) \cap \gamma_h(b)$,
2. $\gamma_h(a \sqcup_h b) \supseteq \gamma_h(a) \cup \gamma_h(b)$.

Proof. A standard case analysis, see Section 3.5.1. □

From this lemma it also immediately follows that γ_h is monotone.

Theorem 3.6. *Set $\mathcal{H}$ forms a lattice under $\sqcap_h$ and $\sqcup_h$ operators.*

Proof. Direct examination of the associativity, commutativity and absorption. Details can be found in Section 3.5.2. □

It is easy to see that $\bot_h$ is the least element in this lattice. The greatest element (denoted by $\top_h$) is a pair $(s_\top, l_\top)$ such that for each $x, y \in \mathcal{V}ar$, if $x \notin \{c^-, c^0, c^+\}$ or $y \notin \{c^-, c^0, c^+\}$, then $(s_\top, l_\top)(x, y) = (\mathsf{Nil}, \mathsf{Nil})$.

If $\mathbb{V}$ is chosen as $\mathbb{R}$ then the lattice $\langle \mathcal{H}, \sqcup_h, \sqcap_h \rangle$ is complete and the abstraction function α_h is uniquely defined as the lower adjoint of the Galois connection $\langle \mathcal{C}tx, \cup, \cap \rangle \underset{\alpha_h}{\overset{\gamma_h}{\leftrightarrows}} \langle \mathcal{H}, \sqcup_h, \sqcap_h \rangle$. When $\mathbb{V}$ is chosen as $\mathbb{Q}$, the lattice $\langle \mathcal{H}, \sqcup_h, \sqcap_h \rangle$ is not complete and, in general, there is no best abstraction α_h, but we can still work in the concretisation-based variant of abstract interpretation (see Section 1.2.8).

WIDENING The widening $a \nabla_h b$ preserves only these constraints from a that are not relaxed in b:

$$a \nabla_h b \triangleq \begin{cases} (s_{a \nabla_h b}, l_{a \nabla_h b}) & \text{if } a = (s_a, l_a) \text{ and } b = (s_b, l_b) \\ a & \text{if } b = \bot_h, \\ b & \text{otherwise,} \end{cases}$$

where for every $x, y \in \mathcal{V}ar$:

- if $s_{\mathfrak{b}}(x,y) \leqslant^{\star} s_{a}(x,y)$ and $l_{a}(x,y) \leqslant_{\star} l_{\mathfrak{b}}(x,y)$ then:

$$s_{a\nabla_h \mathfrak{b}}(x,y) \triangleq s_{a}(x,y) \qquad \text{and} \qquad l_{a\nabla_h \mathfrak{b}}(x,y) \triangleq l_{a}(x,y)\ .$$

- $s_{a\nabla_h \mathfrak{b}}(x,y) \triangleq l_{a\nabla_h \mathfrak{b}}(x,y) \triangleq \mathsf{Nil}$ otherwise.

Theorem 3.7. *The ∇_h operator defined above meets the definition of a widening operator presented in Section 1.2.4.*

Proof. For the proof, refer to Section 3.5.3. □

3.1.2 *Transfer Function*

We present now the transfer function δ_h for all types of instructions. To define $\delta_h(I, a)$ we use the following (self-explanatory, we hope) syntax

$$\frac{\textbf{def } F_I}{\delta_h(I,a) = F_I(a)}$$

where **def** F_I denotes a definition of an auxiliary function F_I.

The definition for a constant-to-variable assignment (Figure 3.5) consists of three cases, depending on the sign of the constant. Basically, when processing an assignment $w \leftarrow a$, all constraints that mention w are invalidated and two new constraints (binding w with one of the artificial variables c^-, c^0 or c^+) are added.

In the transfer rule for a variable-to-variable assignment $w \leftarrow u$ (Figure 3.6) we add for each variable $x \in \mathcal{V}ar$ constraints $w \leqslant a \cdot x$ and $x \leqslant b \cdot w$, whenever there was $u \leqslant a \cdot x$ and $x \leqslant b \cdot u$.

In the case of the unary minus $w \leftarrow -u$, where $u, w \in \mathcal{V}ar$, (Figure 3.7) only constraints binding w with the artificial variables c^-, c^0 and c^+ can be deduced. We do not admit negative coefficients, hence no constraint binding w and u can be inferred.

The key observation in the definition of a transfer rule for the binary plus $w \leftarrow u + v$ (Figure 3.8) is that for each variable $x \in \mathcal{V}ar$, if there are constraints $u \leqslant a \cdot x$ and $v \leqslant b \cdot x$, then a new constraint $w \leqslant (a+b) \cdot x$ can be added. Moreover, if there are inequalities $x \leqslant a \cdot u$ and $x \leqslant b \cdot v$, then (assuming $a \cdot b > 0$) $\frac{1}{a} \cdot x \leqslant u$ and $\frac{1}{b} \cdot x \leqslant v$, thus $w = u + v \geqslant (\frac{1}{a} + \frac{1}{b}) \cdot x = \frac{a+b}{ab} \cdot x$, which is equivalent to $x \leqslant \frac{ab}{a+b} \cdot w$.

The definition of the rule for the binary minus can be justified in a very similar manner. If we have two inequalities $u \leqslant a \cdot x$ and $x \leqslant b \cdot v$ (which is equivalent to $v \geqslant \frac{1}{b} \cdot x$, if $b > 0$) we have that $w \leqslant a \cdot x - \frac{1}{b} \cdot x = \frac{ab-1}{b} \cdot x$.

$$\frac{\big(F(f)\big)(x,y) \triangleq \begin{cases} f(x,y) & \text{if } w \notin \{x,y\}, \\ a & \text{if } x = w, y = c^+, \\ 1/a & \text{if } x = c^+, y = w, \\ \mathsf{Nil} & \text{otherwise} \end{cases}}{\delta_h\big(w \leftarrow a, (s,l)\big) = \big(F(s), F(l)\big)}\; a > 0$$

(a)

$$\frac{\big(F(f)\big)(x,y) \triangleq \begin{cases} f(x,y) & \text{if } w \notin \{x,y\}, \\ |a| & \text{if } x = w, y = c^-, \\ |1/a| & \text{if } x = c^-, y = w, \\ \mathsf{Nil} & \text{otherwise} \end{cases}}{\delta_h\big(w \leftarrow a, (s,l)\big) = \big(F(s), F(l)\big)}\; a < 0$$

(b)

$$\frac{\big(F(f)\big)(x,y) \triangleq \begin{cases} f(x,y) & \text{if } w \notin \{x,y\}, \\ 1 & \text{if } x = w, y = c^0, \\ 1 & \text{if } x = c^0, y = w, \\ \mathsf{Nil} & \text{otherwise} \end{cases}}{\delta_h\big(w \leftarrow a, (s,l)\big) = \big(F(s), F(l)\big)}\; a = 0$$

(c)

Figure 3.5: Transfer rule for a constant-to-variable assignment

$$\frac{\big(F(f)\big)(x,y) \triangleq \begin{cases} f(x,y) & \text{if } w \notin \{x,y\}, \\ f(u,y) & \text{if } x = w, y \neq u, \\ f(x,u) & \text{if } y = w, x \neq u, \\ 1 & \text{if } (x,y) \in \{(w,u),(u,w),(w,w)\} \end{cases}}{\delta_h\big(w \leftarrow u, (s,l)\big) = \big(F(s), F(l)\big)}$$

Figure 3.6: Transfer rule for variable-to-variable assignment

$$\frac{\big(F(f)\big)(x,y) \triangleq \begin{cases} f(x,y) & \text{if } w \notin \{x,y\}, \\ \frac{1}{l(c^-,u)} & \text{if } x = w, y = c^+, 0 <_\star l(c^-,u), \\ \frac{1}{l(u,c^-)} & \text{if } x = c^+, y = w, 0 <_\star l(u,c^-), \\ \frac{1}{s(u,c^+)} & \text{if } x = c^-, y = w, 0 <_\star s(u,c^+), \\ \frac{1}{s(c^+,u)} & \text{if } x = w, y = c^-, 0 <_\star s(c^+,u), \\ f(u,c^0) & \text{if } x = c^0, y = w, \\ f(c^0,u) & \text{if } x = w, y = c^0, \\ \mathsf{Nil} & \text{otherwise} \end{cases}}{\delta_h\big(w \leftarrow -u, (s,l)\big) = \big(F(s), F(l)\big)}$$

Figure 3.7: Transfer rule for unary minus

$$\frac{\big(F(f)\big)(x,y) \triangleq \begin{cases} f(x,y) & \text{if } w \notin \{x,y\}, \\ 1 & \text{if } x = y = w, \\ f(u,y) + f(v,y) & \text{if } x = w, \mathsf{Nil} \notin \{f(u,y), f(v,y)\}, \\ \frac{f(x,u)\cdot f(x,v)}{f(x,u)+f(x,v)} & \text{if } y = w \land \mathsf{Nil} \notin \{f(x,u), f(v,x)\} \\ & \land f(x,u)\cdot f(x,v) \neq 0, \\ \mathsf{Nil} & \text{otherwise} \end{cases}}{\delta_h\big(w \leftarrow u + v, (s,l)\big) = \big(F(s), F(l)\big)}$$

$$\frac{\big(F(f)\big)(x,y) \triangleq \begin{cases} f(x,y) & w \notin \{x,y\}, \\ 1 & x = y = w, \\ \frac{f(u,y)\cdot f(y,v)-1}{f(y,v)} & x = w \land \mathsf{Nil} \notin \{f(u,y), f(y,v)\} \\ & \land f(u,y)\cdot f(y,v) \geqslant 1, \\ \frac{f(x,u)}{1-f(x,u)\cdot f(v,x)} & y = w \land \mathsf{Nil} \notin \{f(x,u), f(v,x)\} \\ & \land f(x,u)\cdot f(v,x) < 1, \\ \mathsf{Nil} & \text{otherwise} \end{cases}}{\delta_h\big(w \leftarrow u - v, (s,l)\big) = \big(F(s), F(l)\big)}$$

Figure 3.8: Transfer rules for binary plus and minus

$$m(x,f) = \begin{cases} \frac{1}{f(c^+,x)} & f(c^+,x) \notin \{0, \mathsf{Nil}\}, \\ 0 & f(c^0,x) \neq \mathsf{Nil}, \\ \frac{-1}{f(c^-,x)} & f(c^-,x) \notin \{0, \mathsf{Nil}\}, \\ \mathsf{Nil} & \text{otherwise} \end{cases} \qquad M(x,f) = \begin{cases} -f(x,c^-) & f(x,c^-) \neq \mathsf{Nil}, \\ 0 & f(x,c^0) \neq \mathsf{Nil}, \\ f(x,c^+) & f(x,c^+) \neq \mathsf{Nil}, \\ \mathsf{Nil} & \text{otherwise} \end{cases}$$

$$m_w(f) = \min_{\leqslant_\star}\big(m(u,f)\cdot m(v,f), M(u,f)\cdot m(v,f), m(u,f)\cdot M(v,f), M(u,f)\cdot M(v,f)\big)$$

$$M_w(f) = \max_{\leqslant^\star}\big(m(u,f)\cdot m(v,f), M(u,f)\cdot m(v,f), m(u,f)\cdot M(v,f), M(u,f)\cdot M(v,f)\big)$$

$$\big(F(f)\big)(x,y) \triangleq \begin{cases} \frac{1}{m_w(f)} & x = c^+, y = w, m_w(f) \neq \mathsf{Nil}, m_w(f) > 0, \\ 0 & x = c^0, y = w, m_w(f) = 0, \\ \frac{-1}{m_w(f)} & x = c^-, y = w, m_w(f) \neq \mathsf{Nil}, m_w(f) < 0, \\ -M_w(f) & x = w, y = c^-, M_w(f) \neq \mathsf{Nil}, M_w(f) < 0, \\ 0 & x = w, y = c^0, M_w(f) = 0, \\ M_w(f) & x = w, y = c^+, M_w(f) \neq \mathsf{Nil}, M_w(f) > 0, \\ \frac{1}{m(t,f)}\cdot f(x,r) & y = w, (r,t) \in \{(u,v),(v,u)\}, m(t,f) > 0, f(x,r) > 0, \\ M(t,f)\cdot f(r,y) & x = w, (r,t) \in \{(u,v),(v,u)\}, m(t,f) > 0, \\ \mathsf{Nil} & \text{otherwise when } w \in \{x,y\}, \\ f(x,y) & \text{otherwise} \end{cases}$$

$$\overline{\qquad\qquad}$$

$$\delta_h\big(w \leftarrow u\cdot v, (s,l)\big) = \big(F(s), F(l)\big)^*$$

Figure 3.9: Transfer rule for binary multiplication; $(F(s), F(l))^*$ denotes an application of the transitive closure algorithm, see Section 3.3

$$s_{\mathsf{True}}(x,y) = \begin{cases} s(x,y) & \text{if } x \neq u, y \neq w, \\ \min_{\leqslant^\star}(s(u,w),1) & \text{otherwise} \end{cases}$$

$$l_{\mathsf{True}}(x,y) = \begin{cases} l(x,y) & \text{if } x \neq u, y \neq w, \\ \max_{\leqslant_\star}(l(u,w),1) & \text{otherwise} \end{cases}$$

$$s_{\mathsf{False}}(x,y) = \begin{cases} s(x,y) & \text{if } x \neq w, y \neq u, \\ \min_{\leqslant^\star}(s(w,u),1) & \text{otherwise} \end{cases}$$

$$l_{\mathsf{False}}(x,y) = \begin{cases} l(x,y) & \text{if } x \neq w, y \neq u, \\ \max_{\leqslant_\star}(l(w,u),1) & \text{otherwise} \end{cases}$$

$$\pi_{\mathsf{h}}\big(u \leqslant w, (s,l)\big) = \big((s_{\mathsf{True}}, l_{\mathsf{True}}), (s_{\mathsf{False}}, l_{\mathsf{False}})\big)$$

Figure 3.10: Abstract semantics of a boolean predicate $u \leqslant w$

Let us explain the transfer rule for multiplication $w \leftarrow u \cdot v$ presented in Figure 3.9. For the given weighted hexagon (s,l) and a variable $x \in \mathcal{V}ar$, $m(x,s)$ (respectively $m(x,l)$) denotes the lower bound of the interval constraints for x encoded in s (respectively in l). Dually $M(x,s)$ (resp. $M(x,l)$) denotes the upper bound of the interval for x in s (resp. in l).

Using these auxiliary functions we compute the interval constraints for w (denoted by $m_w(s)$, $M_w(s)$, $m_w(l)$ and $M_w(l)$).

In the definition of $(F(f))(x,y)$ the first three cases encode the previously computed lower interval bound for w. Similarly, the next three cases encode the upper interval bound for w.

If at least one of the variables (say u) is positive, then $w = u \cdot v$ is equivalent to $v = \frac{1}{u} \cdot w$, which yields hexagonal constraints on (v,w) and (w,v), i.e. if $0 < m(u,s) \leqslant u \leqslant M(u,s)$, then $w \leqslant M(u,s) \cdot v$ and $v \leqslant \frac{1}{m(u,s)} \cdot w$ (and the same for $m(u,l)$ and $M(u,l)$). Additionally, if $v \leqslant c \cdot z$, then $w \leqslant M(u,s) \cdot c \cdot z$, which explains the seventh and eighth cases of the definition of $F(f)$.

All other constraints that involve w are invalidated and replaced by constraints inferred as a transitive closure of constraints not affected by the multiplication and those described above. The algorithm for computing the transitive closure is presented in detail in Section 3.3.

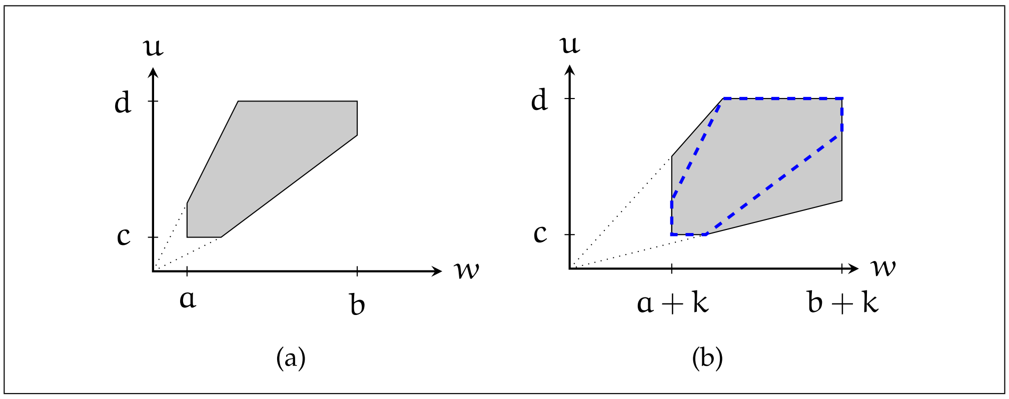

Figure 3.11: Weighted hexagon before (a) and after (b) assignment $w \leftarrow w + k$; the dashed shape represents the exact result of the assignment.

3.1.3 *Abstract Semantics of Boolean Predicates*

We show now how the boolean predicate $u \leqslant w$ (where $u, w \in \mathcal{Var}$) is interpreted in the domain of weighted hexagons, i.e. we provide the definition of $\pi_h(u \leqslant w, (s, l)) = \big((s_{\mathsf{True}}, l_{\mathsf{True}}), (s_{\mathsf{False}}, l_{\mathsf{False}})\big)$. Intuitively, $(s_{\mathsf{True}}, l_{\mathsf{True}})$ represents the hexagon (s, l) with an additional constraint $u \leqslant w$. If there was no constraint on (u, w) in (s, l), then $u \leqslant w$ is just added. In the other case the stronger of the existing constraint and $u \leqslant w$ is kept. Similarly, $(s_{\mathsf{False}}, l_{\mathsf{False}})$ represents (s, l) with $w \leqslant u$ added (which over-approximates $\neg(u \leqslant w)$). The detailed definition is presented in Figure 3.10.

If $(s_{\mathsf{True}}, l_{\mathsf{True}})$ (or $(s_{\mathsf{False}}, l_{\mathsf{False}})$) is not satisfiable, it is replaced by $\perp_h$ (and the respective branch of the execution is discarded). An algorithm to determine the satisfiability of a pair of functions (s, l) is presented in Section 3.3.

The above arguments can be summarised by the following theorem:

Theorem 3.8. *The transfer function δ_h and abstract semantics of boolean predicates π_h are a sound abstraction of the static transfer function $\overrightarrow{\mathcal{T}}$ defined in Section 1.2.2.*

3.1.4 *Precision of the Transfer Function*

In the domain of weighted hexagons it is often not possible to provide exact transfer rules. We demonstrate this issue on a (degenerated) example of a binary plus, namely on $w \leftarrow w + k$, where $w \in \mathcal{Var}$, $k \in \mathbb{V}$ and $k \neq 0$ (which is equivalent to $w \leftarrow w + x$, where $w, x \in \mathcal{Var}$ and $x \leqslant k$ and $k \leqslant x$—the two inequalities can be encoded as constraints between x and one of the special variables c^-, c^0, c^+). The exact result of

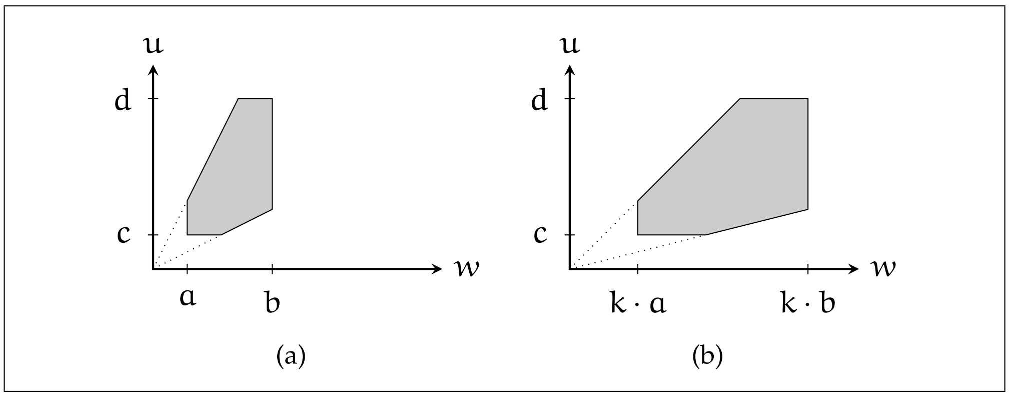

Figure 3.12: Weighted hexagon before (a) and after (b) assignment $w \leftarrow k \cdot w$

such an assignment (marked in Figure 3.11 as a dashed shape) cannot be represented as a weighed hexagon. Thus, $\delta_h(w \leftarrow w + k, (s, l))$ only over-approximates the exact result, as it is shown in Figure 3.11 using the grey polygon. Indeed, after the assignment, the constraint on (w, u) is described as $s'(w, u) = s(w, u) + s(x, u)$ (where x denotes the variable constantly equal to k). It is now easy to check that $s'(w, u) = s(w, u) + \frac{k}{d}$ (where d is the maximum value of the variable u, as in Figure 3.11), hence it is equal to the one depicted in Figure 3.11 (b). In the same manner one could justify all other constraints after this assignment.

Similar problems arise in all abstract domains. For example, in the domain of octagons, even the simplest multiplication $w \leftarrow k \cdot w$ for $w \in \mathcal{V}ar$ and $k \in \mathbb{V}_{\geqslant 0}$ cannot be handled exactly, while in the weighted hexagons the transfer function δ_h defined as in Figure 3.9 introduces no loss of precision (Figure 3.12). This fact follows directly from the seventh and eighth cases of the definition of $F(f)$ in Figure 3.9.

3.1.5 *Variable Introduction and Elimination*

Both the variable introduction and elimination are straightforward. Let $(s, l) \in \mathcal{H}(V)$. The elimination is defined as

$$(s, l)\downarrow_v \triangleq (s|_{(V \setminus \{v\}) \times (V \setminus \{v\})}, l|_{(V \setminus \{v\}) \times (V \setminus \{v\})}) .$$

Elimination defined in this way may loose some information that is entailed by (s, l). Using the transitive closure algorithm for (s, l) presented in Section 3.3 one can define the exact elimination. Let (s^*, l^*) denote the closure of (s, l):

$$(s, l)\downarrow_v \triangleq ((s^*)|_{(V \setminus \{v\}) \times (V \setminus \{v\})}, (l^*)|_{(V \setminus \{v\}) \times (V \setminus \{v\})}) .$$

The introduction puts Nil on $s(x,y)$ and $l(x,y)$ whenever x or y is equal to the introduced variable v:

$$(s,l)\Uparrow_v(x,y) \triangleq \begin{cases} (s(x,y), l(x,y)) & \text{if } v \notin \{x,y\}, \\ (\mathsf{Nil}, \mathsf{Nil}) & \text{otherwise.} \end{cases}$$

We do not allow the elimination of any of the special variables c^-, c^0 or c^+.

3.2 GRAPH MODEL

We have defined a weighted hexagon as a pair of functions $(s,l) \in \mathcal{H}$ that represent a system of hexagonal constraints $\mathcal{I}$. It is not clear how to efficiently implement the domain using such representation of abstract states.

We present now another convenient way to represent hexagonal constraints which uses (weighted and directed) graphs. In this case, each vertex corresponds to a variable, while each edge represents a constraint between the two corresponding variables. A weight attached to an edge is equal to the coefficient in the constraint.

3.2.1 *Graphs*

We start by introducing some basic graph terminology. A *directed graph* (a *digraph*) is a pair $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is some finite set and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. The elements of $\mathcal{V}$ are called *vertices*, while elements of $\mathcal{E}$ are referred to as *edges*. We assume that the set $\mathcal{V}$ is linearly ordered. A non-empty sequence of vertices $p = \langle v_0, \dots, v_k \rangle$ for $k \geqslant 0$ is called a *path* if for each $i \in \{0, \dots, k-1\}$ $(v_i, v_{i+1}) \in \mathcal{E}$. A path p *has type* $u \rightsquigarrow v$ (we write $p\colon u \rightsquigarrow v$), if $p = \langle u, \dots, v \rangle$. A path $p\colon u \rightsquigarrow v$ is *simple* if each vertex appears in p at most once. A path $c\colon u \rightsquigarrow u$ is called a *cycle*. A cycle $c\colon u \rightsquigarrow u$ is simple, if the only repeated vertex is u (and it occurs only as the first and last vertex in c). Simple paths and circles will be denoted as $p\colon u \overset{\bullet}{\rightsquigarrow} v$ and $c\colon u \overset{\bullet}{\rightsquigarrow} u$, respectively.

Let S be some set. We say that an algebraic structure $\langle S, +, \mathbf{0} \rangle$ is a *monoid* if

1. for each $a, b \in S$ $a + b \in S$,
2. $+$ is associative, i.e. for each $a, b, c \in S$, $(a+b)+c = a+(b+c)$,
3. $a + \mathbf{0} = \mathbf{0} + a = a$.

We say that $\langle S, +, \cdot, \mathbf{0}, \mathbf{1} \rangle$ is a *semiring* if

1. $\langle S, +, \mathbf{0} \rangle$ and $\langle S, \cdot, \mathbf{1} \rangle$ are monoids,
2. $a \cdot \mathbf{0} = \mathbf{0} \cdot a = \mathbf{0}$ for each $a \in S$,
3. the $+$ operation is commutative and idempotent,
4. $\cdot$ distributes over $+$, i.e. $a \cdot (b + c) = a \cdot b + a \cdot c$ and $(a + b) \cdot c = a \cdot c + b \cdot c$.

We say that $\langle S, +, \cdot, \mathbf{0}, \mathbf{1}, \Sigma \rangle$ is a *closed semiring* if $\langle S, +, \cdot, \mathbf{0}, \mathbf{1} \rangle$ is a semiring and the summation operation $\Sigma \colon \mathcal{P}(S) \to S$ obeys the following properties:

- $\Sigma\emptyset = \mathbf{0}$,
- $\Sigma\{a\} = a$ for each $a \in S$,
- $\Sigma S_1 + \Sigma S_2 = \Sigma(S_1 \cup S_2)$,
- $\Sigma\{\Sigma S_i \mid i \in \mathbb{I}\} = \Sigma(\bigcup_{i \in \mathbb{I}} S_i)$,
- $\Sigma S_1 \cdot \Sigma S_2 = \Sigma\{a_1 \cdot a_2 \mid a_1 \in S_1, a_2 \in S_2\}$.

Let $\langle S, +, \cdot, \mathbf{0}, \mathbf{1}, \Sigma \rangle$ be a closed semiring. A *closure* of $a \in S$ (denoted by a^*) is defined as $\Sigma\{a^i \mid i \in \mathbb{N}\}$, where $a^0 \triangleq \mathbf{1}$ and $a^i \triangleq a \cdot a^{i-1}$.

A *weighted* digraph is a triple $\langle \mathcal{V}, \mathcal{E}, \omega \rangle$ such that $\mathcal{V}$ and $\mathcal{E}$ are vertices and edges as above and $\omega \colon \mathcal{V} \times \mathcal{V} \to S$ is a *weight function*. A weight $\Pi_{\mathcal{G}}$ of a (finite) path $p = \langle v_0, v_1 \ldots, v_k \rangle$ is given by

$$\Pi_{\mathcal{G}}(p) \triangleq \begin{cases} \mathbf{1} & \text{if } k = 0, \\ \omega(v_0, v_1) & \text{if } k = 1, \\ \omega(v_0, v_1) \cdot \Pi_{\mathcal{G}}(\langle v_1, \ldots v_k \rangle) & \text{if } k > 1. \end{cases}$$

A standard way to represent a weighted graph, is to keep its *adjacency matrix*, that is a two-dimensional array with the number of rows and columns equal to the number of vertices. The entry in the i-th row and j-th column is equal to the weight of the edge between v_i and v_j (and Nil when no such edge exists).

We present now the generalised transitive closure algorithm [1]. It is a generalisation of the well-known Floyd-Warshall algorithm [9] for finding shortest paths between all pairs of vertices and follows the ideas of Kleene [45] and McNaughton-Yamada [51] algorithms.

Execute the following steps:

1. $\Omega_0(v_i, v_j) \triangleq \begin{cases} \mathbf{1} + \omega(v_i, v_j) & \text{if } v_i = v_j, \\ \omega(v_i, v_j) & \text{otherwise.} \end{cases}$
2. for each $k \in \{1, \ldots, |\mathcal{V}|\}$, let $\Omega_k(v_i, v_j)$ be
$$\Omega_{k-1}(v_i, v_j) + \Omega_{k-1}(v_i, v_k) \cdot \left(\Omega_{k-1}(v_k, v_k)\right)^* \cdot \Omega_{k-1}(v_k, v_j),$$
3. $\Omega \triangleq \Omega_{|\mathcal{V}|}$.

It is easy to see that this algorithm performs $O(|\mathcal{V}|^3)$ steps (assuming that the $+, \cdot, ^*$ operations can be performed in the semiring in a constant time). We will refer to this algorithm as *generalised transitive closure algorithm*.

The computed output $\Omega\colon \mathcal{V} \times \mathcal{V} \to S$ of the algorithm is equal to $\Omega(u, v) \triangleq \Sigma\{\Pi_{\mathcal{G}}(p) \mid p\colon u \rightsquigarrow v\}$. Depending on the choice of the semiring, Ω can represent for instance transitive closure of the graph (i.e. $\Omega(u, v) = \mathsf{True}$ if and only if there is a path $p\colon u \rightsquigarrow v$), or the shortest paths between each two vertices.

3.2.2 *Graph Encoding of a Weighted Hexagon*

A weighted hexagon (s, l) can be encoded as a pair of weighted digraphs. We choose the set of vertices $\mathcal{V}$ as the set of variables $\mathcal{V}ar$ (we will use the notions of variables and vertices interchangeably). A pair $(x, y) \in \mathcal{V}ar \times \mathcal{V}ar$ is in $\mathcal{E}$ if and only if $s(x, y) \neq \mathsf{Nil}$.

The weighted hexagon (s, l) is now represented as weighted digraphs $\mathcal{G}_s = (\mathcal{V}, \mathcal{E}, s)$ and $\mathcal{G}_l = (\mathcal{V}, \mathcal{E}, l)$. In the former we use a closed semiring $S_s = \langle \mathbb{V}_{\geqslant 0} \cup \{\mathsf{Nil}, +\infty\}, \inf_{\leqslant^\star}, \cdot, \mathsf{Nil}, 1, \inf_{\leqslant^\star} \rangle$. In the latter we use $S_l = \langle \mathbb{V}_{\geqslant 0} \cup \{\mathsf{Nil}, +\infty\}, \sup_{\leqslant_\star}, \cdot, \mathsf{Nil}, 1, \sup_{\leqslant_\star} \rangle$. The closure a^* in S_s and b^* in S_l are equal to

$$a^* = \begin{cases} 1 & \text{if } 1 \leqslant^\star a, \\ 0 & \text{otherwise} \end{cases} \qquad b^* = \begin{cases} 1 & \text{if } b \leqslant_\star 1 \\ +\infty & \text{otherwise.} \end{cases}$$

Now, the result Ω_s of the generalised transitive closure algorithm applied to graph $\mathcal{G}_s$ for $u, v \in \mathcal{V}$ is equal to the infimum of weights of all paths $p\colon u \rightsquigarrow v$. Dually, the result $\Omega_l(u, v)$ of the algorithm for graph $\mathcal{G}_l$ is equal to the supremum of weights of all paths $p\colon u \rightsquigarrow v$.

We often call the weight of a path in $\mathcal{G}_s$ (or in $\mathcal{G}_l$) the *product length* of this path.

A typical pair of graphs $\mathcal{G}_s$ and $\mathcal{G}_l$ is depicted in Figure 3.13 (the inequalities among c^-, c^0 and c^+ are omitted for clarity).

$$\begin{cases} x \leqslant \frac{1}{2}y \\ x \leqslant 2y \\ x \leqslant 5 \\ -3 \leqslant x \\ 0 \leqslant z \end{cases}$$

(a)

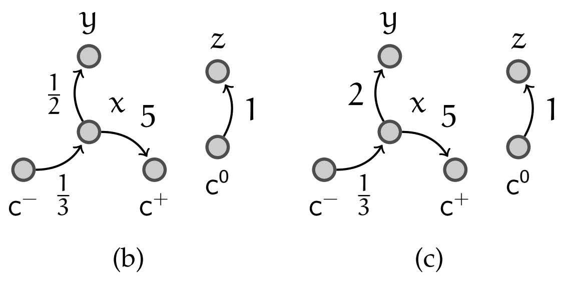

Figure 3.13: A system of inequalities (a) and its graph representation $\mathcal{G}_s$ (b) and $\mathcal{G}_l$ (c).

In graph $\mathcal{G}_s$ (or, equivalently, in $\mathcal{G}_l$) a variable u is *positive* if there exists in $\mathcal{E}$ a path $p\colon c^+ \rightsquigarrow u$ with weight $\Pi_{\mathcal{G}_s}(p) > 0$. Similarly u is called *negative* if there is a path $p\colon u \rightsquigarrow c^-$ with weight $\Pi_{\mathcal{G}_l}(p) > 0$. Intuitively, if a variable $u \in \mathcal{V}$ is positive, *all* valuations that satisfy the weighted hexagon (s, l) must assign to the variable u a positive value. Likewise, all such valuations must assign a negative value to each negative variable.

3.3 SATISFIABILITY TESTING AND NORMAL FORM

It may happen that multiple weighted hexagons have the same set of satisfying valuations. We present an algorithm that computes their normal form defined as the smallest (with respect to the lattice order in $\langle \mathcal{H}, \sqcap_h, \sqcup_h \rangle$) weighted hexagon that has the same set of solutions as the given one. The algorithm also determines whether the given weighted hexagon is satisfiable or not. We have already observed that the satisfiability test has to be performed while computing the meet as well as during the interpretation of boolean predicates.

The algorithm, for a weighted hexagon $a = (s, l)$ and the corresponding digraphs $\mathcal{G}_s, \mathcal{G}_l$ as above, finds the tightest possible constraints between all pairs of variables. In the graph model, this is achieved by finding in $\mathcal{G}_s$ and $\mathcal{G}_l$ paths between all pairs of vertices with smallest and largest product weights, respectively. This algorithm is based on the generalised transitive closure algorithm.

If the weighted hexagon (s, l) is not satisfiable, the algorithm returns False. Otherwise it computes a normal form $a^* \triangleq (s^*, l^*)$ that contains for each two variables $x, y \in \mathcal{V}ar$ the most extreme hexagonal constraints that are entailed by a.

Algorithm 1. Execute the following steps:

1. Add to (s, l) trivial constraints (such as $c^0 \leqslant 0 \cdot x$ or $x \leqslant +\infty \cdot c^+$) binding program variables to the artificial c^-, c^0, c^+. To achieve this, we define $(s', l')(x, y)$ as:

$$\begin{cases} (0,\ \max_{\leqslant_\star}(0, l(x,y))) & \text{if } x \in \{c^-, c^0\},\ y \notin \{c^-, c^0, c^+\} \\ (\min_{\leqslant^\star}(+\infty, s(x,y)),\ +\infty) & \text{if } y = c^+,\ x \notin \{c^-, c^0, c^+\} \\ (s, l)(x, y) & \text{otherwise.} \end{cases}$$

2. Find the tightest possible constraints entailed by s and l, using the generalised transitive closure algorithm in $\mathcal{G}_{s'}$ and $\mathcal{G}_{l'}$. Let s'' and l'' denote the respective outputs.

3. Identify all positive variables $P \triangleq \{x \mid 0 < s''(c^+, x)\}$ and negative variables $N \triangleq \{y \mid 0 < l''(y, c^-)\}$. If $P \cap N \neq \emptyset$ or $c^0 \in P$ or $c^0 \in N$, return False.

4. Find all vertices on cycles c and $\tilde{c}$ with product length $\Pi_{\mathcal{G}_{s''}}(c) < 1$ and $1 < \Pi_{\mathcal{G}_{l''}}(\tilde{c})$ characterised by: $\mathcal{X} \triangleq \{v \in \mathcal{Var} \mid s''(v, v) = 0\}$ and $\mathcal{Y} \triangleq \{v \in \mathcal{Var} \mid l''(v, v) = +\infty\}$. If $\mathcal{X}$ contains positive variables or $\mathcal{Y}$ contains negative ones, return False.

5. For x such that $s''(x, y) = 0$ for some y, let $(\bar{s}, \bar{l})(x, c^0) \triangleq (0, +\infty)$.

6. Apply the generalised transitive closure algorithm to $\mathcal{G}_{\bar{s}}$ and $\mathcal{G}_{\bar{l}}$. Let s^* and l^* denote the respective outputs. If there exist $s^*(c^+, v) = 0$ (which corresponds to an inequality $1 \leqslant 0 \cdot v$) or $l^*(v, c^-) = +\infty$ (that represents $v \leqslant -\infty$), return False. Otherwise return True.

In step 1 of Algorithm 1 we add some trivial constraints that must be included in the normal form, but cannot be found while computing the standard transitive closure, as they need not be a consequence of the original inequalities. In step 2 the generalised transitive closure algorithm is used to infer tightest possible constraints between each two variables. This step will generate also the diagonal constraints $x \leqslant 1 \cdot x$ (when no tighter are found). Step 3 allows us to find variables the value of which must be positive (or negative). In step 4 we check if there exists a positive variable x and an inequality $x \leqslant a \cdot x$ where $a < 1$ or a negative y and an inequality $y \leqslant b \cdot y$ where $b > 1$. Such inequalities cannot be satisfied. As for the step 5, if the generalised transitive closure algorithm yields $s''(x, y) = 0$, then it follows that $x \leqslant 0$, hence we add the (tightest possible) constraints on (x, c^0). Finally we propagate the modifications, using the transitive closure algorithm once more in step 6.

The theorems stated below express the main properties of Algorithm 1. We start by showing that the computed output (s^*, l^*) is equivalent to the input (s, l).

Theorem 3.9 (Correctness)**.** *If Algorithm 1 returns* True*, then the computed output* (s^*, l^*) *has the same set of satisfying valuations as the input* (s, l)*, i.e* $\gamma_h((s, l)) = \gamma_h((s^*, l^*))$.

Proof. We show that each satisfying valuation of the input (s, l) is also a satisfying valuation of the normal form and that each solution of the normal form satisfies the input. Details are postponed to Section 3.5.4 □

The following theorem shows that Algorithm 1 can be used as a satisfiability test.

Theorem 3.10 (Satisfiability Test)**.** *System* $\mathcal{I}$ *of hexagonal constraints is satisfiable if and only if Algorithm 1 returns* True *for the corresponding weighted hexagon* (s, l)*.*

Proof. If $\mathcal{I}$ is satisfiable, then no step of Algorithm 1 can return False. If the algorithm returns True, we construct a valuation ρ using the computed normal form (s^*, l^*) that satisfies (s, l) by Theorem 3.9. Details can be found in Section 3.5.5. □

Now we can show that the output of Algorithm 1 is indeed a normal form of all domain elements that have the same set of solutions.

Theorem 3.11 (Normal Form)**.** *If* (s, l) *is satisfiable then*

$$\bigsqcap_h \{c \in \mathcal{H} \mid \gamma_h(c) = \gamma_h((s, l))\}$$

is well defined and equal to (s^*, l^*)*.*

Proof. We show that the constraints in (s^*, l^*) cannot be tightened, hence $(s^*, l^*) \sqsubseteq_h (\tilde{s}, \tilde{l})$ for each $(\tilde{s}, \tilde{l})$ such that $\gamma_h((\tilde{s}, \tilde{l})) = \gamma_h((s, l))$. Details can be found in Section 3.5.6. □

As we have stated in Lemma 3.5, for each two weighted hexagons $a, b \in \mathcal{H}$, it holds that $\gamma_h(a) \cup \gamma_h(b) \subseteq \gamma_h(a \sqcup_h b)$. The following theorem shows that normalising a and b before performing the join, results in a smallest weighted hexagon, whose set of solutions over-approximates $\gamma_h(a) \cup \gamma_h(b)$:

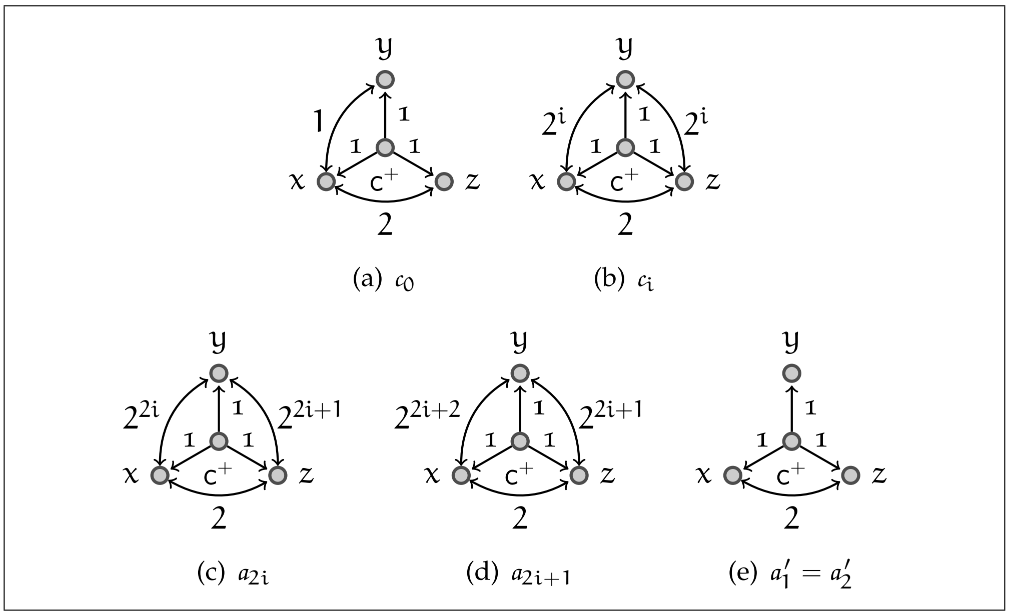

Figure 3.14: Normalisation of the result of widening may give a strictly increasing infinite sequence.

Theorem 3.12 (Best Approximation). *Normal forms a^* and b^* computed by Algorithm 1 can be used to find the smallest weighted hexagon, whose set of solutions over-approximates $\gamma_h(a) \cup \gamma_h(b)$:*

$$\gamma_h(a^* \sqcup_h b^*) = \inf_{\subseteq}\{\gamma_h(c) \mid c \in \mathcal{H} \wedge \gamma_h(c) \supseteq \gamma_h(a) \cup \gamma_h(b)\} .$$

Proof. Given $c \in \mathcal{H}$ such that $\gamma_h(c) \supseteq \gamma_h(a) \cup \gamma_h(b)$, we find for each pair of variables x, y two valuations $\rho_a \in \gamma_h(a^*)$ and $\rho_b \in \gamma_h(b^*)$ which ensure that $s_{a^* \sqcup_h b^*}(x, y) \leqslant^* s_c(x, y)$ (and $l_c(x, y) \leqslant_* l_{a^* \sqcup_h b^*}(x, y)$). This means that $a^* \sqcup_h b^* \sqsubseteq_h c$. Details can be found in Section 3.5.7. $\square$

The normalisation should not be applied in the widening, i.e. one should not replace $(s_{a \nabla_h b}, l_{a \nabla_h b})$ with $(s_{a \nabla_h b}, l_{a \nabla_h b})^*$. Figure 3.14 presents a sequence $c_0, c_1, \ldots$ that generates an infinite increasing sequence $a_0, a_1, \ldots$ given by $a_0 \triangleq c_0$ and $a_{i+1} \triangleq (a_i \nabla_h a_{i+1})^*$. For simplicity, only a part of the $\mathcal{G}_s$ graph is presented (the variables c^- and c^0 are not shown). Note that using the widening operator without normalisation, the sequence a' stabilises after the first step (Figure 3.14(e)).

3.4 FURTHER REMARKS

The domain of weighted hexagons is well defined when the set of numerical values $\mathbb{V}$ is chosen as real numbers $\mathbb{R}$. One can use also the rationals $\mathbb{Q}$, but in this case the lattice $\langle \mathcal{H}, \sqcup_h, \sqcap_h \rangle$ is not complete (as

$\langle \mathbb{Q}, \leqslant \rangle$ is not complete) and the concretisation γ_h does not determine a best abstraction. However, we can still work in a weaker formalisation of the abstract interpretation framework, as discussed in Section 1.2.8.

The domain cannot be used when $\mathbb{V}$ is chosen as integers $\mathbb{Z}$. The encoding of interval constraints (e.g. $x \in [5, 10]$) immediately results in a hexagonal constraint with a non-integer coefficient ($c^+ \leqslant \frac{1}{5} \cdot x$ in our example). Further, the normalisation algorithm would promote the non-integer coefficients to other constraints. The proofs of main theorems are not valid when $\mathbb{V} = \mathbb{Z}$. For instance, the proof of Theorem 3.10 would not ensure that any existing solution assigns an integer to each variable.

COMPUTATIONAL COMPLEXITY The representation of a system of hexagonal constraints consists of two two-dimensional matrices, thus it is quadratic in the number of variables. The most computationally complex operation is the normalisation. It relies on the generalised transitive closure algorithm and works in $O(|\mathcal{V}ar|^3)$ time.

NEGATIVE COEFFICIENTS We have restricted the possible values of coefficients to non-negative numbers. The solution we have developed cannot be applied to arbitrary (possibly negative coefficients). The crucial observation that it is enough to keep only the two extreme constraints is not valid in such a case. A more complicated representation that consists of four constraints (two extreme non-negative and two extreme negative) for each pair of variables would be necessary. It is also not possible to run two independent analyses that use our solution, one for non-negative and the other for non-positive coefficients. Non-positive coefficients are not closed under the multiplication (which is essential in the normalisation algorithm). We have performed some experiments, using a tool to gather statistics on Java programs [28], which have shown that positive coefficients are much more often used in real-life programs, thus we have decided to formally develop the domain only in the form presented in this chapter.

3.5 PROOFS

We recall here all theorems and lemmas stated throughout this chapter and present their proofs.

3.5.1 *Proof of Lemma 3.5*

Let us recall the lemma:

Lemma (3.5; recalled)**.** *For all $a, b \in \mathcal{H}$ we have that:*

1. $\gamma_h(a \sqcap_h b) = \gamma_h(a) \cap \gamma_h(b)$,
2. $\gamma_h(a \sqcup_h b) \supseteq \gamma_h(a) \cup \gamma_h(b)$.

Proof. Let us start with the proof of property 1. We first show the inclusion $\gamma_h(a \sqcap_h b) \subseteq \gamma_h(a) \cap \gamma_h(b)$. This is trivial when $a \sqcap_h b = \bot_h$, so assume that $a = (s_a, l_a)$, $b = (s_a, l_b)$ with $\gamma_h((s_{a\sqcap_h b}, l_{a\sqcap_h b})) \neq \emptyset$, where $s_{a\sqcap_h b}$ and $l_{a\sqcap_h b}$ are given by (3.3). We take $\rho \in \gamma_h((s_{a\sqcap_h b}, l_{a\sqcap_h b}))$ and consider arbitrary $x, y \in \mathcal{V}ar$. Suppose that $s_a(x,y)$ and $l_a(x,y)$ are not equal to Nil (the other case is trivial). Then $s_{a\sqcap_h b}(x,y) \neq \text{Nil}$ and $l_{a\sqcap_h b}(x,y) \neq \text{Nil}$. Hence both inequalities $\rho(x) \leqslant s_{a\sqcap_h b}(x,y) \cdot \rho(y)$ and $\rho(x) \leqslant l_{a\sqcap_h b}(x,y) \cdot \rho(y)$ hold. We are to show that also $\rho(x) \leqslant s_a(x,y) \cdot \rho(y)$ and $\rho(x) \leqslant l_a(x,y) \cdot \rho(y)$ must be true. Indeed, in case where $0 \leqslant \rho(y)$ we have that

$$\begin{aligned}\rho(x) \leqslant s_{a\sqcap_h b}(x,y) \cdot \rho(y) &= \min_{\leqslant^\star}\big(s_a(x,y), s_b(x,y)\big) \cdot \rho(y) \\ &\leqslant s_a(x,y) \cdot \rho(y) \leqslant l_a(x,y) \cdot \rho(y)\ .\end{aligned}$$

Similarly when $\rho(y) \leqslant 0$ we conclude that

$$\begin{aligned}\rho(x) \leqslant l_{a\sqcap_h b}(x,y) \cdot \rho(y) &= \max_{\leqslant_\star}\big(l_a(x,y), l_b(x,y)\big) \cdot \rho(y) \\ &\leqslant l_a(x,y) \cdot \rho(y) \leqslant s_a(x,y) \cdot \rho(y)\ .\end{aligned}$$

Therefore $\rho \in \gamma_h(a)$. By symmetry, we get that $\rho \in \gamma_h(b)$ as well.

We proceed now with the inclusion $\gamma_h(a) \cap \gamma_h(b) \subseteq \gamma_h(a \sqcap_h b)$. Suppose that $a = (s_a, l_a)$, $b = (s_a, l_b)$ (again, the case with $a = \bot_h$ or $b = \bot_h$ is trivial) and take $\rho \in \gamma_h(a) \cap \gamma_h(b)$. For a given pair of variables $x, y \in \mathcal{V}ar$ we consider values of $s_a(x,y)$ and $s_b(x,y)$. If they are both Nil then there is no constraint on (x,y) in $(s_{a\sqcap_h b}, l_{a\sqcap_h b})$.

When $s_a(x,y) \neq \text{Nil}$ and $s_b(x,y) = \text{Nil}$ (the case $s_b(x,y) \neq \text{Nil}$ and $s_a(x,y) = \text{Nil}$ is symmetric) then we know that $\rho(x) \leqslant s_a(x,y) \cdot \rho(y)$ holds. The inequality $\rho(x) \leqslant \min_{\leqslant^\star}\big(s_a(x,y), s_b(x,y)\big) \cdot \rho(y)$ holds as well, as $\min_{\leqslant^\star}(s_a(x,y), \text{Nil}) = s_a(x,y)$, so ρ satisfies $x \leqslant s_{a\sqcap_h b}(x,y) \cdot y$.

If $s_a(x,y) \neq \text{Nil}$ and $s_b(x,y) \neq \text{Nil}$ then both $\rho(x) \leqslant s_a(x,y) \cdot \rho(y)$ and $\rho(x) \leqslant s_b(x,y) \cdot \rho(y)$ hold, hence $\rho(x) \leqslant \min_{\leqslant^\star}\big(s_a(x,y), s_b(x,y)\big) \cdot \rho(y)$ holds as well. Again, this means that ρ satisfies $x \leqslant s_{a\sqcap_h b}(x,y) \cdot y$.

Using virtually the same argument we can justify the corresponding inequalities for $l_{a\sqcap_h b}$. This completes the proof of $\gamma_h(a \sqcap_h b) \supseteq \gamma_h(a) \cap \gamma_h(b)$.

Now we proceed with the proof of property 2. The inclusion is obvious when one of a or b is $\bot_h$, so assume that $a = (s_a, l_a)$ and $b = (s_b, l_b)$. Let ρ be an arbitrary valuation from $\gamma_h(a)$. When for some $x, y \in \mathcal{V}ar$ we have that $\max_{\leqslant^\star}(s_a(x,y), s_b(x,y))$ is greater than $\min_{\leqslant_\star}(l_a(x,y), l_b(x,y))$ then $s_{a \sqcup_h b}(x,y) = l_{a \sqcup_h b}(x,y) = \mathsf{Nil}$ (by the definition (3.4)) and so $(s_{a \sqcup_h b}, l_{a \sqcup_h b})$ introduces no constraint on variables x and y. The same happens when s_a and l_a (or s_b and l_b) are equal to Nil on (x, y).

So, assume otherwise: $s_a(x,y)$ and $l_a(x,y)$ are not equal to Nil and the inequalities

$$\rho(x) \leqslant s_a(x,y) \cdot \rho(y) \qquad \text{and} \qquad \rho(x) \leqslant l_a(x,y) \cdot \rho(y) \tag{3.13}$$

simultaneously hold. In case where $0 \leqslant \rho(y)$ we simply have

$$\begin{aligned}\rho(x) \leqslant s_a(x,y) \cdot \rho(y) \leqslant \max_{\leqslant^\star}(s_a(x,y), s_b(x,y)) \cdot \rho(y) \\ = s_{a \sqcup_h b}(x,y) \cdot \rho(y)\end{aligned}$$

using the first inequality from (3.13). For $\rho(y) \leqslant 0$ we first note that $s_{a \sqcup_h b}(x,y) = \max_{\leqslant^\star}(s_a(x,y), s_b(x,y)) \leqslant l_a(x,y)$ and obtain

$$\rho(x) \leqslant l_a(x,y) \cdot \rho(y) \leqslant s_{a \sqcup_h b}(x,y) \cdot \rho(y)$$

taking the latter inequality of (3.13). Similar argument settles the inequality $\rho(x) \leqslant l_{a \sqcup_h b}(x,y) \cdot \rho(y)$ thus proving that $\rho \in \gamma_h(a \sqcup_h b)$, hence $\gamma_h(a) \subseteq \gamma_h(a \sqcup_h b)$.

The same reasoning can be used to justify $\gamma_h(b) \subseteq \gamma_h(a \sqcup_h b)$. □

3.5.2 *Proof of Theorem 3.6*

We start by recalling the theorem:

Theorem (3.6; recalled). *Set $\mathcal{H}$ forms a lattice under $\sqcap_h$ and $\sqcup_h$ operators.*

Proof. Clearly, $\sqcap_h$ and $\sqcup_h$ are uniquely defined and commutative. We now establish the associativity, beginning with

$$a \sqcap_h (b \sqcap_h c) = (a \sqcap_h b) \sqcap_h c \qquad \text{for all } a, b, c \in \mathcal{H}\ . \tag{3.14}$$

By Lemma 3.5 we have that

$$\begin{aligned}\gamma_h\big(a \sqcap_h (b \sqcap_h c)\big) = \gamma_h(a) \cap \gamma_h(b \sqcap_h c) = \gamma_h(a) \cap \gamma_h(b) \cap \gamma_h(c) \\ = \gamma_h(a \sqcap_h b) \cap \gamma_h(c) = \gamma_h\big((a \sqcap_h b) \sqcap_h c\big)\ .\end{aligned}$$

Therefore the left-hand side of (3.14) is $\bot_h$ iff the right-hand side is equal to $\bot_h$. At this point we can assume that $a \sqcap_h (b \sqcap_h c) \neq \bot_h$ and $(a \sqcap_h b) \sqcap_h c \neq \bot_h$ (which implies $a \sqcap_h b \neq \bot_h$ and $b \sqcap_h c \neq \bot_h$). So neither a, b nor c is $\bot_h$ and we can assume that $a = (s_a, l_a)$, $b = (s_b, l_b)$ and $c = (s_c, l_c)$. With these settings we have the following identity (we omit variables $x, y \in \mathcal{Var}$ and write s instead of $s(x,y)$ for sake of brevity):

$$\begin{aligned} s_{a \sqcap_h (b \sqcap_h c)} &= \min_{\leqslant^\star}(s_a, s_{b \sqcap_h c}) = \min_{\leqslant^\star}(s_a, \min_{\leqslant^\star}(s_b, s_c)) \\ &= \min_{\leqslant^\star}(s_a, s_b, s_c) = \min_{\leqslant^\star}(\min_{\leqslant^\star}(s_a, s_b), s_c) \\ &= s_{(a \sqcap_h b) \sqcap_h c} \,. \end{aligned}$$

In the same manner, but using $\max_{\leqslant_\star}$ in place of $\min_{\leqslant^\star}$, we can derive that $l_{a \sqcap_h (b \sqcap_h c)} = l_{(a \sqcap_h b) \sqcap_h c}$, completing the proof of (3.14).

Now we are to demonstrate that

$$a \sqcup_h (b \sqcup_h c) = (a \sqcup_h b) \sqcup_h c \qquad \text{for all } a, b, c \in \mathcal{H}\,. \tag{3.15}$$

When at least one of a, b, c is $\bot_h$ then the equality is evident. Otherwise, we have the usual setting $a = (s_a, l_a)$, $b = (s_b, l_b)$ and $c = (s_c, l_c)$. If for some $x, y \in \mathcal{Var}$ $s(x,y) = l(x,y) = \mathsf{Nil}$ for any $(s,l) \in \{a, b, c\}$, then also $s_{a \sqcup_h (b \sqcup_h c)}(x,y) = s_{(a \sqcup_h b) \sqcup_h c}(x,y) = \mathsf{Nil}$ by the definition of $\max_{\leqslant^\star}$ and $\min_{\leqslant_\star}$. If none of the above is equal to Nil, consider the the right-hand side of (3.15). Values of $s_{a \sqcup_h b}$ and $l_{a \sqcup_h b}$ are given by (3.4) only if $\max_{\leqslant^\star}(s_a, s_b) \leqslant \min_{\leqslant_\star}(l_a, l_b)$ (we omit the obvious applications to variables x, y here and below for brevity). If additionally

$$\max_{\leqslant^\star}(\max_{\leqslant^\star}(s_a, s_b), s_c) \leqslant \min_{\leqslant_\star}(\min_{\leqslant_\star}(l_a, l_b), l_c) \tag{3.16}$$

then

$$\begin{aligned} s_{a \sqcup_h (b \sqcup_h c)} &= \max_{\leqslant^\star}(\max_{\leqslant^\star}(s_a, s_b), s_c) \\ &= \max_{\leqslant^\star}(s_a, \max_{\leqslant^\star}(s_b, s_c)) = s_{(a \sqcup_h b) \sqcup_h c} \end{aligned}$$

and similarly $l_{a \sqcup_h (b \sqcup_h c)} = l_{(a \sqcup_h b) \sqcup_h c}$. This is because

$$\begin{aligned} &\max_{\leqslant^\star}(s_b, s_c) \leqslant \max_{\leqslant^\star}(s_a, \max_{\leqslant^\star}(s_b, s_c)) \\ &= \max_{\leqslant^\star}(\max_{\leqslant^\star}(s_a, s_b), s_c) \leqslant \min_{\leqslant_\star}(\min_{\leqslant_\star}(l_a, l_b), l_c) \\ &\qquad = \min_{\leqslant_\star}(l_a, \min_{\leqslant_\star}(l_b, l_c)) \leqslant \min_{\leqslant_\star}(l_b, l_c)\,. \end{aligned}$$

Thus $s_{b \sqcup_h c}$ and $s_{a \sqcup_h (b \sqcup_h c)}$ are not Nil, but they are given by formulæ derived from (3.4). This completes the proof when (3.16) holds. Suppose that this is not the case. Then $s_{(a \sqcup_h b) \sqcup_h c} = \mathsf{Nil}$. If we had that $s_{a \sqcup_h (b \sqcup_h c)} \neq \mathsf{Nil}$ then it would be

$$\max_{\leqslant^\star}(s_a, s_b, s_c) \leqslant \min_{\leqslant_\star}(l_a, l_b, l_c) \tag{3.17}$$

which immediately implies (3.16) if $\max_{\leqslant^\star}(s_a, s_b) \leqslant \min_{\leqslant_\star}(l_a, l_b)$. A contradiction. The only remaining case is when $\max_{\leqslant^\star}(s_a, s_b)$ is greater than $\min_{\leqslant_\star}(l_a, l_b)$. Then we also have $s_{(a \sqcup_h b) \sqcup_h c} = \mathsf{Nil}$ and $s_{a \sqcup_h (b \sqcup_h c)} \neq \mathsf{Nil}$ would again imply (3.17)—contradiction as well.

We are left with showing the absorption laws, namely

$$a \sqcup_h (a \sqcap_h b) = a \tag{3.18}$$
$$a \sqcap_h (a \sqcup_h b) = a \tag{3.19}$$

for all $a, b \in \mathcal{H}$.

If $a = \bot_h$ then both sides of (3.18) are equal to $\bot_h$ as required. If $a \sqcap_h b = \bot_h$, then both sides of (3.18) are equal to a. Similarly for $a = \bot_h$ or $a \sqcup_h b = \bot_h$ and property (3.19). Thus we can consider the case where $a = (s_a, l_a)$, $b = (s_b, l_b)$ and, additionally for (3.18), $a \sqcap_h b \neq \bot_h$. Then for arbitrary variables x and y (again, we do not write them explicitly)

$$s_{a \sqcup_h (a \sqcap_h b)} = \max_{\leqslant^\star}(s_a, \min_{\leqslant^\star}(s_a, s_b)) = s_a$$

and similarly $l_{a \sqcup_h (a \sqcap_h b)} = \min_{\leqslant_\star}(l_a, \max_{\leqslant_\star}(l_a, l_b)) = l_a$ so $s_{a \sqcup_h (a \sqcap_h b)} \leqslant l_{a \sqcup_h (a \sqcap_h b)}$ as $s_a \leqslant l_a$. This proves (3.18).

It remains to establish (3.19) when $a \sqcup_h b \neq \bot_h$. If $s_{a \sqcup_h b} = l_{a \sqcup_h b} = \mathsf{Nil}$ (for some $x, y \in \mathcal{V}ar$) then $s_{a \sqcap_h (a \sqcup_h b)} = \min_{\leqslant^\star}(s_a, \mathsf{Nil}) = s_a$ and $l_{a \sqcap_h (a \sqcup_h b)} = l_a$. When $s_{a \sqcup_h b}$ and $l_{a \sqcup_h b}$ are not Nil then

$$s_{a \sqcap_h (a \sqcup_h b)} = \min_{\leqslant^\star}(s_a, \max_{\leqslant^\star}(s_a, s_b)) = s_a \; .$$

Much the same for $l_{a \sqcap_h (a \sqcup_h b)}$. This completes the proof of (3.19). □

3.5.3 *Proof of Theorem 3.7*

Theorem (3.7; recalled). *The operator $\triangledown_h$ given by*

$$a \triangledown_h b \triangleq \begin{cases} (s_{a \triangledown_h b}, l_{a \triangledown_h b}) & \textit{if } a = (s_a, l_a) \textit{ and } b = (s_b, l_b) \\ a & \textit{if } b = \bot_h, \\ b & \textit{otherwise,} \end{cases}$$

where for any $x, y \in \mathcal{V}ar$:

- *if $s_b(x, y) \leqslant^\star s_a(x, y)$ and $l_a(x, y) \leqslant_\star l_b(x, y)$ then:*

$$s_{a \triangledown_h b}(x, y) = s_a(x, y) \qquad \textit{and} \qquad l_{a \triangledown_h b}(x, y) = l_a(x, y) \; .$$

- $s_{a \nabla_h b}(x, y) = l_{a \nabla_h b}(x, y) = \mathrm{Nil}$ *otherwise*

meets the requirements imposed on a widening operator in Section 1.2.4.

Proof. First we prove the over-approximation property. For all $a, b \in \mathcal{H}$:

$$a \sqcup_h b \sqsubseteq_h a \nabla_h b \, . \tag{3.20}$$

If $a = \bot_h$, then both $a \sqcup_h b = b$ and $a \nabla_h b = b$, thus (3.20) holds. Similarly, if $b = \bot_h$ then $a \sqcup_h b = a \nabla_h b = a$. The only case left is when $a \neq \bot_h$ and $b \neq \bot_h$. By the definition of $\sqcup_h$ and Lemma 3.5 we also have $a \sqcup_h b \neq \bot_h$.

We establish

$$(a \sqcup_h b) \sqcap_h (a \nabla_h b) = a \sqcup_h b,$$

which is equivalent to (3.20). Further analysis of $a \nabla_h b$ for any pair of variables x and y leads to following cases (we omit the obvious applications to variables x, y here and below for brevity):

- $s_a = l_a = \mathrm{Nil}$. Then also $s_{a \nabla_h b} = l_{a \nabla_h b} = \mathrm{Nil}$ so $s_{(a \sqcup_h b) \sqcap_h (a \nabla_h b)} = s_{a \sqcup_h b}$ and $l_{(a \sqcup_h b) \sqcap_h (a \nabla_h b)} = l_{a \sqcup_h b}$.
- $s_b = l_b = \mathrm{Nil}$. Then (3.20) follows as in the previous case.
- $s_a \neq \mathrm{Nil}$, $s_b \neq \mathrm{Nil}$ with $s_b \leqslant s_a$ and $l_a \leqslant l_b$. Thus we can see that $s_{a \sqcup_h b} = \max_{\leqslant^\star}(s_a, s_b) = s_a$ and $l_{a \sqcup_h b} = \min_{\leqslant_\star}(l_a, l_b) = l_a$ so $s_{a \sqcup_h b} \leqslant l_{a \sqcup_h b}$. The definition of widening yields $s_{a \nabla_h b} = s_a$. Merging these observations together we get

$$\begin{aligned} s_{(a \sqcup_h b) \sqcap_h (a \nabla_h b)} &= \min_{\leqslant^\star}(s_{a \sqcup_h b}, s_{a \nabla_h b}) \\ &= \min_{\leqslant^\star}(s_a, s_a) = s_a = s_{a \sqcup_h b} \end{aligned}$$

  and similarly $l_{(a \sqcup_h b) \sqcap_h (a \nabla_h b)} = l_a = l_{a \sqcup_h b}$.
- $s_a \neq \mathrm{Nil}$ and $s_b \neq \mathrm{Nil}$, but either $s_b \nleqslant s_a$ or $l_a \nleqslant l_b$. In this case $s_{a \nabla_h b} = l_{a \nabla_h b} = \mathrm{Nil}$, which immediately gives $s_{(a \sqcup_h b) \sqcap_h (a \nabla_h b)} = s_{a \sqcup_h b}$ and similarly for $l_{(a \sqcup_h b) \sqcap_h (a \nabla_h b)}$.

This proves that ∇_h over-approximates $\sqcup_h$.

Now we prove the finite sequence property for ∇_h. Let us take an arbitrary infinite sequence of abstract states $c_0, c_1, \ldots$. We prove that the sequence $a_0, a_1, \ldots$ defined by $a_0 \triangleq c_0$ and $a_{i+1} \triangleq a_i \nabla_h c_{i+1}$ is not strictly increasing. Let us consider the following cases:

- there exists $i > 0$ such that $c_i = \bot_h$. In this case, by the definition of ∇_h we immediately get $a_{i-1} \nabla_h c_i = a_{i-1}$

- otherwise, for all $i > 0$, $c_i \neq \bot_h$. Then, for $i > 0$, $a_i \neq \bot_h$ ($a_i = \bot_h$ iff both $a_{i-1} = \bot_h$ and $c_i = \bot_h$). Let us consider the pair of constraints $s_{a_i}(x, y)$ and $l_{a_i}(x, y)$ in the step $i + 1$ of the widening. By the definition of ∇_h there are two possibilities:
  - both are replaced by Nil,
  - both remain unchanged.

  Note that a pair of constraints $s_{a_i}(x, y) = l_{a_i}(x, y) = \text{Nil}$ is always preserved by ∇_h: $s_{a_i \nabla_h c_{i+1}}(x, y) = l_{a_i \nabla_h c_{i+1}}(x, y) = \text{Nil}$.

  This means that for every pair of variables $x, y \in \mathcal{V}ar$ the constraints between them may be modified at most once in the whole widening sequence. As the number of variables is finite (say n), the total number of constraint modifications is not greater then n^2 and the sequence $\langle a_i \rangle$ stabilises after at most n^2 steps, i.e. there exists $1 \leqslant i \leqslant n^2$ such that $a_i \nabla_h c_{i+1} = a_i$.

We have shown that the finite sequence property holds for ∇_h. This completes the proof of Theorem 3.7. □

3.5.4 *Proof of Theorem 3.9*

We prove now Theorem 3.9 that formalises the correctness of Algorithm 1:

Theorem (3.9; recalled). *If Algorithm 1 returns* True, *then the computed output* (s^*, l^*) *has the same set of satisfying valuations as the input* (s, l), *i.e* $\gamma_h((s, l)) = \gamma_h((s^*, l^*))$.

Proof. We show that no step of the algorithm changes the set of solutions of the given weighted hexagon (s, l).

In the first step a new weighted hexagon (s', l') is constructed, by adding some trivial constraints to (s, l). Each of the newly added constraints must be satisfied by each solution of (s, l), thus $\gamma_h((s, l)) \subseteq \gamma_h((s', l'))$. For each $x, y \in \mathcal{V}ar$, (s', l') satisfies $s'(x, y) \leqslant^\star s(x, y)$ and $l(x, y) \leqslant_\star l'(x, y)$. Thus $(s', l') \sqsubseteq_h (s, l)$ and using monotonicity of γ_h we get $\gamma_h((s', l')) \subseteq \gamma_h((s, l))$.

In step 2 the shortest paths algorithm is used to generate (s'', l''). From the correctness of this algorithm, we immediately get that for each $x, y \in \mathcal{V}ar$, $s''(x, y) \leqslant^\star s'(x, y)$ and $l'(x, y) \leqslant_\star l''(x, y)$, hence $(s'', l'') \sqsubseteq_h (s', l')$. As γ_h is monotone, we get $\gamma_h((s'', l'')) \subseteq \gamma_h((s', l'))$.

We are left with showing the inclusion $\gamma_h((s', l')) \subseteq \gamma_h((s'', l''))$. Assume that $\rho \in \gamma_h((s', l'))$. Consider any pair of variables $x, y \in \mathcal{V}ar$.

We show that ρ fulfils the constraint $x \leqslant^\star s'(x,y) \cdot y$. If $s''(x,y) = \mathsf{Nil}$, then there is no constraint in (s'', l'') on (x,y), hence the property trivially holds.

Otherwise, ρ fulfils $\rho(x) \leqslant^\star \Pi_{\mathcal{G}_s}(p) \cdot \rho(y)$ for each $p\colon x \rightsquigarrow y$. Thus

$$\rho(x) \leqslant^\star \inf_{\leqslant^\star}\{\Pi_{\mathcal{G}_s}(p) \cdot \rho(y) \mid p\colon \ \rightsquigarrow y\}$$

which is equivalent to

$$\rho(x) \leqslant^\star \left(\Sigma\{\Pi_{\mathcal{G}_s}(p) \mid p\colon x \rightsquigarrow y\}\right) \cdot \rho(y)\ .$$

However, by the definition of s'', we get $s''(x,y) = \Sigma\{\Pi_{\mathcal{G}_s}(p) \mid p\colon x \rightsquigarrow y\}$, which finally gives $\rho(x) \leqslant^\star s''(x,y) \cdot \rho(y)$. In the same way one can show that $\rho(x) \leqslant l''(x,y) \cdot \rho(y)$, thus $\gamma_h((s',l')) \subseteq \gamma_h(s'',l''))$.

Steps 3 and 4 do not modify (s'', l''). In step 5 $(\bar{s}, \bar{l})$ is obtained from (s'', l'') by defining $(\bar{s}, \bar{l})(x, c^0) \triangleq (0, +\infty)$, if $s''(x,y) = 0$ for some $y \in \mathcal{Var}$. As each valuation ρ of (s'', l'') satisfies $x \leqslant 0 \cdot y$, then $\rho(x) \leqslant 0$, hence the new constraints on (x, c^0) are satisfied as well. This justifies that $\gamma_h((s'', l'')) \subseteq \gamma_h((\bar{s}, \bar{l}))$. Again, it is easy to see that $(\bar{s}, \bar{l}) \sqsubseteq_h (s'', l'')$, hence $\gamma_h((\bar{s}, \bar{l})) \subseteq \gamma_h((s'', l''))$ and, in consequence, $\gamma_h((\bar{s}, \bar{l})) = \gamma_h((s'', l''))$.

Finally, in the last step (s^*, l^*) is obtained as a result of the generalised transitive closure algorithm applied to $(\bar{s}, \bar{l})$. The argument used to establish the correctness of step 2 can be repeated to show that $\gamma_h((\bar{s}, \bar{l})) = \gamma_h((s^*, l^*))$. This completes the proof of Theorem 3.9.

□

3.5.5 *Proof of Theorem 3.10*

Again, let us first restate the theorem:

Theorem (3.10; recalled). *$\mathcal{I}$ is satisfiable if and only if Algorithm 1 returns* True *for the corresponding weighted hexagon (s, l).*

Proof. If the algorithm returns True, we can construct a valuation $\sigma\colon \mathcal{Var} \to \mathbb{V}$ that satisfies $\mathcal{I}$ in the following way:

- if u was marked in the third step of the algorithm as positive then $\sigma(u) \triangleq \frac{1}{s^*(c^+,u)}$ (note that $s^*(c^+, u) > 0$),
- if u was marked in the third step of the algorithm as negative then $\sigma(u) \triangleq -l^*(u, c^-)$ (note that $l^*(u, c^-) < +\infty$),
- $\sigma(u) \triangleq 0$ in other cases.

Each inequality $x \leqslant a \cdot y$ from the original system $\mathcal{I}$ matches one of the following cases:

- x, y are positive. In this case $\sigma(x) = 1/s^*(c^+, x)$ and $\sigma(y) = 1/s^*(c^+, y)$. Because $s^*(c^+, y)$ denotes the smallest product length of all paths of type $c^+ \rightsquigarrow y$, we have that
$$s^*(c^+, y) \leqslant a \cdot s^*(c^+, x) \ .$$
Both $s^*(c^+, x) > 0$ and $s^*(c^+, y) > 0$ (because x, y are positive). Dividing both sides by $s^*(c^+, x) \cdot s^*(c^+, y)$ we get:
$$\frac{1}{s^*(c^+, x)} \leqslant a \cdot \frac{1}{s^*(c^+, y)} \ .$$
Hence σ satisfies the inequality $x \leqslant a \cdot y$, that is $\sigma(x) \leqslant a \cdot \sigma(y)$.

- x, y are both negative. Hence $\sigma(x) = -l^*(x, c^-)$ and $\sigma(y) = -l^*(y, c^-)$. Because $l^*(x, c^-)$ denotes the greatest product length of all paths between x and c^-, it holds that
$$a \cdot l^*(y, c^-) \leqslant l^*(x, c^-) \ .$$
Multiplying both sides by -1 we get
$$-l^*(x, c^-) \leqslant a \cdot \left(-l^*(y, c^-)\right) \ .$$
and therefore the valuation σ satisfies $x \leqslant a \cdot y$.

- In other cases x was not marked as positive and y was not marked as negative (the case when x is positive and y is negative is impossible, due to the properties of Algorithm 1). According to the definition of σ, $\sigma(x) \leqslant 0$ and $\sigma(y) \geqslant 0$. All coefficients in all inequalities are nonnegative and so valuation σ satisfies $x \leqslant a \cdot y$.

We argue now that if the algorithm returns False, then $\gamma_h((s, l)) = \emptyset$. As shown in the proof of Theorem 3.9 in Section 3.5.4, no step of the algorithm modifies the set of solutions of the considered weighed hexagon.

The algorithm returns False in the following cases:

- when c^0 is marked as positive (step 3). This corresponds to $c^+ \leqslant a \cdot c^0$, which cannot be satisfied,

- when c^0 is marked as negative (step 3). This represents $c^0 \leqslant a \cdot c^-$ (for $a > 0$), which cannot be satisfied,

- when a variable is positive and negative at once,
- when there exists a cycle c with a product length $\Pi_{\mathcal{G}_{s'}}(c) < 1$ that contains a positive variable x — from the properties of the shortest paths algorithm, it is enough to check $s''(x,x) = 0$ (in step 4). For each variable x marked as positive and each valuation ρ that satisfies (s,l), it must hold that $\rho(x) > 0$. Then the inequality $\rho(x) \leqslant s''(x,x) \cdot \rho(x)$ cannot be satisfied for $s''(x,x) < 1$, therefore $\gamma_h((s'',l'')) = \gamma_h((s',l')) = \gamma_h((s,l)) = \emptyset$,
- when there exists a cycle $\tilde{c}$ with a product length $1 < \Pi_{\mathcal{G}_{l'}}(\tilde{c})$ that contains a negative variable. Similar argument as above can be used to justify $\gamma_h((s,l)) = \emptyset$.
- The last case, when Algorithm 1 returns False, is when computed output (s^*,l^*) contains either $s^*(c^+,v) = 0$ (which cannot be satisfied, as it encodes $c^+ \leqslant 0 \cdot v$) or $l^*(v,c^-) = +\infty$ (which represents unsatisfiable $v \leqslant -\infty$).

This completes the proof that if the algorithm returns False, then the input (s,l) is not satisfiable. □

3.5.6 *Proof of Theorem 3.11*

We start with the following lemma which ensures that the constraints in the normal form (s^*,l^*) cannot be tightened:

Lemma 3.21. *For an output (s^*,l^*) of Algorithm 1 and a pair of variables $x,y \in \mathcal{V}ar$ one of the following conditions hold:*

1. *if $s^*(x,y) \neq \mathsf{Nil}$ and $l^*(x,y) \neq \mathsf{Nil}$, then*
   a) *for each $b < s^*(x,y)$ there exists a solution of (s^*,l^*) that violates $x \leqslant b \cdot y$ and*
   b) *for each $c > l^*(x,y)$ there exists a solution of (s^*,l^*) that violates $x \leqslant c \cdot y$.*
2. *if $s^*(x,y) = l^*(x,y) = \mathsf{Nil}$, then for each $c \in \mathbb{V}_{\geqslant 0}$ there is a solution of (s^*,l^*) that violates $x \leqslant c \cdot y$.*

Proof. Let us start with the first case. We prove only the case 1a, as the proof of 1b is almost identical. To begin with, observe that if $s^*(x,y) = 0$, then 1a is trivial, as there is no $b \in \mathbb{V}_{\geqslant 0}$ such that $b < s^*(x,y)$. Thus we assume that $s^*(x,y) > 0$.

Considering the possible markings of x done in Algorithm 1 we have the following cases:

i x is marked as negative. In this case $s^*(x,y) = 0$ for each $y \in \mathcal{Var}$, (since in step 1 of the algorithm we add $s'(c^-,y) = 0$) hence there is no $b \in \mathbb{V}_{\geqslant 0}$ such that $b < s^*(x,y)$, hence 1a holds,

ii x is marked neither as negative nor as positive,

iii x is marked as positive. In this case, let $(\widehat{s}, \widehat{l}) \triangleq (s^*, l^*)$.

In case of (ii) we transform (s^*, l^*) so that x must be positive, i.e we construct a weighted hexagon $(\widehat{s}, \widehat{l})$ that is equal to (s^*, l^*) except for the pair (c^+, x). Let $M = \frac{s^*(x,y)}{s^*(c^+,y)}$, when $s^*(c^+,y) \neq \mathsf{Nil}$ and $M = 1$ otherwise (note that thanks to step 6 in Algorithm 1, $s^*(c^+,y) \neq 0$).

If $s^*(x,c^+) = \mathsf{Nil}$, then we define

$$\widehat{s}(c^+,x) \triangleq \max(1, M) \quad \text{and} \quad \widehat{l}(c^+,x) \triangleq +\infty\,.$$

Otherwise, if $s^*(x,c^+) = 0$, then the situation is identical as in (i) (thanks to step 5 of Algorithm 1 and the constraint $s(c^0,y) = 0$ added in step 1). Thus we assume that $s^*(x,c^+) > 0$ and we define

$$\widehat{s}(c^+,x) \triangleq \max(\frac{1}{s^*(x,c^+)}, M) \quad \text{and} \quad \widehat{l}(c^+,x) \triangleq +\infty\,.$$

We have only added some new constraints, hence $(\widehat{s}, \widehat{l}) \sqsubseteq_h (s^*, l^*)$. It is easy to see that $(\widehat{s}, \widehat{l})$ is satisfiable. All newly introduced cycles must contain x and c^+, hence their product length cannot be smaller than $\widehat{s}(x,c^+) \cdot \widehat{s}(c^+,x) \geqslant 1$ (note that new cycles may occur only if there was a path $p\colon x \rightsquigarrow c^+$, thus $s^*(x,c^+) \neq \mathsf{Nil}$).

It is also important that the smallest product length of all paths $p\colon x \rightsquigarrow y$ in the graph $\mathcal{G}_{\widehat{s}}$ is still equal to $s^*(x,y)$. Any path $p\colon x \rightsquigarrow y$ that uses the only one added edge $\widehat{s}(c^+,x)$ must clearly contain a cycle $\langle x, \ldots c^+, x\rangle$. But the weight of each such cycle is greater or equal to 1, by the definition of $\widehat{s}(c^+,x)$. Thus, $\Pi_{\mathcal{G}_{\widehat{s}}}(p) \geqslant s^*(x,y)$.

The rest of the proof for cases (ii) and (iii) is identical. We construct another weighted hexagon $(\tilde{s}, \tilde{l})$ that is equal to $(\widehat{s}, \widehat{l})$, except for the pair (y,x), for which we define

$$\tilde{s}(y,x) \triangleq \frac{1}{\widehat{s}(x,y)} \quad \text{and} \quad \tilde{l}(y,x) \triangleq +\infty\,.$$

As $\tilde{s}(y,x) \leqslant^* \widehat{s}(y,x)$ and $\widehat{l}(y,x) \leqslant_* \tilde{l}(y,x)$ and no other constraints were modified, it holds that $(\tilde{s}, \tilde{l}) \sqsubseteq_h (\widehat{s}, \widehat{l})$, thus each solution of $(\tilde{s}, \tilde{l})$ is also a solution of $(\widehat{s}, \widehat{l})$, and in consequence of (s^*, l^*).

All new cycles added in $(\tilde{s}, \tilde{l})$ must contain x and y. The product length of each path $p\colon x \rightsquigarrow y$ is greater or equal than $\widehat{s}(x,y)$, hence no added cycle may have a product length less than 1. This means that $(\tilde{s}, \tilde{l})$ is satisfiable.

We have defined $(\tilde{s}, \tilde{l})$ so that each its solution ρ satisfies the equality $x = \tilde{s}(x, y) \cdot y$ (since $x \leqslant \tilde{s}(x, y) \cdot y$ and $y \leqslant \frac{1}{\tilde{s}(x,y)} \cdot x$) and $\rho(x) > 0$ and $\rho(y) > 0$ (because we have added a constraint on (c^+, x)). Such ρ would violate each constraint $x \leqslant b \cdot y$ for $b < \tilde{s}(x, y)$.

This observation, together with the fact that $\tilde{s}(x, y) = s^*(x, y)$ and $\gamma_h((\tilde{s}, \tilde{l})) \subseteq \gamma_h((s^*, l^*))$ completes the proof of 1a. An almost identical reasoning can be used to show 1b.

We are left with the case $s^*(x, y) = l^*(x, y) = \mathsf{Nil}$. Let us assume that a constraint $x \leqslant c \cdot y$ can be added to (s^*, l^*) without modifying the set of solutions. We proceed in the same way as in the proof of 1a. Only in the last step, in the definition of $(\tilde{s}, \tilde{l})$ we put:

$$\tilde{s}(y, x) \triangleq \frac{1}{c + 1} \quad \text{and} \quad \tilde{l}(y, x) \triangleq +\infty \, .$$

We conclude that each solution ρ of $(\tilde{s}, \tilde{l})$ satisfies $x \geqslant (c + 1) \cdot y$, which cannot be satisfied together with $x \leqslant c \cdot y$ (as $\rho(x) > 0$ and $\rho(y) > 0$), hence $x \leqslant c \cdot y$ cannot be added to (s^*, l^*).

This completes the proof of Lemma 3.21.

Theorem 3.11 is a straightforward consequence of Lemma 3.21:

Theorem (3.11; recalled)**.** *If* (s, l) *is satisfiable then*

$$\bigsqcap_h \{c \in \mathcal{H} \mid \gamma_h(c) = \gamma_h((s, l))\}$$

is well defined and equal to (s^*, l^*).

Let (s', l') be a weighted hexagon such that $\gamma_h((s', l')) = \gamma_h((s^*, l^*))$. Using Lemma 3.21 for each pair of variables $x, y \in \mathcal{V}ar$ we get

- $s^*(x, y) \leqslant^\star s'(x, y)$,
- $l'(x, y) \leqslant_\star l^*(x, y)$.

This immediately gives $(s^*, l^*) \sqcup_h (s', l') = (s', l')$, i.e. $(s^*, l^*) \sqsubseteq_h (s', l')$. This holds for every element of the set $X \triangleq \{a \mid \gamma_h(a) = \gamma_h((s^*, l^*))\}$, so in consequence we get that (s^*, l^*) is the least element of X. □

3.5.7 *Proof of Theorem 3.12*

Let us first recall Theorem 3.12:

Theorem (3.12; recalled). *Normal forms a^* and b^* computed by Algorithm 1 can be used to find the smallest weighted hexagon, whose set of solutions over-approximates $\gamma_h(a) \cup \gamma_h(b)$:*

$$\gamma_h(a^* \sqcup_h b^*) = \inf_{\subseteq}\{\gamma_h(c) \mid c \in \mathcal{H} \wedge \gamma_h(c) \supseteq \gamma_h(a) \cup \gamma_h(b)\} .$$

Proof. We start with showing the following property:

$$a^* \sqcup_h b^* = \bigsqcap_h \{c \mid c \in \mathcal{H} \wedge \gamma_h(c) \supseteq \gamma_h(a) \cup \gamma_h(b)\} . \tag{3.22}$$

If $a^* = \bot_h$ or $b^* = \bot_h$ then the above property is evident. So suppose that $a^* = (s^*_a, l^*_a)$ and $b^* = (s^*_b, l^*_b)$. Consider any $c \in \mathcal{H}$ such that $\gamma_h(c) \supseteq \gamma_h(a) \cup \gamma_h(b)$. Clearly $c \neq \bot_h$, hence we may write $c = (s_c, l_c)$. We show now that $a^* \sqcup_h b^* \sqsubseteq_h c$.

For each pair of variables $x, y \in \mathcal{V}ar$ one of the following cases holds:

- $s^*_a(x,y) = l^*_a(x,y) = \mathsf{Nil}$ or $s^*_b(x,y) = l^*_b(x,y) = \mathsf{Nil}$. Using a technique almost identical as in the proof of Lemma 3.21(2), it can be shown that $s_c(x,y) = l_c(x,y) = \mathsf{Nil}$,

- $s^*_a(x,y) > 0$ and $s^*_b(x,y) > 0$. Let $\rho_a \in \gamma_h(a^*)$ and $\rho_b \in \gamma_h(b^*)$ be chosen so that $\rho_a(x) > 0$ and $\rho_a(x) = s^*_a(x,y) \cdot \rho_a(y)$, and dually $\rho_b(x) > 0$ and $\rho_b(x) = s^*_b(x,y) \cdot \rho_b(y)$ (existence of such valuations can be shown in the same way as in the proof of Lemma 3.21(1a)). Note that $\rho_a, \rho_b \in \gamma_h(a) \cup \gamma_h(b)$, hence $\rho_a, \rho_b \in \gamma_h(c)$. It is now easy to see that $s_{a^* \sqcup_h b^*}(x,y) = \max(s^*_a(x,y), s^*_b(x,y)) \leqslant s_c(x,y)$ (otherwise either ρ_a or ρ_b would violate $x \leqslant s_c(x,y) \cdot y$),

- exactly one of $s^*_a(x,y)$ and $s^*_b(x,y)$ is equal to 0 (say $s^*_a(x,y) = 0$). Using the same argument as above we show that $s_{a^* \sqcup_h b^*}(x,y) = \max(0, s^*_b(x,y)) \leqslant s_c(x,y)$,

- $s^*_a(x,y) = s^*_b(x,y) = 0$. In this case $s_{a^* \sqcup_h b^*}(x,y) = 0$ and clearly $0 \leqslant s_c(x,y)$.

Using the same type of reasoning we justify that $l_c(x,y) \leqslant_\star l_{a^* \sqcup_h b^*}(x,y)$. This means that $a^* \sqcup_h b^* \sqsubseteq_h c$ and completes the proof of (3.22).

As γ_h is monotone, using Theorem 3.9 we have $\gamma_h(a) \cup \gamma_h(b) = \gamma_h(a^*) \cup \gamma_h(b^*)$ and consequently, from Lemma 3.5 we get $\gamma_h(a) \cup \gamma_h(b) \subseteq \gamma_h(a^* \sqcup_h b^*)$ which together with (3.22) immediately implies Theorem 3.12. □

4 STRICT WEIGHTED HEXAGONS

The domain of weighted hexagons developed in the previous chapter, can be used only to model systems of non-strict hexagonal constraints, such as $x \leqslant a \cdot y$, where $x, y \in \mathcal{Var}$ and $a \in \mathbb{V}_{\geqslant 0}$. However, strict constraints are in some contexts more likely to be used. For instance, in many programming languages arrays are indexed starting from zero. In this case, loops iterating over an array use typically a strict constraint in the guard, as shown in Figure 4.1.

To prove correctness of the array accesses in the loop body, an invariant $i \leqslant \frac{1}{2} \cdot \mathrm{Out.len} - 1$ is needed. Because both i and $\mathrm{Out.len}$ are integers, the inequality is equivalent to a strict constraint $i < \frac{1}{2} \cdot \mathrm{Out.len}$.

In this chapter we extend the domain of weighted hexagons so that it can be used to model systems that contain both strict and non-strict variants of hexagonal constraints.

4.1 SYSTEMS WITH STRICT CONSTRAINTS

We have already observed that to represent systems of standard (i.e. non-strict) hexagonal constraints, it is sufficient to keep only two extreme inequalities for each pair of variables. Let us now study some properties of systems that may contain also strict constraints.

```
**Require array** In
Out $\leftarrow$ **new array**$(2 \cdot \mathrm{In.len})$
$1 \leqslant \mathrm{Out.len} \leqslant 2 \cdot \mathrm{In.len} \wedge 1 \leqslant \mathrm{In.len} \leqslant \frac{1}{2} \cdot \mathrm{Out.len}$
$i \leftarrow 0$
**while** $i < \mathrm{In.len}$ **do**
    $\ldots 0 \leqslant i < \frac{1}{2} \cdot \mathrm{Out.len} \ldots$
    $\mathrm{Out}[2 \cdot i] \leftarrow \mathrm{In}[i]$
    $\mathrm{Out}[2 \cdot i + 1] \leftarrow \mathrm{In}[i]$
    $i \leftarrow i + 1$
**end while**
```

Figure 4.1: Duplication of an array

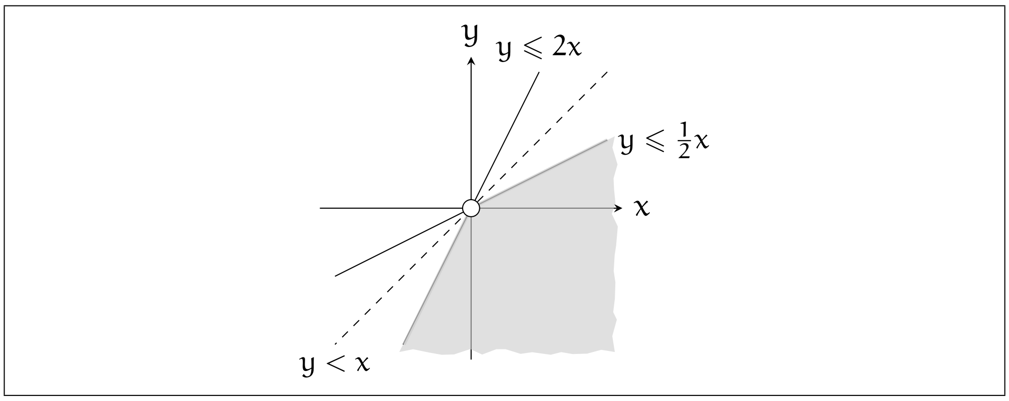

Figure 4.2: Keeping only extreme constraints may not be sufficient for systems containing strict inequalities.

If the system $\mathcal{I}$ contains $x \leqslant a \cdot y$ and $x < a \cdot y$ then the strict constraint always entails the non-strict one. Hence for the following system

$$\begin{cases} x \leqslant \frac{1}{2} \cdot y \\ x < \frac{1}{2} \cdot y \\ x \leqslant 5 \cdot y \\ x < 5 \cdot y \end{cases}$$

is equivalent to $\{x < \frac{1}{2} \cdot y, x < 5 \cdot y\}$. We may say that a strict constraint is always more "extreme" than a non-strict one with the same coefficient.

However, in some cases it is not enough to store for each pair of variables only the extreme constraints. In the example presented in Figure 4.2 the inequality $x < y$ carries information that cannot be deduced using only $y \leqslant \frac{1}{2} \cdot x$ and $y \leqslant 2 \cdot x$. In this case the extreme constraints are non-strict, hence they admit a valuation ρ such that $\rho(x) = \rho(y) = 0$, which is not a correct solution for the original system (as ρ does not fulfil $x < y$). This problem can be solved by keeping for each pair of variables $x, y \in \mathcal{V}ar$, in addition to the extreme constraints, also information if there exists a strict inequality $x < a \cdot y$ for some $a \in \mathbb{V}_{\geqslant 0}$.

Each of the extreme constraints can be represented as a pair $(a, t) \in \mathcal{C}onstr \triangleq (\mathbb{V}_{\geqslant 0} \cup \{+\infty\}) \times \mathcal{B}ool$, where a is equal to the corresponding coefficient, while $t \in \mathcal{B}ool$ is a *strictness indicator*: $t = \mathsf{True}$ if and only if the constraint is strict. We equip $\mathcal{Constr}$ with the order relations $\preceq$, $\preceq^{\bullet}$ and $\preceq_{\bullet}$ defined as:

$$\begin{aligned}(a_1, t_1) \preceq^{\bullet} (a_2, t_2) &\text{ iff } a_1 < a_2 \text{ or } (a_1 = a_2 \text{ and } t_1 \Rightarrow t_2)\\ (a_1, t_1) \preceq_{\bullet} (a_2, t_2) &\text{ iff } a_1 < a_2 \text{ or } (a_1 = a_2 \text{ and } t_2 \Rightarrow t_1)\\ \mathcal{a} \preceq \mathcal{b} &\text{ iff } \mathcal{a} \preceq_{\bullet} \mathcal{b} \text{ and } \mathcal{a} \preceq_{\bullet} \mathcal{b}\ .\end{aligned}$$

Intuitively, a bullet below the order symbol indicates that a strict constraint is in this order smaller than a non-strict with the same coefficient (i.e. $(a, \mathsf{True}) \preceq_{\bullet} (a, \mathsf{False})$). Dually, a bullet above the symbol means that the order treats strict constraints as greater than their non-strict counterparts, that is $(a, \mathsf{False}) \preceq^{\bullet} (a, \mathsf{True})$.

Each valuation ρ that satisfies two inequalities $x \leqslant a \cdot y$ and $y < b \cdot z$ satisfies also their transitive closure $x < a \cdot b \cdot z$. The resulting constraint is strict whenever either of the two inequalities was strict. This motivates the following definition of a multiplication $\otimes$: $\mathcal{Constr} \times \mathcal{Constr} \to \mathcal{Constr}$:

$$(a_1, t_1) \otimes (a_2, t_2) \triangleq (a_1 \cdot a_2, t_1 \vee t_2)\ .$$

We can encode numerical intervals (both strict and non-strict) as hexagonal constraints in the same manner as we did in the previous chapter. Thus, we assume that the three special variables c^-, c^0 and c^+ are contained in $\mathcal{Var}$ and that $\mathcal{I}$ contains the following trivial inequalities:

$$\begin{aligned}&\left\{c^- < 0 \cdot x \mid x \in \{c^-, c^0, c^+\}\right\}\\ \cup &\left\{c^- < +\infty \cdot x \mid x \in \{c^0, c^+\}\right\}\\ \cup &\left\{c^- \leqslant 1 \cdot c^-\right\}\\ \cup &\left\{c^0 \leqslant 0 \cdot x \mid x \in \{c^-, c^0, c^+\}\right\}\\ \cup &\left\{c^0 \leqslant +\infty \cdot c^0, c^0 < +\infty \cdot c^+\right\}\\ \cup &\left\{c^+ \leqslant 1 \cdot c^+, c^+ < +\infty \cdot c^+\right\}\ .\end{aligned}$$

We can now represent the extreme (smallest and largest) constraints as two functions s, l: $\mathcal{Var} \times \mathcal{Var} \to \mathcal{Constr} \cup \{\mathsf{Nil}\}$ that are defined as:

$$\begin{aligned}s(x, y) &\triangleq \min_{\preceq_{\bullet}} \left\{(a, \mathtt{strict}(\lhd)) \mid\ x \lhd a \cdot y \text{ is in } \mathcal{I}\right\}\\ l(x, y) &\triangleq \max_{\preceq^{\bullet}} \left\{(a, \mathtt{strict}(\lhd)) \mid\ x \lhd a \cdot y \text{ is in } \mathcal{I}\right\}\end{aligned} \qquad (4.1)$$

where $\lhd$ denotes $<$ or $\leqslant$ and $\mathtt{strict}(\lhd)$ is True if and only if $\lhd$ is $<$.

We introduce also an additional function $e : \mathcal{Var} \times \mathcal{Var} \to \mathcal{Bool} \cup \{\mathsf{Nil}\}$ (e stands for "evidence") to mark that there exists a strict inequality between a pair of variables:

$$e(x, y) \triangleq \mathsf{True} \text{ iff inequality } x < a \cdot y \text{ is in } \mathcal{I} \text{ for some } a \in \mathbb{V}_{\geqslant 0}\ .$$

As in the standard weighted hexagons, when there is no constraint between variables x and y, all s, l and e are equal to Nil.

For each order $\sqsubseteq \in \{\preceq, \preceq_\bullet, \preceq^\bullet\}$ we define the following orders on $\mathcal{Constr} \cup \{\mathsf{Nil}\}$:

$$\begin{aligned} &a \sqsubseteq_\star b \text{ iff } a = \mathsf{Nil} \text{ or } a \neq \mathsf{Nil}, b \neq \mathsf{Nil}, a \sqsubseteq b \\ &a \sqsubseteq^\star b \text{ iff } b = \mathsf{Nil} \text{ or } a \neq \mathsf{Nil}, b \neq \mathsf{Nil}, a \sqsubseteq b\ . \end{aligned}$$

Additionally, the orders $\preceq$, $\preceq_\bullet$ and $\preceq^\bullet$ are lifted to $\mathcal{Constr} \cup \{\mathsf{Nil}\}$, by putting $\mathsf{Nil} \sqsubseteq \mathsf{Nil}$ for each $\sqsubseteq \in \{\preceq, \preceq_\bullet, \preceq^\bullet\}$.

The operators for computing minimum and maximum for arbitrary, yet finite, number of elements, with respect to these orders are defined in a standard way.

The multiplication $\otimes$ is extended to $\mathcal{Constr} \cup \{\mathsf{Nil}\}$ by $a \otimes \mathsf{Nil} = \mathsf{Nil} \otimes a = \mathsf{Nil}$.

If for some pair of variables $x, y \in \mathcal{Var}$ (at least) one of the extreme constraints is strict, then clearly $e(x, y) = \mathsf{True}$. Additionally, for each $x, y \in \mathcal{Var}$ it holds $s(x, y) \preceq l(x, y)$. Any triple (s, l, e) that encodes some system $\mathcal{I}$ (thus, satisfies the above *well-formedness conditions*) will be called a *strict weighted hexagon*.

As we admit both booleans and Nil as valid values of the function e, we extend the standard boolean operators $\diamond \in \{\wedge, \vee\}$ to $\diamond_\star, \diamond^\star \colon (\mathcal{Bool} \cup \{\mathsf{Nil}\})^2 \to \mathcal{Bool} \cup \{\mathsf{Nil}\}$ by:

$$t_1 \diamond^\star t_2 \triangleq \begin{cases} \mathsf{Nil} & \text{if } t_1 = \mathsf{Nil} \text{ or } t_2 = \mathsf{Nil} \\ t_1 \diamond t_2 & \text{otherwise} \end{cases}$$

and dually

$$t_1 \diamond_\star t_2 \triangleq \begin{cases} t_1 & \text{if } t_2 = \mathsf{Nil} \\ t_2 & \text{if } t_1 = \mathsf{Nil} \\ t_1 \diamond t_2 & \text{otherwise}\ . \end{cases}$$

We can formalise now when a valuation ρ *satisfies* the strict weighted hexagon (s, l, e). We say that ρ satisfies a function $f \in \{s, l\}$ if it fulfils the following conditions:

1. $\rho(c^-) = -1$, $\rho(c^0) = 0$, $\rho(c^+) = 1$ and
2. for each $x, y \in \mathcal{Var}$ one of the following cases holds:
   - $f(x, y) = \mathsf{Nil}$ or
   - $f(x, y) = (a, \mathsf{True})$ and $\rho(x) < a \cdot \rho(y)$ or

- $f(x,y) = (a, \mathsf{False})$ and $\rho(x) \leqslant a \cdot \rho(y)$.

Similarly, we say that ρ satisfies $e\colon \mathcal{V}ar \times \mathcal{V}ar \to \mathcal{B}ool \cup \{\mathsf{Nil}\}$ if

1. $\rho(c^-) = -1$, $\rho(c^0) = 0$, $\rho(c^+) = 1$ and
2. for each $x, y \in \mathcal{V}ar$, if $e(x,y) = \mathsf{True}$ then $\rho(x) \neq 0$ or $\rho(y) \neq 0$.

Finally, ρ satisfies (s, l, e) if and only if it simultaneously satisfies s, l and e.

The correctness of representation defined in such a way is expressed by the following fact:

Fact. A valuation ρ satisfies (s, l, e) if and only if it is a solution of the system $\mathcal{I}$ represented by (s, l, e).

4.2 DOMAIN DEFINITION

The compact representation of systems of (strict and non-strict) hexagonal constraints described above will be used now to build the *abstract domain of strict weighted hexagons*

$$\mathsf{SH} = \langle \mathcal{SH}, \sqcup_{\mathsf{sh}}, \sqcap_{\mathsf{sh}}, \top_{\mathsf{sh}}, \bot_{\mathsf{sh}}, \gamma_{\mathsf{sh}}, \alpha_{\mathsf{sh}}, \delta_{\mathsf{sh}}, \pi_{\mathsf{sh}}, \nabla_{\mathsf{sh}} \rangle \ .$$

The set of abstract states $\mathcal{SH}$ consists of all satisfiable strict weighted hexagons and one special element $\bot_{\mathsf{sh}}$ with the concretisation γ_{sh} defined in the standard way:

$$\gamma_{\mathsf{sh}}(a) \triangleq \begin{cases} \emptyset & \text{if } a = \bot_{\mathsf{sh}}, \\ \{\rho \mid \rho \text{ satisfies } a\} & \text{otherwise.} \end{cases}$$

MEET The meet $a \sqcap_{\mathsf{sh}} b$ corresponds to the conjunction $\mathcal{I}_{a \sqcap_{\mathsf{sh}} b}$ of the systems $\mathcal{I}_a$ and $\mathcal{I}_b$ encoded as a and b (recall that a conjunction is just a union $\mathcal{I}_a \cup \mathcal{I}_b$), thus it should satisfy all constraints from a and b. In the components s and l we select the more extreme constraints. The conjunction $\mathcal{I}_{a \sqcap_{\mathsf{sh}} b}$ contains some strict inequality between a pair of variables, whenever any of $\mathcal{I}_a$ or $\mathcal{I}_b$ contained. Clearly, if $a = \bot_{\mathsf{sh}}$ or $b = \bot_{\mathsf{sh}}$, also the meet $a \sqcap_{\mathsf{sh}} b$ is equal to $\bot_{\mathsf{sh}}$. Otherwise we can define a triple $(\tilde{s}, \tilde{l}, \tilde{e})$:

$$\begin{aligned} \tilde{s}(x,y) &\triangleq \min_{\preceq^{\star}_{\bullet}} (s_a(x,y), s_b(x,y)), \\ \tilde{l}(x,y) &\triangleq \max_{\preceq^{\bullet}_{\star}} (l_a(x,y), l_b(x,y)) \qquad \text{and} \\ \tilde{e}(x,y) &\triangleq e_a(x,y) \vee_{\star} e_b(x,y) \ . \end{aligned}$$

Finally, we define the meet as

$$a \sqcap_{\mathsf{sh}} b \triangleq \begin{cases} (\tilde{s}, \tilde{l}, \tilde{e}) & \text{if } a \neq \bot_{\mathsf{sh}}, b \neq \bot_{\mathsf{sh}}, \gamma_{\mathsf{sh}}((\tilde{s}, \tilde{l}, \tilde{e})) \neq \emptyset, \\ \bot_{\mathsf{sh}} & \text{otherwise .} \end{cases}$$

We can state now the following lemma about the meet:

Lemma 4.2. *For all $a, b \in \mathcal{SH}$, $\gamma_{\mathsf{sh}}(a) \cap \gamma_{\mathsf{sh}}(b) = \gamma_{\mathsf{sh}}(a \sqcap_{\mathsf{sh}} b)$.*

Proof. The proof is essentially the same as in case of Lemma 3.5(1). We omit the details. □

JOIN The join $a \sqcup_{\mathsf{sh}} b$ is designed as an over-approximation of the alternative of $\mathcal{I}_a$ and $\mathcal{I}_b$. If $a = \bot_{\mathsf{sh}}$ then $a \sqcup_{\mathsf{sh}} b \triangleq b$; dually if $b = \bot_{\mathsf{sh}}$, then $a \sqcup_{\mathsf{sh}} b \triangleq a$. Otherwise, we may assume that $a = (s_a, l_a, e_a)$ and $b = (s_b, l_b, e_b)$ and define a triple $(\tilde{s}, \tilde{l}, \tilde{e})$ as the weaker constraints from a and b, for every $x, y \in \mathcal{V}ar$:

$$\begin{aligned} \tilde{s}(x,y) &\triangleq \max_{\preceq^{\star}_{\bullet}}(s_a(x,y), s_b(x,y)), \\ \tilde{l}(x,y) &\triangleq \min_{\preceq^{\bullet}_{\star}}(l_a(x,y), l_b(x,y)) \qquad \text{and} \\ \tilde{e}(x,y) &\triangleq e_b(x,y) \wedge^{\star} e_b(x,y) \ . \end{aligned}$$

However, $(\tilde{s}, \tilde{l}, \tilde{e})$ need not be a valid strict weighted hexagon, as it may violate the well-formedness conditions. It may happen that for some $x, y \in \mathcal{V}ar$ the requirement $\tilde{s}(x,y) \preceq \tilde{l}(x,y)$ is violated. In this case all constraints on (x,y) must be dropped. It may also happen that one of the extreme constraints (say $\tilde{s}(x,y)$) is strict (i.e. it is equal to (a, True)), while $\tilde{e}(x,y) = \mathsf{False}$. In this case, we restore the well-formedness property by relaxing the strict constraint (a, True) to its non-strict counterpart (a, False):

$$(\bar{s}, \bar{l}, \bar{e})(x,y) \triangleq \begin{cases} (\mathsf{Nil}, \mathsf{Nil}, \mathsf{Nil}) & \text{if } \neg\tilde{s}(x,y) \preceq \tilde{l}(x,y), \\ \big((a, t_1 \wedge t), (b, t_2 \wedge t), t\big) & \text{if } (\tilde{s}, \tilde{l}, \tilde{e})(x,y) = \\ & \quad = \big((a, t_1), (b, t_2), t\big), \\ (\tilde{s}, \tilde{l}, \tilde{e})(x,y) & \text{otherwise.} \end{cases} \tag{4.3}$$

It is now easy to check that $(\bar{s}, \bar{l}, \bar{e})$ is a valid strict weighted hexagon and that $\gamma_{\mathsf{sh}}((\bar{s}, \bar{l}, \bar{e})) \neq \emptyset$ (we have assumed that both a and b are satisfiable and $(\bar{s}, \bar{l}, \bar{e})$ is defined by taking less restrictive constraints from a and b). This allows us to define the join $a \sqcup_{\mathsf{sh}} b$ as

$$a \sqcup_{\mathsf{sh}} b \triangleq \begin{cases} a & \text{if } b = \bot_{\mathsf{sh}}, \\ b & \text{if } a = \bot_{\mathsf{sh}}, \\ (\bar{s}, \bar{l}, \bar{e}) & \text{otherwise.} \end{cases}$$

The join $a \sqcup_{\mathsf{sh}} b$ over-approximates the alternative of systems $\mathcal{I}_a$ and $\mathcal{I}_b$. This fact is formalised by the following lemma:

Lemma 4.4. *For all $a, b \in \mathcal{SH}$, $\gamma_{\mathsf{sh}}(a) \cup \gamma_{\mathsf{sh}}(b) \subseteq \gamma_{\mathsf{sh}}(a \sqcup_{\mathsf{sh}} b)$.*

Proof. The proof is essentially the same as in the case of Lemma 3.5(2). We skip here the details. □

Theorem 4.5. *Set $\mathcal{SH}$ forms a lattice under $\sqcap_{\mathsf{sh}}$ and $\sqcup_{\mathsf{sh}}$ operators.*

Proof. Direct examination of the required properties. Details can be found in Section 4.4.1. □

The special element $\bot_{\mathsf{sh}}$ is the least element in this lattice, while $\top_{\mathsf{sh}}$ equal to $\lambda x, y.(\mathsf{Nil}, \mathsf{Nil}, \mathsf{Nil})$ is the top. The lattice is complete, whenever $\langle \mathbb{V}_{\geqslant 0}, \leqslant \rangle$ is complete. In this case the abstraction function α_{sh} is uniquely defined by γ_{sh} as the lower adjoint of the Galois connection $\langle \mathcal{C}tx, \cup, \cap \rangle \underset{\alpha_{\mathsf{sh}}}{\overset{\gamma_{\mathsf{sh}}}{\leftrightarrows}} \langle \mathcal{SH}, \sqcup_{\mathsf{sh}}, \sqcap_{\mathsf{sh}} \rangle$.

WIDENING The domain of standard weighed hexagons has infinite height. Since each standard weighted hexagon can be monotonically encoded as a strict weighted hexagon, it immediately means that the domain of strict weighted hexagons also contains infinite strictly increasing sequences. Thus, a widening operator $\nabla_{\mathsf{sh}} \colon \mathcal{SH} \times \mathcal{SH} \to \mathcal{SH}$ must be introduced. It is defined in the same manner as in the standard weighted hexagons, namely we preserve only the weaker constraints from the first argument:

$$a \nabla_{\mathsf{sh}} b \triangleq \begin{cases} (\tilde{s}, \tilde{l}, \tilde{e}) & \text{if } a = (s_a, l_a, e_a) \text{ and } b = (s_b, l_b, e_b), \\ a & \text{if } b = \bot_{\mathsf{sh}}, \\ b & \text{otherwise,} \end{cases}$$

where for any $x, y \in \mathcal{V}ar$:

- if $s_b(x, y) \preceq^{\star}_{\bullet} s_a(x, y)$, $l_a(x, y) \preceq^{\bullet}_{\star} l_b(x, y)$ and $e_a(x, y) \Rightarrow e_b(x, y)$, then $\tilde{s}(x, y) \triangleq s_a(x, y)$, $\tilde{l}(x, y) \triangleq l_a(x, y)$ and $\tilde{e}(x, y) \triangleq e_a(x, y)$,

$$\tilde{e}(x,y) = \begin{cases} e(x,y) & \text{if } x \neq w, y \neq w, \\ \mathsf{False} & \text{if } x = y = w, \\ e(u,y) \vee^{\star} e(v,y) & \text{if } x = w, \\ e(x,u) \vee^{\star} e(x,v) & \text{if } y = w. \end{cases}$$

$$F(f)(x,y) = \begin{cases} f(x,y) & \text{if } x \neq w, y \neq w, \\ (1, \mathsf{False}) & \text{if } x = y = w, \\ (a+b, t_1 \vee t_2) & \text{if } x = w,\ f(u,y) = (a,t_1), \\ & \quad f(v,y) = (b,t_2), \\ (\frac{a \cdot b}{a+b}, t_1 \vee t_2) & \text{if } y = w,\ f(x,u) = (a,t_1), \\ & \quad f(x,v) = (b,t_2), a+b \neq 0, \\ \mathsf{Nil} & \text{otherwise} \end{cases}$$

$$\frac{}{\delta\big(w \leftarrow u+v, (s,l,e)\big) = (F(s), F(l), \tilde{e})}$$

Figure 4.3: Transfer function for a binary plus $w \leftarrow u+v$

- $\tilde{s}(x,y) = \tilde{l}(x,y) = \tilde{e}(x,y) = \mathsf{Nil}$ otherwise.

Theorem 4.6. *The operator* ∇_{sh} *is a widening operator for the domain* SH.

Proof. The proof is almost identical as for the corresponding Theorem 3.7 in the standard weighted hexagons; we omit the details. □

4.2.1 *Abstract Transfer Function*

The abstract transfer function δ_{sh} for the strict weighted hexagons is a straightforward extension of the transfer function for standard weighted hexagons presented in Section 3.1.2. We present here (see Figure 4.3) only the transfer rule for a binary plus: the detailed definition of $\delta(w \leftarrow u+v, (s,l,e))$. The abstract transfer rules for other statements and the abstract semantics of boolean predicates π_{sh} are obtained in a similar way from δ_{h} and π_{h} given in Section 3.1.2.

4.3 GRAPH REPRESENTATION & NORMALISATION

The lattice operations presented above require a feasible procedure to check the emptiness of γ_{sh}. In this section we present a graph-based representation of domain elements that is used in the satisfiability test and normalisation algorithm.

We represent a strict weighted hexagon (s, l, e) using weighted and directed graphs. As in the standard domain of weighted hexagons, the set of vertices $\mathcal{V}$ is chosen just as the set of variables $\mathcal{V}ar$. The set of edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ consists of these pairs of vertices (u, v), for which $s(u, v) \neq \mathsf{Nil}$.

A strict weighted hexagon (s, l, e) can be represented as a triple of graphs $\mathcal{G}_s = (\mathcal{V}, \mathcal{E}, s)$, $\mathcal{G}_l = (\mathcal{V}, \mathcal{E}, l)$ and $\mathcal{G}_e = (\mathcal{V}, \mathcal{E}, e)$. To apply the generalised transitive closure algorithm introduced in Section 3.2, we define the following closed semirings:

1. $\langle \mathcal{C}onstr \cup \{\mathsf{Nil}\}, \inf_{\preceq^{\star}_{\bullet}}, \otimes, \mathsf{Nil}, (1, \mathsf{False}), \inf_{\preceq^{\star}_{\bullet}} \rangle$ for the graph $\mathcal{G}_s$. In this case the closure a^* can be computed as:
$$a^* = \begin{cases} (0, t) & \text{if } a = (\alpha, t) \text{ and } \alpha < 1, \\ \inf_{\preceq^{\star}_{\bullet}}\{a, (1, \mathsf{False})\} & \text{otherwise.} \end{cases}$$
2. $\langle \mathcal{C}onstr \cup \{\mathsf{Nil}\}, \sup_{\preceq^{\bullet}_{\star}}, \otimes, \mathsf{Nil}, (1, \mathsf{False}), \sup_{\preceq^{\bullet}_{\star}} \rangle$ for the graph $\mathcal{G}_l$, with a closure a^* equal to
$$a^* = \begin{cases} (+\infty, t) & \text{if } a = (\alpha, t) \text{ and } 1 < \alpha, \\ \sup_{\preceq^{\bullet}_{\star}}\{a, (1, \mathsf{False})\} & \text{otherwise.} \end{cases}$$
3. $\langle \mathcal{B}ool \cup \{\mathsf{Nil}\}, \vee_\star, \vee^\star, \mathsf{Nil}, \mathsf{False}, \vee_\star \rangle$ for the graph $\mathcal{G}_e$, where the closure t^* is equal to
$$t^* = \mathsf{False} \vee_\star t \,.$$

Using these semirings, the generalised transitive closure algorithm applied to the graph $\mathcal{G}_s$ will compute for each pair of vertices $v_1, v_2 \in \mathcal{V}$ the infimum (with respect to the $\preceq^{\star}_{\bullet}$ order) of weights of all paths of type $v_1 \rightsquigarrow v_2$. Dually, it finds in $\mathcal{G}_l$ the supremum (with respect to $\preceq^{\bullet}_{\star}$) of weights of paths $p\colon v_1 \rightsquigarrow v_2$, for each $v_1, v_2 \in \mathcal{V}$. In case of $\mathcal{G}_e$ the algorithm determines for $v_1, v_2 \in \mathcal{V}$ whether there exists a path $p\colon v_1 \rightsquigarrow v_2$ that contains at least one edge (v_i, v_j) such that $e(v_i, v_j) = \mathsf{True}$.

We say that a path $p = \langle v_1, v_2, ..., v_k \rangle$ in $\mathcal{E}$ is *strict*, if $\mathcal{I}$ contains some strict inequality $v_j < \alpha \cdot v_{j+1}$ for any $j \in \{1, ..., k-1\}$. In other words, p is strict if and only if $e(v_j, v_{j+1}) = \mathsf{True}$ for some j.

We say that a vertex $u \in \mathcal{V}$ is *positive*, if in $\mathcal{E}$ there exists either a path $p\colon c^+ \rightsquigarrow u$ or a strict path $p\colon c^0 \rightsquigarrow u$ both such that $\Pi_{\mathcal{G}_s}(p) = (\alpha, t)$ for $0 < \alpha$. Similarly, u is *negative* if there is in $\mathcal{E}$ either a path $p\colon u \rightsquigarrow c^-$ such that $\Pi_{\mathcal{G}_l}(p) = (\alpha, t)$ for $0 < \alpha$ or a strict path $p\colon u \rightsquigarrow c^0$.

4.3.1 *Satisfiability Testing and Normal Form*

In this section we address the problem of finding one common representation for all strict weighted hexagons with the same set of satisfying valuations. The presented normalisation algorithm determines also whether the given input system (s, l, e) is satisfiable, i.e if $\gamma_{\mathsf{sh}}((s, l, e)) \neq \emptyset$. If yes, the algorithm computes the normal form $a^* = (s^*, l^*, e^*)$. The principles of this algorithm are similar as in the standard weighted hexagons, however they differ in some technical aspects.

Algorithm 2. Execute the following steps:

1. Add to the input system $\mathcal{I} = (s, l, e)$ trivial constraints relating program variables with the three artificial c^-, c^0 and c^+:

$$\begin{aligned} \mathcal{I}' \triangleq \mathcal{I} &\cup \left\{c^- < 0 \cdot x \mid x \in \mathcal{V}ar \setminus \{c^0, c^-, c^+\}\right\} \\ &\cup \left\{c^0 \leqslant 0 \cdot x \mid x \in \mathcal{V}ar \setminus \{c^0, c^-, c^+\}\right\} \\ &\cup \left\{x < +\infty \cdot c^+ \mid x \in \mathcal{V}ar \setminus \{c^0, c^-, c^+\}\right\} \end{aligned}$$

   Now let (s', l', e') denote the encoding of $\mathcal{I}'$; (s', l', e') can be computed from (s, l, e) in an obvious way, similarly as in the first step of Algorithm 1.

2. Apply the generalised transitive closure algorithm from Section 3.2 to the graphs $\mathcal{G}_{s'}$, $\mathcal{G}_{l'}$ and $\mathcal{G}_{e'}$. Let s'', l'' and e'' denote the respective outputs.

3. Find all positive and negative variables, let $P \triangleq \{x \mid e''(c^0, x) = \mathsf{True}, s''(c^0, x) = (a, t), a > 0\}$ and $N \triangleq \{y \mid e''(y, c^0) = \mathsf{True}\}$. If $P \cap N \neq \emptyset$ or $c^0 \in P \cup N$, return False.

4. Find cycles in (s'', l'', e'') that cannot be satisfied. Let
   - $\mathcal{X} \triangleq \{v \in \mathcal{V}ar \mid s''(v, v) = (a, t) \text{ and } (a, t) \preceq^{\star}_{\bullet} (1, \mathsf{True})\}$,
   - $\mathcal{Y} \triangleq \{v \in \mathcal{V}ar \mid l''(v, v) = (a, t) \text{ and } (1, \mathsf{True}) \preceq^{\bullet}_{\star} (a, t)\}$,

   If $\mathcal{X}$ contains positive variables or $\mathcal{Y}$ contains negative ones, return False.

5. Let $\mathcal{Z} \triangleq \{z \in \mathcal{V}ar \mid \exists_y \, s''(z, y) = (0, b)\}$. For every $z \in \mathcal{Z}$ let $b_z \triangleq \bigvee\{b \mid s''(z, y) = (0, b)\}$. We define $(\bar{s}, \bar{l}, \bar{e})$:
   - $\bar{s}(z, c^0) \triangleq (0, b_z)$,
   - $\bar{l}(z, c^0) \triangleq (+\infty, b_z)$

- $\overline{e}(z, c^0) \triangleq b_z \vee_\star e''(z, c^0)$

and $(\overline{s}, \overline{l}, \overline{e})(x, y) \triangleq (s'', l'', e'')(x, y)$ in all other cases.

6. Apply the transitive closure algorithm to $\mathcal{G}_{\overline{s}}$, $\mathcal{G}_{\overline{l}}$, and $\mathcal{G}_{\overline{e}}$ and denote the outputs as s^*, l^* and e^*, respectively. If any of the following cases holds:
   - $s^*(c^+, v) = (0, t)$ (corresponds to $1 \leqslant 0 \cdot v$ or $1 < 0 \cdot v$),
   - $s^*(c^0, v) = (0, \mathsf{True})$ (stands for $0 < 0 \cdot v$),
   - $s^*(v, v) = (1, \mathsf{True})$ or $l^*(v, v) = (1, \mathsf{True})$ (is equivalent to $v < v$),
   - $l^*(v, c^-) = (+\infty, t)$ (represents $v \leqslant -\infty$ or $v < -\infty$),
   - $e^*(c^0, c^0) = \mathsf{True}$ (meaning $0 < 0$)

   return False. Otherwise return True.

Intuitively, in the first step some trivial constraints (such as $c^- < 0 \cdot x$) are added. These constraints are always fulfilled, but they would not be found as a transitive closure of the existing ones. In step 2 we find the transitive closure of the constraints encoded in graphs $\mathcal{G}_{s'}$, $\mathcal{G}_{l'}$ and $\mathcal{G}_{e'}$ using the generalised transitive closure algorithm. In the next step we identify variables the value of which must be positive (or negative) in each solution of the input system. Clearly, if some variable must be positive and negative at once, the system is not satisfiable. In step 4 we check if there exists a cycle with weight (a, t) such that $a < 1$ (or a strict cycle with weight (a, True) such that $a \leqslant 1$) that contains a positive vertex. A satisfiable strict weighted hexagon cannot contain such a cycle. Similarly, we check if there is an unsatisfiable cycle with a negative variable. If the transitive closure contains for some $x, y \in \mathcal{V}ar$ an inequality $x \leqslant 0 \cdot y$ (or $x < 0 \cdot y$) a constraint $x \leqslant c^0$ (or $x < c^0$) must be added (step 5). Finally, in step 6 the changes are propagated (using the generalised transitive closure algorithm again). If the computed output does not contain an unsatisfiable constraint, the algorithm returns True.

The correctness of this algorithm is expressed by the following theorem:

Theorem 4.7. *If Algorithm 2 returns* True, *then the computed normal form* (s^*, l^*, e^*) *has the same set of solutions as the input* (s, l, e).

Proof. No step of the algorithm modifies the set of solutions of the input. The argument is essentially the same as for Theorem 3.9 for the standard weighted hexagons, thus we omit here the details. □

We need to show that the algorithm is a valid satisfiability test for strict weighted hexagons.

Theorem 4.8. *The set of solutions of the given input system* (s, l, e) *is not empty if and only if Algorithm 2 returns* True.

Proof. We transform the computed Strict Weighted Hexagon (s^*, l^*, e^*) into a standard Weighted Hexagon $(\tilde{s}, \tilde{l})$ that is enclosed in (s^*, l^*, e^*) and we argue that the satisfiability test defined for standard Weighted Hexagons returns True for $(\tilde{s}, \tilde{l})$. Details can be found in Section 4.4.2. □

Theorem 4.9 (Normal Form)**.** *If* $a = (s, l, e)$ *is satisfiable, then* (s^*, l^*, e^*) *is equal to*

$$\bigsqcap_{\mathsf{sh}}\{c \mid \gamma_{\mathsf{sh}}(c) = \gamma_{\mathsf{sh}}(a)\}\,.$$

Proof. (Sketch) We show that no constraint in the normal form can be tightened without modifying the set of solutions. In case of non-strict constraints the proof is identical as in the standard weighed hexagons (Theorem 3.11).

The proof can be also easily adapted to strict inequalities. The idea is to show that for a constraint $x < a \cdot y$ and arbitrary constant $0 \leqslant c < a$ the inequality $x < a \cdot y$ cannot be replaced by $x \leqslant c \cdot y$. We proceed as in the proof of Lemma 3.21, except of the last step, where we add a constraint $y \leqslant \frac{1}{c+\epsilon} \cdot x$ (where $c + \epsilon < a$) and argue that the system is still satisfiable. The added constraint ensures that for each solution ρ of the system $\rho(x) \geqslant (c + \epsilon) \cdot \rho(y)$, hence ρ would violate $x \leqslant c \cdot y$ (as we first ensure that $\rho(x) > 0$ and $\rho(y) > 0$). □

REMARKS The discussion concerning possible choices of $\mathbb{V}$ for the standard weighted hexagons (see Section 3.4) holds also for their strict version. Also the computational complexity is identical. An abstract state is represented as three matrices of size $O(|\mathcal{V}ar|^2)$ and the dominating normalisation operation is computed in $O(|\mathcal{V}ar|^3)$ time.

4.4 PROOFS

In this section we present proofs of theorems formulated in this chapter.

4.4.1 *Proof of Theorem 4.5*

Let us first recall the theorem:

Theorem (4.5; recalled)**.** *Set $\mathcal{SH}$ forms a lattice under $\sqcap_{\mathsf{sh}}$ and $\sqcup_{\mathsf{sh}}$ operators.*

Proof. Both $\sqcap_{\mathsf{sh}}$ and $\sqcup_{\mathsf{sh}}$ are symmetrical, so their commutativity is trivial. We show now the associativity of the join, i.e.

$$(a \sqcup_{\mathsf{sh}} b) \sqcup_{\mathsf{sh}} c = a \sqcup_{\mathsf{sh}} (b \sqcup_{\mathsf{sh}} c) \;. \tag{4.10}$$

If $a = \bot_{\mathsf{sh}}$, then $(a \sqcup_{\mathsf{sh}} b) \sqcup_{\mathsf{sh}} c = (\bot_{\mathsf{sh}} \sqcup_{\mathsf{sh}} b) \sqcup_{\mathsf{sh}} c = b \sqcup_{\mathsf{sh}} c$ and also $a \sqcup_{\mathsf{sh}} (b \sqcup_{\mathsf{sh}} c) = \bot_{\mathsf{sh}} \sqcup_{\mathsf{sh}} (b \sqcup_{\mathsf{sh}} c) = b \sqcup_{\mathsf{sh}} c$, so (4.10) holds. In the same way one can justify the case $b = \bot_{\mathsf{sh}}$ and $c = \bot_{\mathsf{sh}}$.

Let us assume that $a = (s_a, l_a, e_a)$, $b = (s_b, l_b, e_b)$ and $c = (s_c, l_c, e_c)$. Let us consider any pair $x, y \in \mathcal{V}ar$. If $s_a(x, y) = \mathsf{Nil}$, then $s_{a \sqcup_{\mathsf{sh}} b}(x, y) = \mathsf{Nil}$, thus also $s_{(a \sqcup_{\mathsf{sh}} b) \sqcup_{\mathsf{sh}} c}(x, y) = \mathsf{Nil}$. We also immediately get that $s_{a \sqcup_{\mathsf{sh}} (b \sqcup_{\mathsf{sh}} c)}(x, y) = \mathsf{Nil}$, hence (4.10) holds. In the same manner we proceed in the case when $s_b(x, y) = \mathsf{Nil}$ or $s_c(x, y) = \mathsf{Nil}$.

We assume now that $\mathsf{Nil} \notin \{s_a(x, y), s_b(x, y), s_c(x, y)\}$. Consider the case when $s_{a \sqcup_{\mathsf{sh}} b}(x, y) = \mathsf{Nil}$. By the definition of the join, this happens when $\neg(\max_{\preceq^{\star}_{\bullet}}(s_a, s_b) \preceq \min_{\preceq^{\bullet}_{\star}}(l_a, l_b))$ (for brevity, we omit the variables (x, y)). We immediately get $s_{(a \sqcup_{\mathsf{sh}} b) \sqcup_{\mathsf{sh}} c} = \mathsf{Nil}$. We show that also $s_{a \sqcup_{\mathsf{sh}} (b \sqcup_{\mathsf{sh}} c)} = \mathsf{Nil}$. If $s_{b \sqcup_{\mathsf{sh}} c} = \mathsf{Nil}$, then the property trivially holds. Otherwise $s_b \preceq^{\star}_{\bullet} \max_{\preceq^{\star}_{\bullet}}(s_b, s_c)$ and $\min_{\preceq^{\bullet}_{\star}}(l_b, l_c) \preceq^{\bullet}_{\star} l_b$ and we immediately get

$$\neg(\max_{\preceq^{\star}_{\bullet}}(s_a, \max_{\preceq^{\star}_{\bullet}}(s_b, s_c)) \preceq \min_{\preceq^{\bullet}_{\star}}(l_a, \min_{\preceq^{\bullet}_{\star}}(l_b, l_c))) \;.$$

This means that $s_{a \sqcup_{\mathsf{sh}} (b \sqcup_{\mathsf{sh}} c)} = \mathsf{Nil}$.

Consider now the case when $s_{a \sqcup_{\mathsf{sh}} b} \neq \mathsf{Nil}$. Assume also that $s_{b \sqcup_{\mathsf{sh}} c} \neq \mathsf{Nil}$ (the other case is symmetrical to the one presented above). Let us recall that each constraint is a pair (a, t) where $a \in \mathbb{V}_{\geqslant 0}$ is the coefficient, while $t \in \mathcal{B}ool$ is a strictness indicator. We focus first on the coefficients. Let a_a, a_b and a_c denote the coefficients in $s_a(x, y)$, $s_b(x, y)$ and $s_c(x, y)$, respectively. By the definition of the join, the coefficient $a_{(a \sqcup_{\mathsf{sh}} b) \sqcup_{\mathsf{sh}} c}$ is given by $\max_{\leqslant^{\star}}(\max_{\leqslant^{\star}}(a_a, a_b), a_c)$. We may use here max with respect the standard order in $\mathbb{V}_{\geqslant 0}$. In this case we immediately get

$$\begin{aligned} a_{(a \sqcup_{\mathsf{sh}} b) \sqcup_{\mathsf{sh}} c} &= \max_{\leqslant}(\max_{\leqslant}(a_a, a_b), a_c) \\ &= \max_{\leqslant}(a_a, \max_{\leqslant}(a_b, a_c)) = a_{a \sqcup_{\mathsf{sh}} (b \sqcup_{\mathsf{sh}} c)} \;. \end{aligned}$$

We show now the equality of the strictness indicators $t_{(a \sqcup_{\mathsf{sh}} b) \sqcup_{\mathsf{sh}} c}$ and $t_{a \sqcup_{\mathsf{sh}} (b \sqcup_{\mathsf{sh}} c)}$. If $\mathsf{False} \in \{e_a, e_b, e_c\}$, then using (4.3) we immediately get $t_{(a \sqcup_{\mathsf{sh}} b) \sqcup_{\mathsf{sh}} c} = t_{a \sqcup_{\mathsf{sh}} (b \sqcup_{\mathsf{sh}} c)} = \mathsf{False}$. In the other case (4.3) is just an identity and $s_{(a \sqcup_{\mathsf{sh}} b) \sqcup_{\mathsf{sh}} c} = \max_{\preceq^\star_\bullet}(\max_{\preceq^\star_\bullet}(s_a, s_b), s_c)$. When $\max_{\preceq^\star_\bullet}(s_a, s_b) = s_b$, we get

$$\begin{aligned} s_{(a \sqcup_{\mathsf{sh}} b) \sqcup_{\mathsf{sh}} c} &= \max_{\preceq^\star_\bullet}(\max_{\preceq^\star_\bullet}(s_a, s_b), s_c) \\ &= \max_{\preceq^\star_\bullet}(s_b, s_c) \\ &= \max_{\preceq^\star_\bullet}(s_a, \max_{\preceq^\star_\bullet}(s_b, s_c)) = s_{a \sqcup_{\mathsf{sh}} (b \sqcup_{\mathsf{sh}} c)} \ . \end{aligned}$$

We are left with the case when $\max_{\preceq^\star_\bullet}(s_a, s_b) = s_a$. If $\max_{\preceq^\star_\bullet}(s_b, s_c) = s_b$, then

$$\begin{aligned} s_{(a \sqcup_{\mathsf{sh}} b) \sqcup_{\mathsf{sh}} c} &= \max_{\preceq^\star_\bullet}(\max_{\preceq^\star_\bullet}(s_a, s_b), s_c) \\ &= \max_{\preceq^\star_\bullet}(s_a, s_c) \\ &= \max_{\preceq^\star_\bullet}(s_a, \max_{\preceq^\star_\bullet}(s_b, s_c)) = s_{a \sqcup_{\mathsf{sh}} (b \sqcup_{\mathsf{sh}} c)} \ . \end{aligned}$$

In the last case, when $\max_{\preceq^\star_\bullet}(s_b, s_c) = s_c$ we also immediately get $s_{(a \sqcup_{\mathsf{sh}} b) \sqcup_{\mathsf{sh}} c} = s_{a \sqcup_{\mathsf{sh}} (b \sqcup_{\mathsf{sh}} c)}$.

This completes the proof that $s_{(a \sqcup_{\mathsf{sh}} b) \sqcup_{\mathsf{sh}} c} = s_{a \sqcup_{\mathsf{sh}} (b \sqcup_{\mathsf{sh}} c)}$. In the same fashion we can justify the equality $l_{(a \sqcup_{\mathsf{sh}} b) \sqcup_{\mathsf{sh}} c} = l_{a \sqcup_{\mathsf{sh}} (b \sqcup_{\mathsf{sh}} c)}$. The proof for $e_{(a \sqcup_{\mathsf{sh}} b) \sqcup_{\mathsf{sh}} c} = e_{a \sqcup_{\mathsf{sh}} (b \sqcup_{\mathsf{sh}} c)}$ is trivial. This completes the proof of (4.10).

We show now the associativity of the meet, that is

$$(a \sqcap_{\mathsf{sh}} b) \sqcap_{\mathsf{sh}} c = a \sqcap_{\mathsf{sh}} (b \sqcap_{\mathsf{sh}} c) \ . \tag{4.11}$$

Note that $\gamma_{\mathsf{sh}}((a \sqcap_{\mathsf{sh}} b) \sqcap_{\mathsf{sh}} c) = \gamma_{\mathsf{sh}}(a) \cap \gamma_{\mathsf{sh}}(b) \cap \gamma_{\mathsf{sh}}(c)$ (by Lemma 4.2). If $\bot_{\mathsf{sh}} \in \{a, b, c, a \sqcap_{\mathsf{sh}} b, b \sqcap_{\mathsf{sh}} c\}$, then both sides of (4.11) are equal to $\bot_{\mathsf{sh}}$. Assume now that $\bot_{\mathsf{sh}} \notin \{a, b, c, a \sqcap_{\mathsf{sh}} b, b \sqcap_{\mathsf{sh}} c\}$. For every $x, y \in \mathcal{V}ar$ we have $s_{(a \sqcap_{\mathsf{sh}} b) \sqcap_{\mathsf{sh}} c} = \min_{\preceq^\star_\bullet}(\min_{\preceq^\star_\bullet}(s_a, s_b), s_c)$ (again, we do not write the variables explicitly).

If $\min_{\preceq^\star_\bullet}(s_a, s_b) = s_b$, then $s_{(a \sqcap_{\mathsf{sh}} b) \sqcap_{\mathsf{sh}} c} = \min_{\preceq^\star_\bullet}(s_b, s_c)$. But in this case $\min_{\preceq^\star_\bullet}(s_b, s_c) \preceq^\star_\bullet s_a$, thus

$$s_{a \sqcap_{\mathsf{sh}} (s_b \sqcap_{\mathsf{sh}} s_c)} = \min_{\preceq^\star_\bullet}(s_a, \min_{\preceq^\star_\bullet}(s_b, s_c)) = \min_{\preceq^\star_\bullet}(s_b, s_c)$$

and (4.11) holds.

We are left with the case when $\min_{\preceq^\star_\bullet}(s_a, s_b) = s_a$. This gives us $s_{(a \sqcap_{\mathsf{sh}} b) \sqcap_{\mathsf{sh}} c} = \min_{\preceq^\star_\bullet}(s_a, s_c)$. We have two cases: $\min_{\preceq^\star_\bullet}(s_b, s_c) = s_b$ and $\min_{\preceq^\star_\bullet}(s_b, s_c) = s_c$. In the first case we get $s_{(a \sqcap_{\mathsf{sh}} b) \sqcap_{\mathsf{sh}} c} = s_a$ and $s_{a \sqcap_{\mathsf{sh}} (b \sqcap_{\mathsf{sh}} c)} = \min_{\preceq^\star_\bullet}(s_a, \min_{\preceq^\star_\bullet}(s_b, s_a)) = \min_{\preceq^\star_\bullet}(s_a, s_b) = s_a$, hence (4.11) holds.

If $\min_{\preceq^\star_\bullet}(s_b, s_c) = s_c$, then $s_{a \sqcap_{\mathsf{sh}} (b \sqcap_{\mathsf{sh}} c)} = \min_{\preceq^\star_\bullet}(s_a, \min_{\preceq^\star_\bullet}(s_b, s_a)) = \min_{\preceq^\star_\bullet}(s_a, s_c)$ and again (4.11) holds.

A very similar argument can be used for the functions l and e. This ends the proof of the associativity of the meet.

It remains to prove the absorption properties. We start with

$$\mathfrak{a} \sqcup_{\mathsf{sh}} (\mathfrak{a} \sqcap_{\mathsf{sh}} \mathfrak{b}) = \mathfrak{a} \tag{4.12}$$

If $\mathfrak{a} = \bot_{\mathsf{sh}}$ then both sides of (4.12) are equal to $\bot_{\mathsf{sh}}$. If $\mathfrak{a} \sqcap_{\mathsf{sh}} \mathfrak{b} = \bot_{\mathsf{sh}}$, then both sides of (4.12) are equal to $\mathfrak{a}$.

Assume now that $(\mathfrak{a} \sqcap_{\mathsf{sh}} \mathfrak{b}) \neq \bot_{\mathsf{sh}}$. Again, we present a proof of (4.12) only for the function s. We consider an arbitrary pair of variables $x, y \in \mathcal{V}ar$ (we do not write them explicitly). If $s_{\mathfrak{a}} = \mathsf{Nil}$, then $s_{\mathfrak{a} \sqcup_{\mathsf{sh}} (\mathfrak{a} \sqcap_{\mathsf{sh}} \mathfrak{b}))} = \mathsf{Nil}$. If $s_{\mathfrak{b}} = \mathsf{Nil}$, then $s_{\mathfrak{a} \sqcup_{\mathsf{sh}} (\mathfrak{a} \sqcap_{\mathsf{sh}} \mathfrak{b}))} = s_{\mathfrak{a}}$. Let $s_{\mathfrak{a}} = (a_{\mathfrak{a}}, t_{\mathfrak{a}})$ and $s_{\mathfrak{b}} = (a_{\mathfrak{b}}, t_{\mathfrak{b}})$.

By the definitions of the meet and join we have that $a_{\mathfrak{a} \sqcup_{\mathsf{sh}} (\mathfrak{a} \sqcap_{\mathsf{sh}} \mathfrak{b})} = \max(a_{\mathfrak{a}}, \min(a_{\mathfrak{a}}, a_{\mathfrak{b}}))$. If $\min(a_{\mathfrak{a}}, a_{\mathfrak{b}}) = a_{\mathfrak{a}}$, then we immediately get $a_{\mathfrak{a} \sqcup_{\mathsf{sh}} (\mathfrak{a} \sqcap_{\mathsf{sh}} \mathfrak{b})} = \max(a_{\mathfrak{a}}, a_{\mathfrak{a}}) = a_{\mathfrak{a}}$. If $\min(a_{\mathfrak{a}}, a_{\mathfrak{b}}) = a_{\mathfrak{b}}$, then $a_{\mathfrak{a} \sqcup_{\mathsf{sh}} (\mathfrak{a} \sqcap_{\mathsf{sh}} \mathfrak{b})} = \max(a_{\mathfrak{a}}, a_{\mathfrak{b}}) = a_{\mathfrak{a}}$.

If $e_{\mathfrak{a}} = \mathsf{False}$, then $t_{\mathfrak{a}} = \mathsf{False}$ (well-formedness) and $t_{\mathfrak{a} \sqcup_{\mathsf{sh}} (\mathfrak{a} \sqcap_{\mathsf{sh}} \mathfrak{b})} = \mathsf{False}$ by (4.3). If $e_{\mathfrak{a}} = \mathsf{True}$ then $e_{\mathfrak{a} \sqcap_{\mathsf{sh}} \mathfrak{b}} = \mathsf{True}$ and $e_{\mathfrak{a} \sqcup_{\mathsf{sh}} (\mathfrak{a} \sqcap_{\mathsf{sh}} \mathfrak{b})} = \mathsf{True}$, thus $s_{\mathfrak{a} \sqcup_{\mathsf{sh}} (\mathfrak{a} \sqcap_{\mathsf{sh}} \mathfrak{b})} = \max_{\preceq^{\star}_{\bullet}}(s_{\mathfrak{a}}, \min_{\preceq^{\star}_{\bullet}}(s_{\mathfrak{a}}, s_{\mathfrak{b}}))$ and the reasoning presented for the coefficients $a_{\mathfrak{a}}$ and $a_{\mathfrak{b}}$ can be extended to constraints $s_{\mathfrak{a}}$ and $s_{\mathfrak{b}}$. This gives us $s_{\mathfrak{a} \sqcup_{\mathsf{sh}} (s_{\mathfrak{a}} \sqcap_{\mathsf{sh}} s_{\mathfrak{b}})} = s_{\mathfrak{a}}$.

The same argument works for the function l. The argument for the evidence function e is trivial. This ends the proof of (4.12).

Finally we show the absorption for the meet, that is

$$\mathfrak{a} \sqcap_{\mathsf{sh}} (\mathfrak{a} \sqcup_{\mathsf{sh}} \mathfrak{b}) = \mathfrak{a}\,. \tag{4.13}$$

If $\mathfrak{a} = \bot_{\mathsf{sh}}$, then both sides of (4.13) are equal to $\bot_{\mathsf{sh}}$. If $\mathfrak{b} = \bot_{\mathsf{sh}}$, then both sides of (4.13) are equal to $\mathfrak{a}$. Let us assume that $\mathfrak{a} \neq \bot_{\mathsf{sh}}$ and $\mathfrak{b} \neq \bot_{\mathsf{sh}}$. Consider arbitrary $x, y \in \mathcal{V}ar$ (again, they will not be written). If $s_{\mathfrak{a}} = \mathsf{Nil}$ then $s_{\mathfrak{a} \sqcup_{\mathsf{sh}} \mathfrak{b}} = \mathsf{Nil}$ and $s_{\mathfrak{a} \sqcap_{\mathsf{sh}} (\mathfrak{a} \sqcup_{\mathsf{sh}} \mathfrak{b})} = \mathsf{Nil}$. If $s_{\mathfrak{b}} = \mathsf{Nil}$ then $s_{\mathfrak{a} \sqcup_{\mathsf{sh}} \mathfrak{b}} = \mathsf{Nil}$ and $s_{\mathfrak{a} \sqcap_{\mathsf{sh}} (\mathfrak{a} \sqcup_{\mathsf{sh}} \mathfrak{b})} = s_{\mathfrak{a}}$. Assume now $s_{\mathfrak{a}} \neq \mathsf{Nil}$, $s_{\mathfrak{b}} \neq \mathsf{Nil}$ and $s_{\mathfrak{a} \sqcup_{\mathsf{sh}} \mathfrak{b}} \neq \mathsf{Nil}$. First observe that $s_{\mathfrak{a}} \preceq^{\star}_{\bullet} s_{\mathfrak{a} \sqcup_{\mathsf{sh}} \mathfrak{b}}$. This is caused by the fact that $s_{\mathfrak{a} \sqcup_{\mathsf{sh}} \mathfrak{b}} \preceq^{\star}_{\bullet} \max_{\preceq^{\star}_{\bullet}}(s_{\mathfrak{a}}, s_{\mathfrak{b}})$. By the definition of the meet we get $s_{\mathfrak{a} \sqcap_{\mathsf{sh}} (\mathfrak{a} \sqcup_{\mathsf{sh}} \mathfrak{b})} = \min_{\preceq^{\star}_{\bullet}}(s_{\mathfrak{a}}, s_{\mathfrak{a} \sqcup_{\mathsf{sh}} \mathfrak{b}}) = s_{\mathfrak{a}}$.

Again, similar arguments can be used for the two other parts l and e of a strict weighted hexagon.

This completes the proof of Theorem 4.5. □

4.4.2 *Proof of Theorem 4.8*

Let us first restate the theorem:

Theorem (4.8; recalled)**.** *The set of solutions of the given input system* (s, l, e) *is not empty if and only if Algorithm 2 returns* True.

Proof. Algorithm 2 returns False in the following situations:

- if c^0 is marked as positive. This may happen either if there is a path $p \colon c^+ \rightsquigarrow c^0$ or if there is a strict cycle $p \colon c^0 \rightsquigarrow c^0$. In the first case an inequality $c^+ \leqslant a \cdot c^0$ can be deduced from the given input system. This constraint may never be satisfied (since we require c^+ to be always equal to one and c^0 to zero). In the second case an inequality $c^0 < a \cdot c^0$, which corresponds to $0 < a \cdot 0$ is entailed,

- if c^0 or a positive node was marked as negative. In this case there is either a path $p \colon c^+ \rightsquigarrow c^-$ or a strict path $p \colon c^0 \rightsquigarrow c^-$. The first case corresponds to an inequality $c^+ \leqslant a \cdot c^-$ which cannot be satisfied for any positive a (as we require that $\rho(c^+) = 1$ and $\rho(c^-) = -1$ for all valuations ρ). In the second case the system entails an unsatisfiable constraint $c^0 < a \cdot c^-$,

- there is a cycle over positive variables with weight less than 1. Each valuation ρ must assign to a positive variable v a positive value. But a cycle with weight less than 1 represents a constraint $v \leqslant a \cdot v$, where $a < 1$. This may be satisfied only if $\rho(v) \leqslant 0$,

- there is a cycle over negative variables with weight greater than 1. The corresponding constraint $v \leqslant b \cdot v$ for $b > 1$ may be satisfied only if $\rho(v) \geqslant 0$.,

- there is a strict cycle with weight equal to 1. The underlying inequality $v < v$ never holds,

- the computed normal form (s^*, l^*, e^*) contains an unsatisfiable edge, such as $c^+ \leqslant 0 \cdot v$, $c^0 < 0 \cdot v$, $v \leqslant +\infty \cdot c^-$ or $c^0 < a \cdot c^0$.

None of the conditions presented above may be satisfied, hence if Algorithm 2 returns False, then $\gamma_{\mathsf{sh}}((s^*, l^*, e^*)) = \emptyset$.

In all other cases the algorithm returns True. The idea of the proof is to construct a standard weighted hexagon $(\tilde{s}, \tilde{l})$ such that $\gamma_{\mathsf{h}}((\tilde{s}, \tilde{l})) \subseteq \gamma_{\mathsf{sh}}((s^*, l^*, e^*))$ (Fig. 4.4) and show that $\gamma_{\mathsf{h}}((\tilde{s}, \tilde{l})) \neq \emptyset$.

Given the output (s^*, l^*, e^*) of Algorithm 2, we construct a weighted hexagon $(\tilde{s}, \tilde{l})$ by replacing each strict inequality by a slightly more extreme non-strict constraint, e.g. if $s^*(x, y) = (a, \mathsf{True})$ then $\tilde{s}(x, y) = \xi \cdot a$ for some $\xi < 1$. Similarly, if $l^*(x, y) = (b, \mathsf{True})$, then $\tilde{l}(x, y) = \delta \cdot b$ where $\delta > 1$.

We choose the parameters ξ and δ so that the standard Algorithm 1 returns True for (s, l). Let us focus on the smallest constraints stored in

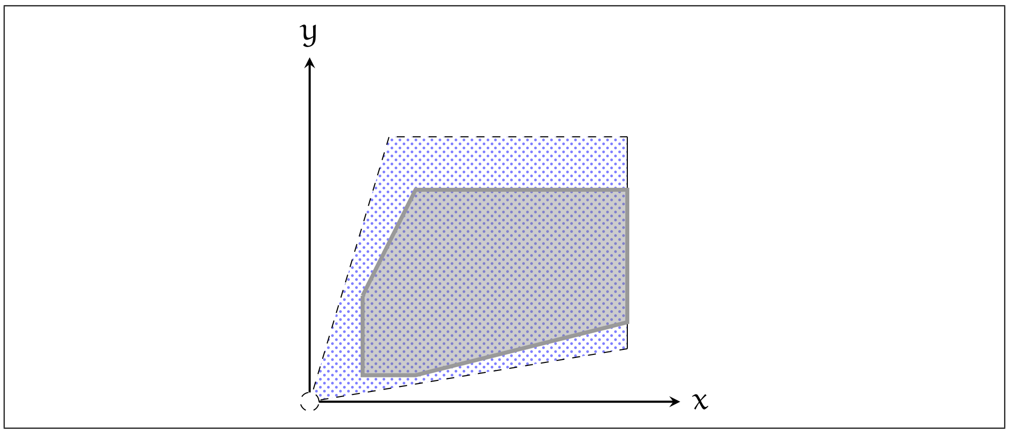

Figure 4.4: Weighted Hexagon enclosed in the given Strict Weighted Hexagon

s^*. All cycles $s^*(v,v)$ that are admissible (pass the test in step 4) must be transformed into admissible cycles in $\tilde{s}$. Let $\mathcal{A}$ denote the set of all strict simple cycles in s^* with weight greater than 1:

$$\mathcal{A} \triangleq \{p \mid p\colon x \overset{\bullet}{\rightsquigarrow} x, \Pi_{\mathcal{G}_{s^*}}(p) = (a, \mathsf{True}) \text{ and } a > 1\}\,.$$

If $\mathcal{A}$ is empty, no cycles must be transformed. Otherwise, let b be the smallest weight of all cycles in $\mathcal{A}$ (note that $\mathcal{A}$ is finite):

$$b \triangleq \inf_{\leqslant}\{a \mid p \in \mathcal{A} \text{ and } \Pi_{\mathcal{G}_{s^*}}(p) = (a, \mathsf{True})\}\,.$$

Clearly, $b > 1$. Each cycle in $\mathcal{A}$ contains at most $n = |\mathcal{Var}|$ strict edges, hence weight of each such cycle after the transformation cannot be smaller than $\xi^n \cdot b$. This justifies the choice of ξ:

$$\xi = \sqrt[n]{\frac{1}{b}}\,.$$

Note that $\xi < 1$. The transformation does not introduce in $\tilde{s}$ any cycle over positive variables with weight less than 1:

- no new edges were added, hence no new cycles were introduced,
- cycles without strict edges were not modified,
- each strict simple cycle over positive variables in s^* is in $\mathcal{A}$. Its weight is not smaller than $\xi^n \cdot b = (\sqrt[n]{\frac{1}{b}})^n \cdot b = 1$,
- each strict cycle consists of a finite number of simple cycles (each of them repeated arbitrary many times), thus its weight is equal to $c = c_1^{n_1} \cdot c_2^{n_2} \ldots c_k^{n_k}$, where each c_i denotes a weight of some simple cycle and $n_i \in \mathbb{N}$. As each $c_i \geqslant 1$, thus $c \geqslant 1$.

It is not enough to replace all strict inequalities from s^* with their stronger non-strict counterparts. A non-strict inequality $x \leqslant \xi \cdot a \cdot y$ would be satisfied by a pair $(0, 0)$, which is not a correct solution for $x < a \cdot y$. If there is no constraint stating that $y \leqslant 0$, we may transform it into a positive variable. This is achieved by adding an edge between c^+ and y.

More formally, let us define the set of variables that are greater than some positive constant:

$$\mathcal{P}_{c^+} \triangleq \{y \mid \tilde{s}(c^+, y) \neq \mathsf{Nil}\}.$$

Let $\mathcal{U}$ be a set of variables that are not negative, do not belong to $\mathcal{P}_{c^+}$ and are strictly greater than some other variable:

$$\mathcal{U} \triangleq \{x \mid \exists_y e^*(y, x) = \mathsf{True}, s^*(x, c^0) = \mathsf{Nil} \text{ and } x \notin \mathcal{P}_{c^+}\}.$$

Let m denote the length of the shortest path in $\tilde{s}$ between any variable from $\mathcal{U}$ and $\mathcal{P}_{c^+}$:

$$m \triangleq \inf_{\leqslant^*}\Big\{\{\tilde{s}(x, w) \cdot \tilde{s}(w, v) \cdot \tilde{s}(v, y) \mid \\ x, w \in \mathcal{U} \text{ and } v, y \in \mathcal{P}_{c^+} \text{ and } (w, y) \in \mathcal{E}\} \cup \{1\}\Big\}.$$

Note that all $\tilde{s}(x, w) \neq 0$, $\tilde{s}(w, v) \neq 0$ and $\tilde{s}(v, y) \neq 0$ (otherwise $x \leqslant 0$, $w \leqslant 0$ or $v \leqslant 0$, what cannot happen), thus $m > 0$. Clearly, $m \leqslant 1$.

We may add now for each $x \in \mathcal{U}$ an edge between c^+ and x as follows:

$$\tilde{s}(c^+, x) \triangleq \frac{1}{m} \text{ and } \tilde{l}(c^+, x) \triangleq +\infty.$$

These edges may introduce new cycles. But each cycle that uses a newly added edge $\tilde{s}(c^+, x)$ must contain also a backward path $p \colon x \rightsquigarrow c^+$. Of course $c^+ \in \mathcal{P}_{c^+}$, hence $\Pi_{\mathcal{G}_{\tilde{s}}}(p) \geqslant m$. Therefore the weight of such cycle may never be smaller than 1:

$$\tilde{s}(c^+, x) \cdot \pi_{\mathcal{G}_s}(p) \geqslant \frac{1}{m} \cdot m = 1.$$

Now s^* is fully transformed into $\tilde{s}$ that contains only more extreme and non-strict constraints. We have also shown that all cycles in $\tilde{s}$ are admissible, that is Algorithm 1 will not return False in step 4.

We shall now briefly discuss other situations, in which Algorithm 1 may return False and conclude that none of them may happen for the function $\tilde{s}$ obtained by the transformation described above:

1. step 3: variables that became positive do not have paths to c^0 and were not marked as negative. The set of negative variables is not modified,
2. step 6: weight of each added edge $\tilde{s}(c^+, x)$ is greater than 0. Similarly weight $\tilde{s}(x, y) = \xi \cdot a$ of each transformed strict edge $s^*(x, y) = (a, \mathsf{True})$ may equal to zero, only if the original weight $a = 0$. Therefore Algorithm 1 applied to $(\tilde{s}, \tilde{l})$ cannot introduce $\tilde{s}^*(c^+, v) = 0$ for any v.

The transformation of the function l^* is very similar. We replace each strict edge $l^*(x, y) = (b, \mathsf{True})$ with $\tilde{l}(x, y) = \delta \cdot b$ for some $\delta > 1$. Let $\mathcal{B}$ denote the set of all strict simple cycles in l^* with weight less than 1:

$$\mathcal{B} \triangleq \{p \mid p \colon x \overset{\bullet}{\rightsquigarrow} x, \Pi_{\mathcal{G}_{l^*}}(p) = (b, \mathsf{True}) \text{ and } b < 1\} .$$

Let d be the largest weight of all cycles in $\mathcal{B}$:

$$d \triangleq \sup_{\leqslant}\{b \mid p \in \mathcal{B} \text{ and } \pi_{\mathcal{G}_{l^*}}(p) = (b, \mathsf{True})\} .$$

As the set $\mathcal{B}$ is finite, it is easy to see that $d < 1$. Each cycle in $\mathcal{B}$ contains at most n strict edges, hence its weight after the transformation is at most $\delta^n \cdot d$. We may choose δ safely as:

$$\delta \triangleq \sqrt[n]{\frac{1}{d}} .$$

Similar reasoning as for s^* may be applied to show that the transformation does not introduce new cycles in $\tilde{l}$ with weight greater than 1.

In the second step of the transformation, we ensure that for each $x, y \in \mathcal{V}ar$, if the input system contained some strict inequality on (x, y) (i.e, $e^*(x, y) = \mathsf{True}$), then no valuation of the output $(\tilde{s}, \tilde{l})$ may admit $\rho(x) = \rho(y) = 0$.

We say that a variable $x \in \mathcal{V}ar$ is *non-negative* in a system $\mathcal{I} = (s, l, e)$ if $\mathcal{I}$ entails $0 \leqslant x$. As $(\tilde{s}, \tilde{l})$ is obtained from a transitive closure (s^*, l^*, e^*) of (s, l, e), it is easy to see that the set of non-negative variables can be characterised as:

$$Q \triangleq \{x \mid \tilde{l}(c^0, x) = +\infty\} .$$

Let $x, y \in \mathcal{V}ar$ be arbitrary variables such that $e^*(x, y) = \mathsf{True}$. If x is non-negative, then y is non-negative as well. It is easy to see that $y \in \mathcal{U}$, hence, while transforming s^*, we have already added $a \leqslant y$ for some positive a (thus, for each solution ρ of $(\tilde{s}, \tilde{l})$, $0 < a \leqslant \rho(y)$).

In the other case x is not non-negative. Let $\mathcal{N}_{c^-}$ denote the set of variables smaller than some negative constant, i.e.:

$$\mathcal{N}_{c^-} \triangleq \{x \mid \tilde{l}(x, c^-) \neq \mathsf{Nil}\} .$$

If x is not non-negative (i.e. $x \notin Q$) and it is not smaller than a negative constant (i.e. $x \notin \mathcal{N}_{c^-}$), then we add to $(\tilde{s}, \tilde{l})$ a constraint $x \leqslant b \cdot c^-$. Let $\mathcal{T}$ denote the set of variables, for which such a constraint must be added:

$$\mathcal{T} \triangleq \{x \mid \exists_y e^*(x, y) = \mathsf{True}, x \notin Q, x \notin \mathcal{N}_{c^-}\} .$$

Let M denote the greatest weight of a path in $\tilde{l}$ between any two variables from $\mathcal{N}_{c^-}$ and $\mathcal{T}$:

$$M \triangleq \sup{}_{\leqslant_\star} \Big\{ \big\{ \tilde{l}(x, w) \cdot \tilde{l}(w, v) \cdot \tilde{l}(v, y) \mid \\ x, w \in \mathcal{N}_{c^-}, v, y \in \mathcal{T}, (w, v) \in \mathcal{E} \big\} \cup \{1\} \Big\}.$$

Note that $M < +\infty$ (as $c^- \in \mathcal{N}_{c^-}$, $M = +\infty$ could only happen if some of the variables in $\mathcal{T}$ was non-negative). Obviously, $m \geqslant 1$. This allows us to add constraints between each variable $x \in \mathcal{T}$ and c^- as follows:

$$\tilde{s}(x, c^-) = 0 \text{ and } \tilde{l}(x, c^-) = \frac{1}{M}.$$

The edges in $\mathcal{G}_{\tilde{l}}$ representing the new constraints may introduce some cycles $c = \langle c^-, ..., x, c^- \rangle$, but each such cycle must contain a backward path $p \colon c^- \rightsquigarrow x$. As $c^- \in \mathcal{N}_{c^-}$ and $\Pi_{\mathcal{G}_{\tilde{l}}}(p) \leqslant M$, we immediately get that the weight of such cycle cannot be greater than 1:

$$\Pi_{\mathcal{G}_{\tilde{l}}}(c) \leqslant M \cdot \frac{1}{M} = 1 .$$

This ends the definition of the transformation of l^*. The proof that Algorithm 1 may never return False for $\tilde{l}$ obtained by this transformation is almost identical as for s^*.

We have shown how to obtain a standard weighted hexagon $(\tilde{s}, \tilde{l})$ for the given strict weighted hexagon (s^*, l^*, e^*) so that each constraint from $(\tilde{s}, \tilde{l})$ is not weaker than the corresponding one from (s^*, l^*, e^*), i.e. each solution of $(\tilde{s}, \tilde{l})$ is also a solution of (s^*, l^*, e^*). We have also proved that Algorithm 1 returns True for $(\tilde{s}, \tilde{l})$. This means that $\gamma_h((\tilde{s}, \tilde{l})) \neq \emptyset$, hence $\gamma_{sh}((s^*, l^*, e^*)) \neq \emptyset$.

This completes the proof of Theorem 4.8. $\square$

Part II

ABSTRACTION OF CONTAINERS

5 OVERVIEW

We have discussed so far abstract interpretation of programs that use only numerical variables. However, a majority of applications use, along with scalar variables, also data structures that can store collections of objects. In many programming languages containers such as dictionaries or arrays are available as predefined building blocks, used by programmers to implement complex systems. This part of the thesis is devoted to abstract analysis of programs that use a particular type of such containers, namely *dictionaries*.

We extend our simple language introduced in Section 1.2.1 so that it contains dictionaries of scalar values. We introduce a finite set of container variables $\mathcal{V}ar_c$ that is disjoint with the set of scalar variables $\mathcal{V}ar$. A *dictionary* is a partial mapping $d\colon \mathbb{K} \rightharpoonup \mathbb{E}$, where $\mathbb{K} \subseteq \mathbb{V}$ denotes the set of valid dictionary keys, while $\mathbb{E} \subseteq \mathbb{V}$ denotes the possible values of dictionary elements. Dictionaries are assumed to be finite. The set of scalar values $\mathbb{V}$ is, unlike in previous chapters, not restricted to numerical values. The set of concrete states $\mathcal{S}tate$ is now defined as

$$\mathcal{S}tate = (\mathcal{V}ar \to \mathbb{V}) \times \big(\mathcal{V}ar_c \to (\mathbb{K} \rightharpoonup \mathbb{E})\big) \cup \{\mathsf{Error}\}\,.$$

The special state Error is used to indicate that a dictionary access error occurred. We add the following types of dictionary simple statements $\mathcal{D}ict\mathcal{S}tmt$ to the language:

1. dictionary creation $T \leftarrow$ **new dict**, where $T \in \mathcal{V}ar_c$,
2. dictionary updates $T[v_1] \leftarrow v_2$ and $T[v_1] \leftarrow c$, where $T \in \mathcal{V}ar_c$, $v_1, v_2 \in \mathcal{V}ar$ and $c \in \mathbb{E}$,
3. a read access $v_2 \leftarrow T[v_1]$ (where $T \in \mathcal{V}ar_c$ and $v_1, v_2 \in \mathcal{V}ar$).

For each type of simple statements we define $t(\mathtt{stmt}, \mathsf{Error}) \triangleq \mathsf{Error}$. The transfer function for simple statements is re-defined in the obvious manner.

Dictionary accesses may occur also as arguments of predicates, i.e. the language contains unary and binary dictionary predicates $\mathcal{D}ict\mathcal{P}red$ $\phi(T[v_1])$ and $\psi(T[v_1], v_2)$, where $T \in \mathcal{V}ar_c$ and $v_1, v_2 \in \mathcal{V}ar$. The transfer function for boolean predicates $p\colon \mathcal{P}red \times \mathcal{S}tate \to \mathcal{B}ool$ (see Section 1.2.1) is extended to $p\colon (\mathcal{Pred} \cup \mathcal{DictPred}) \times \mathcal{State} \to \mathcal{Bool}$ in the straightforward way.

We assume also that we are given a function $\ell\colon (\mathcal{DictStmt} \cup \mathcal{DictPred}) \times \mathcal{State} \to \mathcal{Bool}$ (ℓ stands for "legal") that determines whether the statement (or dictionary predicate) can be executed in this state. The definition of this function depends on the used kind of dictionaries. For instance, when considering arrays (that are a special type of dictionaries discussed later in this section), the function ℓ may return True for the statement $\mathsf{T}[v_1] \leftarrow v_2$ in state (ρ, τ) only when $\rho(v_1)$ is a valid index for the array T. For immutable dictionaries $\ell\big(\mathsf{T}[v_1] \leftarrow v_2, (\rho, \tau)\big) = \mathsf{True}$ only when the statement will not overwrite an existing element in the dictionary, etc. As nothing can be executed in an error state, we assume that $\ell(s, \mathsf{Error}) = \mathsf{False}$ for each $s \in \mathcal{DictStmt} \cup \mathcal{DictPred}$.

Additionally, with each instance of a dictionary, there may be a scalar variable associated. For instance, we represent the length of an array T using a scalar $\mathsf{T}.\mathtt{length}$. Each type of dictionaries can handle its special scalar variables differently. We assume that we are given some function $f\colon \mathcal{DictStmt} \to \mathcal{Stmt}$ that for each dictionary statement determines an associated simple scalar statement which handles the corresponding special scalar variable.

A dictionary statement $s \in \mathcal{DictStmt}$ may be successfully executed only if $\ell\big(s, (\rho, \tau)\big) = \mathsf{True}$ (otherwise it results in an Error state). After the dictionary statement s, we perform the associated scalar instruction $f(s)$. For each $s \in \mathcal{DictStmt}$ we define its semantic $t\colon \mathcal{DictStmt} \times \mathcal{State} \to \mathcal{State}$ as

$$t\big(s, (\rho, \tau)\big) \triangleq \begin{cases} t\big(f(s), \hat{t}\big(s, (\rho, \tau)\big)\big) & \ell\big(s, (\rho, \tau)\big) = \mathsf{True}, \\ \mathsf{Error} & \text{otherwise}, \end{cases}$$

where $\hat{t}$ describes the meaning of each dictionary statement in the case, when it can be executed:

$$\hat{t}\big(\mathsf{T} \leftarrow \textbf{new dict}, (\rho, \tau)\big) \triangleq \big(\rho, \tau[\mathsf{T} \mapsto \epsilon]\big)$$

where ϵ denotes an empty partial function $\mathbb{K} \rightharpoonup \mathbb{E}$. The meaning of the update is defined by:

$$\hat{t}\big(\mathsf{T}[v_1] \leftarrow v_2, (\rho, \tau)\big) \triangleq \big(\rho, \tau\big[\mathsf{T} \mapsto \tau(\mathsf{T})[\rho(v_1) \mapsto \rho(v_2)]\big]\big)\ .$$

Finally, the read access is interpreted as:

$$\hat{t}\big(v_2 \leftarrow \mathsf{T}[v_1], (\rho, \tau)\big) \triangleq \big(\rho[v_2 \mapsto \tau(\mathsf{T})(\rho(v_1))], \tau\big)\ .$$

As we can read only elements that were previously put into the dictionary, $\ell\big(v_2 \leftarrow \mathsf{T}[v_1], (\rho, \tau)\big)$ must check if $\rho(v_1) \in \mathit{Dom}\big(\tau(\mathsf{T})\big)$.

The transfer rules for all existing simple statements are extended in the straightforward way.

If an instruction caused an error, the program should terminate, without executing the next instruction. Similarly, if an illegal dictionary access occurs in a test, the program should stop with an error. We extend the control transfer function $t_c\colon \mathit{Ctrl} \times \mathit{State} \to \mathit{Label}$ (see Section 1.2.1) to $t_c\colon \mathit{Ctrl} \times \mathit{State} \to \mathit{Label} \times \mathit{State}$ in the following way:

1. $t_c(c, \mathsf{Error}) \triangleq (\mathsf{End}, \mathsf{Error})$ for each $c \in \mathit{Ctrl}$,
2. $t_c\big(\mathtt{goto}\,\mathsf{L}, (\rho, \tau)\big) \triangleq \big(\mathsf{L}, (\rho, \tau)\big)$,
3. for each (standard) boolean predicate $\phi \in \mathit{Pred}$,
$$t_c\big(\mathtt{test}\ \ \phi\,\mathsf{L1}\,\mathsf{L2}, (\rho, \tau)\big) \triangleq \begin{cases} \big(\mathsf{L1}, (\rho, \tau)\big) & \text{if } p\big(\phi, (\rho, \tau)\big) = \mathsf{True}, \\ \big(\mathsf{L2}, (\rho, \tau)\big) & \text{if } p\big(\phi, (\rho, \tau)\big) = \mathsf{False}, \end{cases}$$
4. for each dictionary boolean predicate $\psi \in \mathit{DictPred}$:
$$t_c\big(\mathtt{test}\ \ \psi\,\mathsf{L1}\,\mathsf{L2}, (\rho, \tau)\big) \triangleq \begin{cases} (\mathsf{End}, \mathsf{Error}) & \text{if } \ell\big(\psi, (\rho, \tau)\big) = \mathsf{False}, \\ \big(\mathsf{L1}, (\rho, \tau)\big) & \text{if } p\big(\psi, (\rho, \tau)\big) = \mathsf{True}, \\ \big(\mathsf{L2}, (\rho, \tau)\big) & \text{if } p\big(\psi, (\rho, \tau)\big) = \mathsf{False}. \end{cases}$$

Now, as a control statement may change the state, the semantics t_s of a single step of a program execution (see Figure 1.2 in Section 1.2.1) is modified so that

$$\frac{(\mathsf{L}, \mathsf{stmt}, \mathsf{ctrl}) \in \mathsf{Prog} \quad t(\mathsf{stmt}, \rho,) = \rho' \quad t_c(\mathsf{ctrl}, \rho') = (\mathsf{L}', \rho'')}{t_s(\mathsf{Prog}, \mathsf{L}, \rho) = (\mathsf{L}', \rho'')}\ .$$

The rest of the program semantics is defined in the same manner as in Section 1.2.1.

DICTIONARY ANALYSIS We are interested in statically analysing the content of a dictionary. This problem is challenging, since the size of a dictionary is often unbound. In the most trivial approach, a dictionary is treated as $|\mathbb{K}|$ scalar variables, i.e. one abstract value is introduced for each possible key in the dictionary. This approach, known as *expansion* [6], although very precise, can be applied only to dictionaries of bounded (and small) size. Another extreme solution, called *smashing* [6], uses a single abstract value to represent all elements in the dictionary. This technique cannot capture at which key a particular value was stored. Smashing can be applied to arbitrary dictionaries, yet it is very imprecise.

The majority of practically used container analysis techniques lie between the expansion and smashing in terms of both expressiveness and required memory. They divide the unbounded data structure into a bounded number of parts, where each part is represented using a single abstract value. In this approach, the most difficult operation (except of finding a good way to partition the data structure) is the interpretation of an update $\mathsf{T}[\nu_1] \leftarrow \nu_2$. If the modified element is in the partition in a separate singleton group, then after the update, the old value of the corresponding abstract element can be discarded and only the new value must be kept. This case is called a *strong update*. When the modified element is grouped together with other elements, the old value of the abstract element representing this group cannot be forgotten (as it represents not only the modified element but also other elements in the group). Instead, the old value and the freshly added one should be joined. This kind of an update is known as a *weak update*. Strong updates are more precise, thus, whenever possible, one should try to isolate the modified element to perform the strong update.

ARRAYS Many static analysis techniques can deal only with a special type of dictionaries, namely *arrays*. An array is a dictionary whose keys belong to some predefined initial range of natural numbers $\mathbb{N}$. Keys in arrays are commonly called *indices*. Various programming languages differently treat uninitialised array elements (that is elements at indices from the predefined range that have not been written to). They may be implicitly initialised to some default value (as it is done in Java) or may have a random value (as it is done in some cases in C++). We treat uninitialised array elements uniformly with uninitialised elements in arbitrary dictionaries, i.e. an array is a partial mapping $f\colon \{0, \ldots, n\} \rightharpoonup \mathbb{E}$ for some $n \geqslant 0$. The set of array variables will be denoted by $\mathcal{V}ar_a$.

Unlike an arbitrary dictionary, an array has a fixed size (determined when the array is defined). Thus, for each array $\mathsf{T} \in \mathcal{V}ar_a$, we assume that a special variable $\mathsf{T.length}$ is in $\mathcal{V}ar$. Moreover, $\mathsf{T.length}$ must be a natural number (in this case we require that $\mathbb{N} \subseteq \mathbb{V}$).

A new array is created using a statement $\mathsf{T} \leftarrow \textbf{new array}(v_1)$, where $\mathsf{T} \in \mathcal{V}ar_a$. It is a special case of dictionary creation that sets the value of the scalar variable $\mathsf{T.length}$ to $\rho(v_1)$, i.e. we define

$$\hat{t}\big(\mathsf{T} \leftarrow \textbf{new array}, (\rho, \tau)\big) \triangleq \big(\rho, \tau[\mathsf{T} \mapsto \epsilon]\big)$$

and

$$f\big(\mathsf{T} \leftarrow \textbf{new array}(v_1)\big) \triangleq \mathsf{T.length} \leftarrow v_1 \ .$$

The length of an array must be a natural number greater than zero, thus $\ell\big(\mathsf{T} \leftarrow \textbf{new array}(v_1), (\rho, \tau)\big)$ must verify whether $\rho(v_1) \in \mathbb{N} \wedge \rho(v_1) > 0$. As the size of an array is fixed, the value of $\mathsf{T.length}$ is not modified by any other statement, thus $f\big(\mathsf{stmt}, (\rho, \tau)\big) \triangleq \mathtt{skip}$ for all other array statements.

An array cannot be accessed (read or written to) outside of the defined index range. The function ℓ returns True for the statement $\mathsf{T}[v_1] \leftarrow v_2$ if and only if

$$0 \leqslant \rho(v_1) < \rho(\mathsf{T.length}) \wedge \rho(v_1) \in \mathbb{N} \ .$$

A read $v_2 \leftarrow \mathsf{T}[v_1]$ (or a dictionary predicate that includes $\mathsf{T}[v_1]$) is valid in the state (ρ, τ) if $\mathsf{T}[v_1]$ is initialised in this state.

Static analysis of array content is significantly easier than in the case of arbitrary dictionaries. The set of indices is restricted to a subset of natural numbers thus it is totally ordered and it is easier to divide the array elements in a bounded number of groups. In the subsequent sections we survey some existing techniques of array content analysis. Our own technique for analysis of arbitrary dictionaries is presented in Chapter 6.

5.1 PARTITIONING

In the approach of Gopan et al. [32] an unbounded number of concrete array elements (i.e. pairs index, value stored at this index) is partitioned into a bounded number of groups. Each group of concrete array elements is represented by a single abstract array element. Concrete array elements that are assigned to should be isolated in separate groups (so that a strong update can be performed); concrete array elements grouped together should have similar properties, so that the loss of precision induced by the grouping is minimal.

An abstract array element that represents multiple concrete elements is called a *summary* element. An abstract array element is called *non-summary*, when it represents a single concrete array element.

A *partition* P_A of the array A is a set of abstract array elements. Each abstract array element $a \in P_A$ represents a group of concrete array elements. The number of abstract array elements in a partition and the meaning of each such element (i.e. which concrete elements it represents) is determined by numeric relationships among array indices and values of numeric variables. This idea is formalised by the notion of *partition functions*.

PARTITION FUNCTIONS For an array variable $A \in \mathcal{V}ar_a$ and a scalar $v \in \mathcal{V}ar$ we define a *partition function*

$$\pi_{A,v} \colon \{0, \dots, A.\mathtt{length} - 1\} \to \{-1, 0, 1\}\,.$$

This function is evaluated in a concrete state $(\rho, \tau) \in \mathcal{S}tate$ as follows:

$$\pi_{A,v}(i) \triangleq \begin{cases} -1 & \text{if } i < \rho(v), \\ 0 & \text{if } i = \rho(v), \\ 1 & \text{if } i > \rho(v). \end{cases}$$

The set of all partition functions for an array variable $A \in \mathcal{V}ar_a$ is denoted by $\Pi_A = \{\pi_{A,v} \mid v \in \mathcal{V}ar\}$.

A single partition function $\pi_{A,v}$ divides the array A into three groups: elements at indices smaller than the value of v (this group will be denoted by $a_{<v}$), element whose index is equal to the value of v (denoted by $a_{=v}$) and elements at indices greater than the value of v ($a_{>v}$).

The set of all partition functions Π_A yields a partition P_A such that for each partition function $\pi \in \Pi_A$, each abstract array element $a \in P_A$ and every two concrete array elements $A[i]$ and $A[j]$ grouped together in a, $\pi(i) = \pi(j)$. A partition defined in this way consists of at most $3^{|\Pi_A|}$ abstract array elements. For example, if $\mathcal{V}ar = \{v, w\}$ then an abstract array element $a_{<v,>w} \in P_A$ represents the elements of A whose indices are smaller than v and greater than w. The whole partition P_A would be

$$P_A = \{a_{<v,<w}, a_{<v,=w}, a_{<v,>w}, a_{=v,<w}, a_{=v,=w}, \\ a_{=v,>w}, a_{>v,<w}, a_{>v,=w}, a_{>v,>w}\}\,.$$

An abstract array element may represent different sets of concrete array elements at various program points.

If at least one $\pi_{A,v}$ evaluates to zero, then the abstract array element represents at most one concrete element and it is called a *non-summary* element. Otherwise it can potentially represent multiple concrete elements and it is called then a *summary* element.

A map that assigns to each array variable $A \in \mathcal{V}ar_a$ its partition P_A will be denoted by P.

CHOICE OF PARTITION FUNCTIONS Defining a partition function for each array and each scalar in the program leads to partitions containing as many as $3^{|\mathcal{V}ar|}$ abstract array elements. To avoid this problem, for an array $A \in \mathcal{V}ar_a$, we use only partition functions $\pi_{A,\nu}$ parametrised with the scalars $\nu \in \mathcal{V}ar$ that occur in some (read, write or test) access to this array (i.e. there is $A[\nu]$ somewhere in the program text).

ABSTRACTION OF SCALARS Each abstract array element $\mathtt{a} \in P_A$ consists of two scalar values $\mathtt{a.value}$ and $\mathtt{a.index}$ that capture properties of array element indices and values, respectively. The scalar variables $\mathcal{V}ar$ together with the scalars $\mathtt{a.value}$ and $\mathtt{a.index}$ from each abstract array element in partition of any array are modelled within a *summarising numerical abstract domain* D [31] constructed as an extension of some standard numerical abstract domain. Let $d \in \mathcal{D}$ denote an abstract state in this domain.

ABSTRACT PREDICATES The summary abstract elements can be used to express collective properties of concrete elements modelled by such an abstract element (e.g. that values of all these concrete elements are positive). However, relationships among the concrete elements that are abstracted together, are lost. For example, it is not possible to express the property that the values of concrete elements abstracted by one summary element are sorted. To deal with this problem, a set of auxiliary predicates Δ is introduced. A predicate $\delta_A \colon \{0, \dots, \mathtt{A.length} - 1\} \to \{0, 1\}$ in the concrete state $(\rho, \tau) \in \mathcal{S}tate$ holds for an index $k \in \{0, \dots, \mathtt{A.length} - 1\}$ (that is $\delta_A(k) = 1$), if (informally) $A[k]$ obeys the property of interest. The predicates are here implicitly parametrised by the program state.

A concrete predicate δ_A is abstracted in the *3-valued logic* [58] by $\delta_A^\# \colon P_A \to \{0, 1, 1/2\}$. Intuitively, $\delta_A^\#(\mathtt{a}) = 0$ means that δ_A does not hold for any concrete array element abstracted by $\mathtt{a}$, $\delta_A^\#(\mathtt{a}) = 1$ means that δ_A holds for all, while $\delta_A^\#(\mathtt{a}) = 1/2$ indicates that δ_A may hold for some of the concrete array elements. The set of all abstract predicates is denoted by $\Delta^\#$.

```
1: procedure ArrayCopy($A$)
2: $B \leftarrow$ new array($A.length$)
3: $j \leftarrow 0$
4: while $j < A.length$ do
5:     $B[j] \leftarrow A[j]$
6:     $j \leftarrow j + 1$
7: end while
```

Figure 5.1: A routine that copies the content of array A into B

It is a challenging problem to identify the useful predicates Δ. They must be supplied by the user, as there is no mechanism to discover them automatically.

ABSTRACT DOMAIN An abstract memory configuration is a triple $\langle P, d, \Delta^{\#}\rangle$, where P describes partitioning of arrays, $d \in \mathcal{D}$ is an abstract state of the scalars and $\Delta^{\#}$ is an abstraction of predicates. The set of all valid abstract memory configurations can be equipped with the lattice operations and concretisation and abstraction functions so that it forms a proper abstract domain [32].

We do not develop here these operations, instead we demonstrate the analysis on an example, where an array A is copied into B, as shown in Figure 5.1. In this example we assume that the used numerical domain is a relational domain such as the domain of octagons or polyhedra. Assume that the analysis has previously captured that all elements of A range from -5 to 5. The analysis should detect that at the end of the procedure all values stored in B also belong to $[-5, 5]$ and that $B[j] = A[j]$ for each $j \in [0, A.length - 1]$ (note that the arrays A and B have the same size, i.e. $A.length = B.length$). The only scalar used to access the arrays is j, thus the set of partition functions contains $\pi_{A,j}$ and $\pi_{B,j}$. Below we write P_A and P_B to denote partitioning of arrays A and B, respectively. We introduce also an auxiliary predicate $\delta_B \colon \{0, \dots, B.length - 1\} \to \{0, 1\}$ that in any concrete state (ρ, τ) is evaluated as:

$$\delta_B(k) \triangleq [\tau(A)(k) = \tau(B)(k)] \ .$$

At the first loop iteration $j = 0$, hence both $a_{<j} \in P_A$ and $b_{<j} \in P_B$ represent empty sets. The non-summary elements $a_{=j} \in P_A$ and $b_{=j} \in P_B$ represent $A[0]$ and $B[0]$, while $a_{>j} \in P_A$ and $b_{>j} \in P_B$ correspond to the remaining elements. The assignment $B[j] \leftarrow A[j]$ causes $b_j.value = [-5, 5]$ (as we know that each element from A is in this range) and $\delta_B^{\#}(b_{=j}) = 1$. Note that we perform here a strong update.

When j is increased, the current value of $b_{=j}$ is merged with $b_{<j}$, which results in $b_{<j}.value = [-5, 5]$, $0 \leqslant b_{<j}.index < j$ and $\delta^{\#}_{B}(b_{<j}) = 1$. The new value of $b_{=j}$ is extracted from $b_{>j}$. In the k'th loop iteration the summary element $b_{<j}$ represents the already initialised elements (and $b_{<j}.value$ is equal to $[-5, 5]$). The updated element $b_{=j}$ is then merged into $b_{<j}$.

After the loop the summary element $b_{<j}$ represents all elements in the array (as $b_{<j}.index = [0, B.length)$), therefore the analysis may conclude that all elements were initialised to values from the range $[-5, 5]$. Since $\delta^{\#}_{B}(b_{<j}) = 1$, the analysis captured that each element of the array B is equal to the corresponding element in A.

The partition-based approach has some serious drawbacks. It is impossible to express properties of non-contiguous array elements (e.g. to say that all elements at odd indices are equal to 1). The partition functions are chosen using only a syntactic heuristic and the predicates must be supplied by a human. Halbwachs and Péron [38] improved this technique by increasing the level of automation. Marron et al. [50] presented a partition-based approach to represent iterators over collections. Elements in a container are grouped according to their relation to the iterator (before, currently pointed by and after the iterator). This approach does not support dictionaries, since it is not possible to group elements depending on properties of the keys other than their numeric value. The partition-based techniques were reported inefficient in practice [17, 21].

5.2 FUNARRAY

Cousot et al. [17] presented a technique for analysing array content, where each array is automatically divided into a sequence of segments with symbolic bounds. Each segment bound consists of a set of symbolic expressions over the scalar variables that describe the possible indices of elements within this segment. All expressions within one bound are equal (in the concrete domain). The segments are consecutive, without "holes".

The first bound for an array $A \in \mathcal{V}ar_a$ always contains a constant expression 0, while the last one contains $A.length$. In this analysis, it is assumed that all elements at indices $[0, A.length)$ are pre-initialised to some default value. Segment bounds are in increasing order, i.e. all expressions from a bound have values smaller than (or equal to) the expressions from the next bound. If some expressions from two consecutive bounds are equal, then the segment between these bounds is empty.

A segment consists of all array elements whose indices lie between the values of expressions of the lower and upper segment bound. A segment includes its lower bound, but not the upper bound.

An *array segmentation* is a sequence of segments separated by segment bounds. An *abstract segmentation* is a sequence of *abstract segments* separated by *abstract bounds*.

INTUITION Intuitively, an abstract segment is an element of some abstract domain that represents the possible values of array elements within a concrete segment. An abstract bound is a set of symbolic expressions over the scalar variables, restricted to some canonical form. We illustrate this concept by fixing the segment abstraction as the domain of intervals and abstract bounds as expressions of the form $v + c$ where $v \in \mathcal{Var}$ and c is a constant. Let us explain the notation on the following abstract segmentation of an array A:

$$\{0\}\,[1,5]\,\{j\}\,[3,7]\,\{k, l+3\}?\ \top_i\,\{A.\mathtt{length}\}\ .$$

The fragments in curly brackets, i.e. $\{0\}$, $\{j\}$, $\{k, l+3\}?$ and $\{A.\mathtt{length}\}$, denote the abstract bounds, while $[1,5]$, $[3,7]$ and $\top_i$ are abstract segments (in the domain of intervals). In this example, array elements at indices from $[0, j)$ and $[j, k)$ range over $[1,5]$ and $[3,7]$, respectively, while the remaining elements at indices $i \in [k, A.\mathtt{length})$ may have an arbitrary value. Note that the segment bound $\{k, l+3\}$ contains two expressions. This means that $k = l+3$ must hold at a program point described by the considered abstract segmentation (since all expression in a bound are equal). The question mark following this bound indicates that the preceding segment may be empty (when $j = k = l+3$).

Below we discuss in more detail the construction of abstract bounds and abstract segments. We introduce also the FunArray abstract domain and describe how to perform in this model the array updates.

5.2.1 *Building Blocks*

We discuss now all building blocks needed to define the segmentation-based array abstraction.

SCALARS Some abstract domain D (as usual, we write $\mathcal{D}$ to denote the set of abstract elements in D) is used to represent properties of the scalar variables $\mathcal{V}ar$.

EXPRESSIONS The expressions that appear in symbolic bounds are restricted to some canonical form (that is fixed when the analysis is instantiated). The set $\mathcal{E}$ of symbolic expressions must be equipped with equality and inequality tests. For instance for expressions of the form $v + c$, where $v \in \mathcal{V}ar$ and $c \in \mathbb{V}$ one could define a very simple syntactical equality test, whereby $v_1 + c_1 = v_2 + c_2$ if and only if v_1 is syntactically equal to v_2 and $c_1 = c_2$. Similarly, inequality test $v_1 + c_1 \leqslant v_2 + c_2$ may be check if $v_1 = v_2$ and $c_1 \leqslant c_2$. Of course more sophisticated tests can be defined, e.g. taking into account relationships between variables captured by the numeric analysis. We say that $e_1 < e_2$, if $e_1 \leqslant e_2$ and $\neg(e_1 = e_2)$.

Given a concrete valuation of scalars ρ, we denote the value of a symbolic expression $e \in \mathcal{E}$ as $[\![e]\!]_\rho$.

For presentation purposes, we fix the canonical form of expressions as $v + c$, where $v \in \mathcal{V}ar$ and $c \in \mathbb{V}$.

BOUNDS As we have already stated, a bound is a non-empty set of equivalent symbolic expressions. Thus, for a set $\mathcal{E}$ of symbolic expressions, an abstract domain of bounds $B(\mathcal{E})$ is constructed, with $\mathcal{B} \subseteq \mathcal{P}(\mathcal{E})$. The concretisation in the domain of bounds is defined with respect to an abstract state $d \in \mathcal{D}$ of the numerical variables and consists of these valuations $\rho \in \gamma_d(d)$, in which all expressions from the bound b have the same value:

$$\gamma_b(b, d) \triangleq \{\rho \in \gamma_d(d) \mid \forall_{e_1, e_2 \in b} [\![e_1]\!]_\rho = [\![e_2]\!]_\rho\} .$$

For a valuation $\rho \in \gamma_b(b, d)$, $[\![b]\!]_\rho$ is defined as $[\![e]\!]_\rho$ for $e \in b$.

The expression comparison in E is extended to B(E). For $b_1, b_2 \in \mathcal{B}$, we say that $b_1 = b_2$, if there exists $e_1 \in b_1$ and $e_2 \in b_2$ such that $e_1 = e_2$ (and similarly for the inequality test). Note that we rely here on the fact that all expressions in one bound are assumed to be equal.

ABSTRACT SEGMENTS An abstract segment is an element of some abstract domain A that represents pairs of indices and values of array elements in the segment (the concretisation function γ_a is of type $\mathcal{A} \to \mathcal{P}(\mathbb{Z} \times \mathbb{E})$).

One could abstract just the values of array elements, but in this case it would not be possible to model any relationships between the value of an element and its index.

5.2.1.1 *FunArray*

The whole array is abstracted using the array segmentation abstract domain $S(B(\mathcal{E}), A, D)$. The set $\mathcal{S}$ of abstract elements is defined as:

$$\mathcal{S} \triangleq \left\{ (\mathcal{B} \times \mathcal{A} \times \mathcal{B} \times \{\_, ?\}) \times (\mathcal{A} \times \mathcal{B} \times \{\_, ?\})^k \mid k \geqslant 0 \right\} \cup \{\bot_s\} .$$

For each sequence $b_1\, a_1\, b_2[?]\, a_2 \ldots b_m[?]$ the bounds are in an increasing order, that is or each $1 \leqslant i < n$, $b_i < b_{i+1}$. If for some $1 \leqslant j < n$, b_{j+1} is followed by a question mark, then we admit $b_j = b_{j+1}$ (which means that the segment represented by a_j is empty).

CONCRETISATION The concretisation of a segmentation $s \in \mathcal{S}$ of an array A with respect to an abstract state of scalars $d \in \mathcal{D}$, consists of these scalar states $\rho \in \gamma_d(d)$ and arrays $f \colon \mathbb{N} \rightharpoonup \mathbb{E}$ such that

- $\rho \in \gamma_b(b, d)$ for each bound b in the segmentation s,
- the first and last bound are equal to zero and array length respectively, i.e. $[\![b_0]\!]_\rho = 0$ and $[\![b_n]\!]_\rho = \rho(A.\mathtt{length})$,
- ρ preserves the ordering of bounds, i.e. for each $i \in \{0, n-1\}$, $[\![b_i]\!]_\rho <? [\![b_{i+1}]\!]_\rho$ ($<?$ denotes $\leqslant$, when b_{i+1} is followed by ? and $<$ otherwise),
- for each index $j \in [0, \rho(A.\mathtt{length}))$ there exists an abstract segment $b_i\, a_i\, b_{i+1}[?]$ in the segmentation s such that $[\![b_i]\!]_\rho \leqslant j < [\![b_{i+1}]\!]_\rho$ and $(j, f(j)) \in \gamma_a(a_i)$.

DOMAIN OPERATIONS The domain operations are defined directly only for sequences $s_1, s_2 \in \mathcal{S}$ that have the same bounds. Thus, when computing any of the join, meet or widening of $s_1, s_2 \in \mathcal{S}$, one has to perform a *segmentation unification* first.

Given two arbitrary $s_1, s_2 \in \mathcal{S}$, the segmentation unification computes s_1' and s_2' that have the same sequence of bounds and $\gamma_s(s_1) \subseteq \gamma_s(s_1')$ as well as $\gamma_s(s_2) \subseteq \gamma_s(s_2')$. One could obtain s_1' and s_2' by joining all segments in s_1 and s_2, respectively (thus both s_1' and s_2' would contain only one segment). Clearly, such unification would lead to a significant loss of precision. So, intuitively, the unification should join as few segments as possible.

Consider the following array segmentations (with bound expressions in the canonical form as before):

$$\begin{aligned} s_1 &= \{0\}\, a_1^1\, \{i\}\, a_1^2\, \{j\}\, a_1^3\, \{k\}, \\ s_2 &= \{0\}\, a_2^1\, \{j\}\, a_2^2\, \{i\}\, a_2^3\, \{k\}\,. \end{aligned}$$

Given no extra knowledge concerning the relationships between i and j, one can unify s_1 and s_2 either to

$$\begin{aligned} s_1' &= \{0\}\, a_1^1\, \{i\}\, a_1^2 \sqcup_a a_1^3\, \{k\}, \\ s_2' &= \{0\}\, a_2^1 \sqcup_a a_2^2\, \{i\}\, a_2^3\, \{k\} \end{aligned}$$

or to

$$\begin{aligned} s_1'' &= \{0\}\, a_1^1 \sqcup_a a_1^2\, \{j\}\, a_1^3\, \{k\}, \\ s_2'' &= \{0\}\, a_2^1\, \{j\}\, a_2^2 \sqcup_a a_2^3\, \{k\}\,. \end{aligned}$$

Both these unifications have the same (maximal) number of segments, but they are not comparable (none of them is more precise).

The unification algorithm presented for FunArray does not give any guarantee that the computed result will have the maximal possible number of segments, but it is deterministic and always terminates. The algorithm proceeds recursively from left to right, maintaining the invariant that the processed parts of segmentations are already unified.

We present here only the sketch of the algorithm, without going into all technical details. Let $s_1^k = b_1^1\, a_1^1\, b_1^2 \ldots b_1^m$ and $s_2^k = b_2^1\, a_2^1\, b_2^2 \ldots b_2^n$ denote the parts of s_1 and s_2 that remain to be processed before the k-th iteration of the algorithm and let $\widetilde{s_1} = \widetilde{b^1}\, \widetilde{a_1^1}\, \widetilde{b^2} \ldots \widetilde{a_1^l}$ and $\widetilde{s_2} = \widetilde{b^1}\, \widetilde{a_2^1}\, \widetilde{b^2} \ldots \widetilde{a_2^l}$ denote the already unified prefixes.

Roughly, the algorithm computes the set of expressions $\widetilde{b} = b_1^1 \cap b_2^2$ common for b_1^1 and b_2^1. If $\widetilde{b} = \emptyset$, then abstract segments $\widetilde{a_1^l}$ and $\widetilde{a_2^l}$ are replaced with $\widetilde{a_1^l} \sqcup_a a_1^1$ and $\widetilde{a_2^l} \sqcup_a a_2^1$, respectively, and the algorithm proceeds with unifying $b_1^2\, a_1^2 \ldots b_1^m$ with $b_2^2\, a_2^2 \ldots b_2^n$.

Otherwise, i.e. when $\widetilde{b}$ is not empty, $\widetilde{b}\, a_1^1$ and $\widetilde{b}\, a_2^1$ are appended to $\widetilde{s_1}$ and $\widetilde{s_2}$, respectively. In the next step expressions common in $b_1^1 \setminus \widetilde{b}$ and b_2^2 (denoted as $\overline{b_1}$) and those appearing in $b_2^1 \setminus \widetilde{b}$ and b_1^2 (denoted as $\overline{b_2}$) are identified. The algorithm proceeds with unifying $\langle \overline{b_1}?\, a_1^1 \rangle\, b_1^2 \ldots\, b_1^m$ with $\langle \overline{b_2}?\, a_2^1 \rangle\, b_2^2 \ldots\, b_2^n$ (the notation $\langle b\, a \rangle$ means here that the prefix $b\, a$ is present in the segmentation to unify only when $b \neq \emptyset$). The algorithm proceeds until it reaches the last bounds of both input segmentations. The details of the algorithm can be found in the article in which FunArray was introduced [17].

Now, when performing a join, meet or widening, first the segment unification is computed and then the desired operation is performed segment-wise. It is easy to check that the meet and join defined in this way do not form a lattice, as they do not obey the absorption laws. Consider the segmentations $s_1 = \{0\}\,[1,1]\,\{i\}\,[2,2]\,\{\mathtt{T.length}\}$ and $s_2 = \{0\}\,\top_i\{\mathtt{T.length}\}$. As i does not appear in any bound in s_2, it is removed during the unification and s_1 is transformed into $s' = \{0\}\,[1,2]\,\{\mathtt{T.length}\}$. Now it is easy to see that $(s_1 \sqcup_s s_2) \sqcap_s s_1 = s_2 \sqcap_s s_1 = s' \neq s_1$. This point seems to be missed by Cousot et al. [17]. This violates the standard abstract interpretation model introduced in Section 1.2.3. Many properties may be lost (such as existence of the least fixpoint in $\mathcal{S}$, the concretisation and abstraction functions do not form a Galois connection etc.). However the analysis remains sound (any justification of this fact is beyond the scope of this thesis).

ARRAY UPDATES Let us discuss an array update $T[v_1] \leftarrow v_2$. The canonical expression $e \in \mathcal{E}$ for v_1 is just $e \triangleq v_1 + 0$. Let $a_v \in \mathcal{A}$ be a representation of (v_1, v_2) in the domain A (which is computed using a conversion $\kappa_{D \to A}$ from the scalar domain D to the domain A). Let $s = b_1\, a_1\, b_2[?^2] \ldots b_n[?^n]$ be the abstract segmentation of T. Let b_l be the largest bound, for which $b_l \leqslant \{e\}$ (if there is no such bound, we take $b_l \triangleq b_1$). Dually, let b_h be the smallest bound, for which $\{e\} < b_h$ (again, if there is no such bound, we take $b_h \triangleq b_n$). All segments between b_l and b_h are smashed, i.e. we compute $s' = b_1\, a_1 \ldots b_l[?^l] \bigsqcup_{l \leqslant k < h} a_k\, b_h[?^h] \ldots b_n[?^n]$. Now, the smashed segment is split into (up to) three segments resulting in s'' given by

$$b_1\, a_1 \ldots b_l[?^l] \left(\bigsqcup_{l \leqslant k < h} a_k \right) \{e\}[?]\, a_v\, \{e+1\} \left(\bigsqcup_{l \leqslant k < h} a_k \right) b_h[?^h] \ldots b_n[?^n]\,.$$

The updated element is isolated in a separate segment, hence its value can be overwritten (like in a strong update). However, all segments between the bounds b_l and b_h are smashed, causing a loss of precision in the abstraction of other array elements.

If the language was richer, so that some more complex expressions e can be used to access an array (instead of just variables), then it could turn out that e or $e+1$ cannot be put in the chosen canonical form. In this case the modified element cannot be isolated into a separate segment and a weak update on the smashed segment between the bounds b_l and b_h is performed.

The FunArray technique has been implemented in Clousot [23], a tool for static analysis of programs in the .NET intermediate language and was reported efficient and precise in practice. One of its main advantages is the high level of customisation, i.e. one can choose the domains for abstraction of scalars, bound expressions and array elements. The segmentation-based approach has also its disadvantages. It is impossible to represent properties of non-contiguous array elements (as a segment always describes a coherent fragment of the array). It also strongly relies on the linear ordering of indices, hence it is not applicable to other data structures, such as dictionaries with arbitrary keys.

5.3 other approaches

We mention now some other container analysis techniques. The majority of them is defined only in terms of arrays and cannot be used to model dictionaries.

Dillig et al. [21] introduced *fluid updates* that relax the dichotomy between strong and weak updates. This is a heap analysis technique, in which each reference between locations is qualified with paired constraints over- and under-approximating the set of concrete states, in which this reference is established. The constraints can be expressed only in theory of linear integer arithmetic. This approach is easily applicable to arrays (as the indices are natural numbers). Recently it was applied also to other containers [22]. For dictionaries keyed by arbitrary values, the keys are first converted to integers using an invertible uninterpreted function *pos*. The only assumption made about *pos* is that no two different keys may be mapped to the same value. After the conversion, the standard fluid update technique is used. This approach has a major drawback. It is impossible to express any non-trivial properties of the keys. Let us assume that we want to model a dictionary keyed by strings. As the only axiom about the *pos* function states that two different keys cannot be mapped to the same index, we cannot express any partial knowledge about the keys, such as "a key starts with a given prefix". Another problem with this technique is that it is not much customisable. The precision and cost of the analysis strongly depend on the theory of linear integer arithmetic and uninterpreted functions used to express the bracketing constraints, and it cannot be adjusted to the requirements of a specific analysis. There is an ongoing research to adapt the fluid updates technique to the abstract interpretation framework [29].

Flanagan and Quadeer [24] use predicate abstraction [33] to infer universally quantified loop invariants that describe properties of arrays. The necessary predicates are generated using syntactic heuristics. Ramalingam et al. [56] apply predicate abstraction to check if a client program conforms to the constraints for correct usage of a collection, e.g. that a program does not modify the collection while iterating over it. Blanc et al. [5] use a similar technique to verify the usage of STL containers in C++ programs. They check if all preconditions of STL methods are fulfilled. Neither of these two techniques can be used to reason about container content.

Seghir et al. [60] propose a counterexample-guided abstraction technique for verifying quantified assertions over arrays. Henzinger et al. [39] present a method for an automatic inference of quantified invariants for multidimensional arrays. The powerful, yet expensive technique of Gulwani et al. [36] uses user-provided templates for array invariants. Kovács and Voronkov [46] use a theorem prover to generate loop invariants in programs using arrays.

Each of these analyses is useful in some context, but they are not suitable for analysis of dictionary content. In the next chapter we propose our own technique that can be applied to analyse both array and dictionary content.

6 GENERIC ABSTRACTION OF CONTAINERS

In this chapter we present our abstract domain which can be used to uniformly represent arbitrary dictionaries [26]. This technique does not rely on numerical properties of scalar variables, thus it can be instantiated using various existing abstract domains, including non-numerical ones (such as domains for analysis of properties of string variables). Our approach is powerful enough to model relationships between scalar variables and dictionary keys and between scalars and dictionary elements.

The analysis is fully automatic. The container is partitioned according to properties of the keys, captured by the underlying key abstraction. The precision and cost of the analysis are customisable and depend on the choice of the abstractions of keys, dictionary elements and scalar variables.

We show examples in which the technique is used to reason about arrays as well as string-keyed dictionaries.

6.1 GENERIC DICTIONARY ABSTRACTION

In our approach a dictionary $T \in \mathcal{V}ar_c$ is modelled as a set of *abstract segments*. Each abstract segment represents some set of (concrete) keys and corresponding dictionary elements. Let K and V be two scalar abstract domains (with carriers $\langle \mathcal{K}, \sqcup_k, \sqcap_k \rangle$ and $\langle \mathcal{V}, \sqcup_v, \sqcap_v \rangle$, respectively). An abstract segment is a pair $(k, v) \in \mathcal{K} \times \mathcal{V}$, where k models a set of concrete keys (together with their relations to scalars) and is called an *abstract key*, while v abstracts dictionary elements (and their relations to scalars) and is called an *abstract value*. A (concrete) dictionary is abstracted as a finite set of abstract segments.

Let us start with some auxiliary terminology. In a complete lattice $\langle \mathcal{A}, \sqcup_a, \sqcap_a, \top_a, \bot_a \rangle$, we say that $a \in \mathcal{A}$ is *empty*, if $a = \bot_a$. We say that $a \in \mathcal{A}$ *overlaps* with $b \in \mathcal{A}$, when $a \sqcap_a b \neq \bot_a$.

We define now a lattice $\langle \mathcal{D}, \sqcup_d, \sqcap_d \rangle$, where $\mathcal{D} \subseteq \mathcal{P}_{fin}(\mathcal{K} \times \mathcal{V})$ and each $d \in \mathcal{D}$ fulfils the following additional conditions:

1. for each $(k_1, v_1) \in d$ and $(k_2, v_2) \in d$ either $(k_1, v_1) = (k_2, v_2)$ or $k_1 \sqcap_k k_2 = \bot_k$,

2. for each $(k, v) \in \mathcal{d}$, $k \neq \bot_k$ and $v \neq \bot_v$.

The first condition states that every two abstract segments represent disjoint sets of concrete elements. The second condition forbids abstract segments with empty abstract keys or abstract values. An abstract segment $(\bot_k, v)$ would represent an empty fragment of the dictionary, while $(k, \bot_v)$ would model a set of elements that could not have been initialised to any value. Such abstract segments are superfluous in our representation.

For each (concrete) dictionary $d \colon \mathbb{K} \rightharpoonup \mathbb{E}$ represented by an abstract dictionary $\mathcal{d} \in \mathcal{D}$ and for each (concrete) key $n \in \mathit{Dom}(d)$, we will ensure that there exists an abstract segment $(k, v) \in \mathcal{d}$ such that n and $d(n)$ are abstracted by k and v, respectively. This will be formalised later in this chapter.

MEET The greatest lower bound $\mathcal{a} \sqcap_d \mathcal{b}$ of $\mathcal{a}, \mathcal{b} \in \mathcal{D}$ consists of abstract segments obtained as a point-wise meet of some abstract segments from $\mathcal{a}$ and $\mathcal{b}$:

$$\mathcal{a} \sqcap_d \mathcal{b} \triangleq \{(k_{\mathcal{a}} \sqcap_k k_{\mathcal{b}}, v_{\mathcal{a}} \sqcap_v v_{\mathcal{b}}) \mid (k_{\mathcal{a}}, v_{\mathcal{a}}) \in \mathcal{a}, (k_{\mathcal{b}}, v_{\mathcal{b}}) \in \mathcal{b}, \\ k_{\mathcal{a}} \sqcap_k k_{\mathcal{b}} \neq \bot_k, v_{\mathcal{a}} \sqcap_v v_{\mathcal{b}} \neq \bot_v\} \,. \quad (6.1)$$

Lemma 6.2. *The meet operator is well defined, i.e.* $\mathcal{a} \sqcap_d \mathcal{b} \in \mathcal{D}$.

Proof. The property that each two abstract segments are disjoint follows from the well-formedness of $\mathcal{a}$ and $\mathcal{b}$ and from the properties of $\sqcap_k$. The second well-formedness property is trivial. Details can be found in Section 6.5.1. □

JOIN The join $\mathcal{a} \sqcup_d \mathcal{b}$ of $\mathcal{a}, \mathcal{b} \in \mathcal{D}$ should represent all concrete dictionaries abstracted by $\mathcal{a}$ or $\mathcal{b}$, thus one could imagine defining the join as a union $\mathcal{a} \cup \mathcal{b}$. However, in presence of our well-formedness conditions, such simple definition is not valid, as an abstract key $k_{\mathcal{a}}$ in a segment $(k_{\mathcal{a}}, v_{\mathcal{a}}) \in \mathcal{a}$ may overlap with a key $k_{\mathcal{b}}$ in some segment $(k_{\mathcal{b}}, v_{\mathcal{b}}) \in \mathcal{b}$.

We show now how to deal with this problem, i.e. how to transform an arbitrary set of abstract keys into a set in which no two keys overlap. The idea is to identify groups of overlapping keys and replace each group with its least upper bound.

Let $\langle \mathcal{A}, \sqcap_a, \sqcup_a, \top_a, \bot_a \rangle$ be a complete lattice and let S be a finite subset of $\mathcal{A}$.

Definition 6.3. We say that a (finite) family of sets $\mathcal{X} = \{X_1, X_2, \ldots, X_k\}$, where each $X_i \subseteq S$, is a *disjoint partition* of S, if and only if:

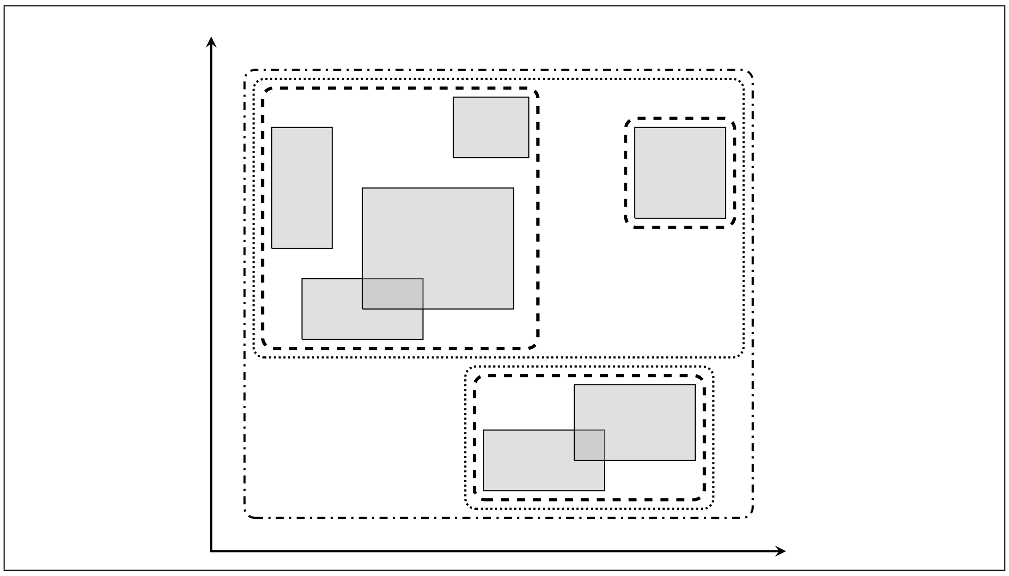

Figure 6.1: A set of elements of the domain of intervals (grey rectangles) and all its disjoint partitions (the least disjoint partition is marked with a thick dashed line)

- $\mathcal{X}$ is a partition of S (i.e. $S = \bigcup \mathcal{X}$ and $X_i \cap X_j = \emptyset$ for $i \neq j$),
- each $X \in \mathcal{X}$ is non-empty,
- for every $X_i, X_j \in \mathcal{X}$, where $i \neq j$, it holds that $(\bigsqcup_a X_i) \sqcap_a (\bigsqcup_a X_j) = \bot_a$.

A disjoint partition of S always exists — if $S = \emptyset$, then $\mathcal{X} \triangleq \emptyset$, otherwise one can take $\mathcal{X} \triangleq \{S\}$. It may happen that S has multiple disjoint partitions. We are particularly interested in a partition $\mathcal{X}$ of S that does not perform any unnecessary grouping:

Definition 6.4. We say that a disjoint partition $\mathcal{C}$ of S is *least*, if for any disjoint partition $\mathcal{X}$ of S it holds that $\forall_{C \in \mathcal{C}} \exists_{X \in \mathcal{X}} C \subseteq X$.

Intuitively, the least disjoint partition groups (i.e. puts into the same X_i) only these elements of S, which must be grouped together in each disjoint partition of S.

Lemma 6.5. *The least disjoint partition of* S *exists and is uniquely defined.*

Proof. We show how to construct the least disjoint partition starting from a (not necessarily disjoint) partition, in which each element of S is in a separate group. Details can be found in Section 6.5.2. $\square$

A sample set of elements of the domain of intervals (in a two-dimensional case) and all its disjoint partitions are shown in Figure 6.1. It is easy to see that no two elements from one group in the least disjoint partition (marked with a thick dashed line) can belong to different groups in any disjoint partition.

We use the concept of the least disjoint partition to transform arbitrary $c \in \mathcal{P}_{fin}(\mathcal{K} \times \mathcal{V})$ into an abstract dictionary $d \in \mathcal{D}$. Let S denote the set of abstract keys of "non-empty" abstract segments $(k, v) \in c$:

$$S \triangleq \{k \mid (k, v) \in c, k \neq \bot_k, v \neq \bot_v\} .$$

Let $\mathcal{C}$ denote the least disjoint partition of S. We define a *disjoint normalisation* function $\mathrm{dNorm}\colon \mathcal{P}_{fin}(\mathcal{K} \times \mathcal{V}) \to \mathcal{D}$ as:

$$\mathrm{dNorm}(c) \triangleq \{(k_1 \sqcup_k k_2 \cdots \sqcup_k k_m, v_1 \sqcup_v v_2 \cdots \sqcup_v v_m) \mid \{k_1, \ldots, k_m\} \in \mathcal{C}, (k_j, v_j) \in c, j \in \{1, \ldots, m\}\} .$$

This normalisation can be computed in $O(|c|^3)$ time, including the computation of the least disjoint partition $\mathcal{C}$. Now the join $a \sqcup_d b$ can be defined just as the normalised union $a \cup b$:

$$a \sqcup_d b \triangleq \mathrm{dNorm}(a \cup b) . \tag{6.6}$$

Theorem 6.7. *Set $\mathcal{D}$ together with the meet and join operators given by* (6.1) *and* (6.6) *forms a lattice.*

Proof. Standard examination of the commutativity, associativity and absorption properties, see Section 6.5.3. $\square$

BOTTOM AND TOP It is now easy to check that the bottom $\bot_d$ is an empty set, while the top $\top_d$ is equal to $\{(\top_k, \top_v)\}$.

WIDENING When the domains K and V are finite, then the widening is just equal to the join. Otherwise we define ∇_d as:

$$\bot_d \nabla_d a \triangleq a \qquad a \nabla_d \bot_d \triangleq a \qquad a \nabla_d b \triangleq \mathrm{dNorm}(a \tilde{\nabla} b),$$

where $a \tilde{\nabla} b$ consists of three types of abstract segments:

1. for two abstract segments $(k, v) \in a$ and $(l, w) \in b$ with overlapping keys, $a \tilde{\nabla} b$ contains their point-wise widening $(k \nabla_k l, v \nabla_v w)$,

2. each abstract segment $(k, v) \in a$ such that k is disjoint with all keys from all segments $(l, w) \in b$ is put into $a \tilde{\nabla} b$,

3. each abstract segment $(l, w) \in \mathcal{b}$ such that l is disjoint with all keys from segments $(k, v) \in \mathcal{a}$ is widened to $(\top_k, w)$:

$$\begin{aligned} \mathcal{a} \tilde{\nabla} \mathcal{b} \triangleq \; & \left\{(k \nabla_k l, v \nabla_v w) \mid (k, v) \in \mathcal{a}, (l, w) \in \mathcal{b}, k \sqcap_k l \neq \bot_k\right\} \\ & \cup \left\{(k, v) \mid (k, v) \in \mathcal{a}, \forall_{(l,w) \in \mathcal{b}}\, k \sqcap_k l = \bot_k\right\} \\ & \cup \left\{(\top_k, w) \mid (l, w) \in \mathcal{b}, \forall_{(k,v) \in \mathcal{a}}\, k \sqcap_k l = \bot_k\right\} . \end{aligned} \tag{6.8}$$

Theorem 6.9. *The operator ∇_d defined above is a widening operator.*

Proof. See Section 6.5.4. □

In the definition above, the last case, where a segment is widened to $(\top_k, w)$, causes the most significant loss of precision (as the dNorm function will smash the whole dictionary into a single abstract segment). It is possible to define a more precise (yet more complicated) widening.

Instead of widening (l, w) to $(\top_k, w)$ in the third part of (6.8), one can choose arbitrary abstract segment $(k, v) \in \mathcal{a}$ and widen (l, w) to $(k \nabla_k l, v \nabla_v w)$. However, in general there is no best (most precise) choice of (k, v). Thus, we propose here only some heuristic. Let $W_{(k,v)}$ denote the set of abstract segments from $\mathcal{a} \cup \mathcal{b}$, whose keys overlap with $k \nabla_k l$:

$$W_{(k,v)} \triangleq \left\{(m, u) \mid (m, u) \in \mathcal{a} \cup \mathcal{b}, m \sqcap_k (k \nabla_k l) \neq \bot_k\right\} .$$

Now we choose $(k, v) \in \mathcal{a}$ such that $W_{(k,v)}$ has the smallest cardinality among all $(k', v') \in \mathcal{a}$ (if there are multiple such elements, any of them may be chosen).

Note that we define the widening whenever K or V is infinite (not necessarily of infinite height). This is necessary, as even for K and V of finite height there may exist strictly increasing infinite sequences in D: let K and V be defined so that their carriers are equal to $\mathbb{N} \cup \{\bot, \top\}$ and with an order where the natural numbers are not comparable. The sequence $d_0, d_1, \ldots$ in $\mathcal{D}$ given by $d_i \triangleq \{(j, 0) \mid j \leqslant i\}$ is strictly increasing.

Note also that the lattice $\langle \mathcal{D}, \sqcap_d, \sqcup_d \rangle$ is in general not complete, but as discussed in Section 1.2.8, the abstract interpretation framework can still be applied.

6.1.1 *Variable Introduction and Elimination*

For an abstract dictionary $\mathcal{d} \in \mathcal{D}(\mathcal{K}, \mathcal{V})$ the elimination $\downarrow_x$ of a variable $x \in \mathcal{V}ar$ (see Section 1.2.5) is defined as:

$$\mathcal{d}\downarrow_x \triangleq \mathrm{dNorm}\big(\{(k\downarrow_x, v\downarrow_x) \mid (k, v) \in \mathcal{d}\}\big) .$$

Similarly, variable introduction $\uparrow_x$ is defined as:

$$\mathcal{d}\uparrow_x \triangleq \mathrm{dNorm}\left\{(k\uparrow_x, v\uparrow_x) \mid (k,v) \in \mathcal{d}\right\} .$$

Additionally, when the introduction $\uparrow_x$ in the domain K is exact, then the application of the disjoint normalisation is superfluous.

The variable introduction and elimination define a forget operator $\updownarrow_x$ such that $\mathcal{d}\updownarrow_x \triangleq (\mathcal{d}\downarrow_x)\uparrow_x$, as explained in Section 1.2.5.

6.2 THE DOMAIN

We utilise now the lattice defined in the previous section to define an abstract domain.

Let us start with the container part of the concrete state. The content of each dictionary will be over-approximated using an element of the lattice $\langle \mathcal{D}, \sqcup_d, \sqcap_d \rangle$ defined above. Roughly, given an abstract segment (k, v), k over-approximates a set of keys of concrete elements and v over-approximates the set of their possible values. If a concrete key is not abstracted at all, then the corresponding element cannot be initialised.

The abstract keys are modelled within an abstract domain K over the set $\mathcal{Var} \cup \{\mathcal{k}\}$, where $\mathcal{k}$ is an artificial *key variable* (similar to the index variable used by Dillig et al. [21]) used to represent the value of a key. Similarly, the abstract values are represented using an abstract domain V over the set $\mathcal{Var} \cup \{t\}$, where t is a *value-tracking variable* that represents the value of a dictionary element. In this approach it is possible to express relations between scalars and keys as well as scalars and dictionary elements.

The structure $\mathcal{D}(\mathcal{K}, \mathcal{V})$ can be used to over-approximate the content of a dictionary, but it does not give any information about which elements must be initialised. We resolve this problem by associating with each dictionary also an element $i \in \mathcal{D}(\mathcal{K}, \mathcal{Bool})$ that is used to over-approximate the set of uninitialised dictionary elements. An abstract segment $(k, \mathsf{True}) \in i$ means that the dictionary elements at keys abstracted by k may be uninitialised. On the other hand, if some key is not abstracted by any segment in i, then the element at this key must be initialised. As False is the bottom in the lattice of booleans, segments (l, False) are automatically removed.

The scalar part of the state is abstracted in some abstract domain A over the set of variables $\mathcal{Var}$. We require also conversion functions $\kappa_{A\to K}$ and $\kappa_{K\to A}$ between the domains A and K as well as $\kappa_{A\to V}$ and

$\kappa_{V \to A}$ between A and V. When $A = K$ (or $A = V$), then the respective conversions are just introduction and elimination of the special variables $\mathcal{k}$ and ι (see Section 1.2.8).

Now we may define the domain $C = \langle \mathcal{C}, \sqcup_c, \sqcap_c, \top_c, \bot_c, \gamma_c, \alpha_c, \delta_c, \pi_c, \nabla_c \rangle$. The set of abstract states $\mathcal{C}$ is given by

$$\mathcal{C} \triangleq \mathcal{A} \times \left(\mathcal{V}ar_c \to \mathcal{D}(\mathcal{K}, \mathcal{V})\right) \times \left(\mathcal{V}ar_c \to \mathcal{D}(\mathcal{K}, \mathcal{B}ool)\right) \cup \{\mathsf{AError}\} \, .$$

A special abstract error state AError is put into $\mathcal{C}$; intuitively, it is used to indicate that a dictionary access error may have occurred.

DOMAIN OPERATIONS The special abstract error state is treated as the greatest element in $\mathcal{C}$, thus for $c \in \mathcal{C}$ we define $c \sqcap_c \mathsf{AError} \triangleq c$ and $c \sqcup_c \mathsf{AError} \triangleq \mathsf{AError}$. In other cases the meet and join operations are given point-wise (with a little abuse of notation we lift the meet and join $\sqcap_d$ and $\sqcup_d$ in $\mathcal{D}$ to $\mathcal{V}ar_c \to \mathcal{D}$):

$$(a_1, d_1, i_1) \sqcap_c (a_2, d_2, i_2) \triangleq (a_1 \sqcap_a a_2, d_1 \sqcap_d d_2, i_1 \sqcap_d i_2)$$

and

$$(a_1, d_1, i_1) \sqcup_c (a_2, d_2, i_2) \triangleq (a_1 \sqcup_a a_2, d_1 \sqcup_d d_2, i_1 \sqcup_d i_2) \, .$$

The widening ∇_c can also be defined point-wise. However, we propose a slightly different approach. In our analysis, the only types of statements that modify a dictionary are $T[v_1] \leftarrow v_2$ and $T[v_1] \leftarrow c$, where $T \in \mathcal{V}ar_c$, $v_1, v_2 \in \mathcal{V}ar$ and $c \in \mathbb{V}$. Thus, the dictionary modifications get stable when the abstract state $a \in \mathcal{A}$ of scalar variables stabilises. Following this observation, we define the widening ∇_c as :

$$(a_1, d_1, i_1) \nabla_c (a_2, d_2, i_2) \triangleq \begin{cases} (a_1 \nabla_a a_2, d_1 \sqcup_d d_2, i_1 \sqcup_d i_2) & a_1 \nabla_a a_2 \neq a_1 \\ (a_1 \nabla_a a_2, d_1 \nabla_d d_2, i_1 \nabla_d i_2) & \text{otherwise.} \end{cases}$$

CONCRETISATION The abstract error state AError represents the set of all possible concrete states, i.e. $\gamma_c(\mathsf{AError}) \triangleq \mathcal{S}tate$. In particular $\mathsf{Error} \in \gamma_c(\mathsf{AError})$.

In a non-error abstract state (a, d, i), the concretisation of scalars is defined using the concretisation γ_a in the domain A, as defined in Section 1.2.3. Let us consider a concrete valuation $\rho \colon \mathcal{V}ar \to \mathbb{V}$ of scalar variables, a dictionary $T \in \mathcal{V}ar_c$. and a concrete key $n \in \mathbb{K}$. If there is an abstract segment $(k, \mathsf{True}) \in i(T)$ such that n is abstracted by k, then $T[n]$ may be uninitialised. If there is an abstract segment $(l, w) \in d(T)$ such that n is abstracted by l, then $T[n]$ may have some value abstracted by w. Eventually, if there is neither $(k, \mathsf{True}) \in i(T)$ nor $(l, w) \in d(T)$ such that n is abstracted by k or l, then $T[n]$ can be neither initialised nor uninitialised, which means that for the chosen valuation $\rho\colon \mathcal{V}ar \to \mathbb{V}$ of scalars, there is no valuation $\tau\colon \mathcal{V}ar_c \to (\mathbb{K} \rightharpoonup \mathbb{E})$ of dictionaries.

Following these observations, we define a predicate $\mathfrak{I}(\rho, T, n, i)$ for $\rho\colon \mathcal{V}ar \to \mathbb{V}$, $T \in \mathcal{V}ar_c$, $n \in \mathbb{K}$ and the initialisation part of the abstract state i that holds if and only if $T[n]$ may be uninitialised:

$$\mathfrak{I}(\rho, T, n, i) \triangleq \mathsf{True} \quad \text{iff} \quad \exists_{(k,\mathsf{True}) \in i(T)} \exists_{\sigma \in \gamma_k(k)} \sigma_{|\mathcal{V}ar} = \rho, \sigma(\mathcal{k}) = n\,. \tag{6.10}$$

Similarly, we define a predicate $\mathfrak{V}(\rho, T, n, m, d)$ for $\rho\colon \mathcal{V}ar \to \mathbb{V}$, $T \in \mathcal{V}ar_c$, $n \in \mathbb{K}$, $m \in \mathbb{E}$ that holds when $T[n]$ may be equal to m:

$$\begin{aligned}\mathfrak{V}(\rho, T, n, m, d) \triangleq \mathsf{True} \quad \text{iff} \quad & \exists_{(k,v) \in d(T)} \exists \sigma_k \in \gamma_k(k) \exists \sigma_v \in \gamma_v(v) \\ & \sigma_{k|\mathcal{V}ar} = \sigma_{v|\mathcal{V}ar} = \rho, \sigma_k(\mathcal{k}) = n, \sigma_v(t) = m\,.\end{aligned} \tag{6.11}$$

This allows us to define the concretisation γ_c as:

$$\begin{aligned}\gamma_c\big((a, d, i)\big) \triangleq \Big\{(\rho,\tau) \mid \rho \in \gamma_a(a), \forall_{T \in \mathcal{V}ar_c} \forall_{n \in \mathbb{K}} \\ \big(n \notin \mathit{Dom}(\tau(T)) \text{ and } \mathfrak{I}(\rho, T, n, i)\big) \text{ or} \\ \big(n \in \mathit{Dom}(\tau(T)) \text{ and } \mathfrak{V}(\rho, T, n, \tau(T)(n), d)\big)\Big\}\,.\end{aligned}$$

We may now state the following lemmas about the meet and join in C:

Lemma 6.12. *If the domains A, K and V have exact meets, i.e.*

- $\gamma_a(a_1) \cap \gamma_a(a_2) = \gamma_a(a_1 \sqcap_a a_2)$,
- $\gamma_k(k_1) \cap \gamma_k(k_2) = \gamma_k(k_1 \sqcap_k k_2)$ *and*
- $\gamma_v(v_1) \cap \gamma_v(v_2) = \gamma_v(v_1 \sqcap_v v_2)$

then the meet $\sqcap_c$ in the domain C is also exact:

$$\gamma_c\big((a_1, d_1, i_1)\big) \cap \gamma_c\big((a_2, d_2, i_2)\big) = \gamma_c\big((a_1, d_1, i_1) \sqcap_c (a_2, d_2, i_2)\big)\,.$$

Proof. For arbitrary dictionary $T \in \mathcal{V}ar_c$ and arbitrary concrete key $n \in \mathbb{K}$ we show both inclusions of the proved equality. We distinguish the cases when $T[n]$ is initialised and uninitialised. Details can be found in Section 6.5.5. □

The same property holds for over-approximating meets, i.e. one can replace all equalities with set inclusion. A similar lemma can be also formulated about the join (we omit the variant for exact joins as they are very rare in practice):

Lemma 6.13. *If the domains* A, K *and* V *have over-approximating joins, i.e.*

- $\gamma_a(a_1) \cup \gamma_a(a_2) \subseteq \gamma_a(a_1 \sqcup_a a_2)$,
- $\gamma_k(k_1) \cup \gamma_k(k_2) \subseteq \gamma_k(k_1 \sqcup_k k_2)$ *and*
- $\gamma_v(v_1) \cup \gamma_v(v_2) \subseteq \gamma_v(v_1 \sqcup_v v_2)$

then the join $\sqcup_c$ *in the domain* C *is also over-approximating:*

$$\gamma_c\big((a_1, d_1, i_1)\big) \cup \gamma_c\big((a_2, d_2, i_2)\big) \subseteq \gamma_c\big((a_1, d_1, i_1) \sqcup_c (a_2, d_2, i_2)\big) .$$

Proof. Direct examination of the inclusion, see Section 6.5.6. □

VARIABLE INTRODUCTION AND ELIMINATION The introduction and elimination of scalar variables are defined point-wise, i.e.

$$(a, d, i)\downarrow_x \triangleq (a\downarrow_x, \lambda T.d(T)\downarrow_x, \lambda T.i(T)\downarrow_x) \quad \text{and}$$
$$(a, d, i)\uparrow_x \triangleq (a\uparrow_x, \lambda T.d(T)\uparrow_x, \lambda T.i(T)\uparrow_x) .$$

Recall that the variable elimination and introduction define the forget operator, i.e. $(a, d, i)\updownarrow_x \triangleq \big((a, d, i)\downarrow_x\big)\uparrow_x$.

6.2.1 *Abstract Transfer Function*

The error state AError is "preserved" by the abstract transfer function, i.e. for each simple statement $s \in \mathit{Stmt}$, we define $\delta_c(s, \mathrm{AError}) \triangleq \mathrm{AError}$. Similarly, for each predicate $p \in \mathit{Pred}$, $\pi_c(p, \mathrm{AError}) \triangleq (\mathrm{AError}, \mathrm{AError})$. The abstract control function $\delta^c_c\colon \mathit{Ctrl} \times \mathcal{C} \to (\mathit{Label} \rightharpoonup \mathcal{C})$ introduced in Section 1.2.3 is also extended so that after a possible error the execution may jump directly to the End label:

$$\delta^c_c(\texttt{test P L1 L2}, \mathrm{AError}) \triangleq [\mathrm{L1} \mapsto \mathrm{AError}, \mathrm{L2} \mapsto \mathrm{AError}, \mathrm{End} \mapsto \mathrm{AError}]$$

and

$$\delta^c_c(\texttt{goto L}, \mathrm{AError}) \triangleq [\mathrm{L} \mapsto \mathrm{AError}, \mathrm{End} \mapsto \mathrm{AError}] .$$

In all definitions of the abstract transfer function presented below, we assume that the input abstract state is (a, d, i). We illustrate the definitions on simple examples, where all A, K and V are chosen as the domain of intervals, with the set of scalar values $\mathbb{V}$, keys $\mathbb{K}$ and dictionary elements $\mathbb{E}$ chosen as integers $\mathbb{Z}$. For clarity, in each abstract segment we show only the values of the key and value-tracking variables $\mathcal{k}$ and $\mathcal{t}$.

$$\frac{\begin{array}{c} d' \triangleq \lambda T.\mathrm{dNorm}\big(\{(\delta_k(I,k),\delta_v(I,v)) \mid (k,v)\in d(T)\}\big) \\ i' \triangleq \lambda T.\mathrm{dNorm}\big(\{(\delta_k(I,k),\mathsf{True}) \mid (k,\mathsf{True})\in i(T)\}\big)\end{array}}{\delta_c\big(I,(a,d,i)\big) \triangleq \big(\delta_a(I,a),d',i'\big)}$$

Figure 6.2: Abstract transfer function for a scalar instruction I

SCALAR STATEMENTS Scalar simple statements are interpreted not only in the scalar domain A, but also in all abstract segments in all containers, as shown in Figure 6.2. This is necessary, since the abstract keys and abstract values model also relationships with scalar variables. Thus, when a scalar variable is modified, all its relations to abstract keys and abstract dictionary elements must be updated.

DICTIONARY STATEMENTS We proceed now with the dictionary statements. First, recall that we have defined a function $\ell\colon (\mathcal{DictStmt} \cup \mathcal{DictPred}) \times \mathcal{State} \to \mathcal{Bool}$ that decides whether the dictionary statement may be executed in the given concrete state. The function ℓ can be lifted to $\mathcal{L}\colon (\mathcal{DictStmt} \cup \mathcal{DictPred}) \times \mathcal{Ctx} \to \mathcal{Bool}$ by defining

$$\mathcal{L}(s,c) \triangleq \mathsf{True} \quad \text{iff} \quad \forall_{(\rho,\tau)\in c}\,\ell(s,(\rho,\tau)) \; .$$

Finally, $\mathcal{L}$ is be abstracted in the domain C as $\Lambda_c\colon (\mathcal{DictStmt} \cup \mathcal{DictPred}) \times \mathcal{C} \to \mathcal{Bool}$ such that

$$\Lambda_c(s,a) \Rightarrow \mathcal{L}(s,\gamma_c(a)) \; .$$

The above implication does not define Λ_c unambiguously, thus it must be provided for each type of statements.

Recall also that we have defined a function $f\colon \mathcal{DictStmt} \to \mathcal{Stmt}$ that for a dictionary statement outputs a simple scalar statement which may modify the special scalar variable associated with the accessed dictionary.

Similarly as in the concrete domain, the abstract transfer function δ_c for dictionary statements depends in the domain C on Λ_c and f. For each dictionary statement $s \in \mathcal{DictStmt}$ and for each $a \in \mathcal{C}$ we define

$$\delta_c(s,a) \triangleq \begin{cases} \delta_c\big(f(s),\widehat{\delta_c}(s,a)\big) & \Lambda_c(s,a) = \mathsf{True} \\ \mathsf{AError} & \text{otherwise.} \end{cases}$$

The auxiliary function $\widehat{\delta_c}$ is the "real" abstract transfer function for the dictionary statements and is defined in the subsequent paragraphs.

Similarly, the abstract semantics of boolean predicates is given with respect to the function Λ_c:

$$\pi_c(\phi, a) \triangleq \begin{cases} \widehat{\pi_c}(\phi, a) & \Lambda_c(s, a) = \mathsf{True} \\ (\mathsf{AError}, \mathsf{AError}) & \text{otherwise.} \end{cases}$$

The function $\widehat{\pi_c}$ is defined below.

INITIALISATION The instruction $\mathsf{T} \leftarrow$ **new dict** that defines a new (empty) dictionary:

$$\widehat{\delta_c}\big(\mathsf{T} \leftarrow \textbf{new dict}, (a, d, i)\big) \triangleq \big(a, d[\mathsf{T} \mapsto \emptyset], i\big[\mathsf{T} \mapsto \{(\top_k, \mathsf{True})\}\big]\big) \ .$$

The dictionary creation instruction can be always executed, i.e.

$$\Lambda_c\big(\mathsf{T} \leftarrow \textbf{new dict}(a, d, i)\big) \triangleq \mathsf{True} \ .$$

READ We proceed now with a dictionary read $v_2 \leftarrow \mathsf{T}[v_1]$, where $\mathsf{T} \in \mathcal{V}ar_c$ and $v_1, v_2 \in \mathcal{V}ar$ (see Figure 6.3). Intuitively, we retrieve from $d(\mathsf{T})$ all abstract segments (there may be more than one) whose keys overlap with the key for the access $\mathsf{T}[v_1]$. Formally, we compute $k \in \mathcal{K}$ by adding to a the artificial key variable $\mathcal{k}$, assigning $\mathcal{k} \leftarrow v_1$ and converting the result into the domain K, i.e. $k \triangleq \kappa_{A \to K}\big(\delta_a(\mathcal{k} \leftarrow v_1, a{\uparrow_{\mathcal{k}}})\big)$. We take the join of all values in all abstract segments $(l, w) \in d(\mathsf{T})$ whose keys overlap with k, i.e. $v \triangleq \bigsqcup_v\big\{w \mid (l, w) \in d(\mathsf{T}), l \sqcap_k k \neq \bot_k\big\}$, convert v back to the domain A, assign $v_2 \leftarrow t$ and eliminate the special value-tracking variable t. At the end we invalidate the old value of v_2 in all abstract segments.

The function Λ_c returns True for the statement $v_2 \leftarrow \mathsf{T}[v_1]$ and abstract state (a, d, i) only if the retrieved element must be initialised. Let k be as above (i.e. $k \triangleq \kappa_{A \to K}\big(\delta_a(\mathcal{k} \leftarrow v_1, a{\uparrow_{\mathcal{k}}})\big)$). We define

$$\Lambda_c\big(v_2 \leftarrow \mathsf{T}[v_1], (a, d, i)\big) \triangleq \mathsf{True} \quad \text{iff} \\ \big\{l \mid (l, \mathsf{True}) \in i(\mathsf{T}), k \sqcap_k l \neq \bot_k\big\} = \emptyset \ .$$

The transfer rule is a sound abstraction of the concrete dictionary read access. This follows from the fact that δ_a is sound and the join $\sqcup_v$ is over-approximating.

Example 6.14. Let us consider a scalar variable $v_1 \in \mathcal{V}ar$ with $a(v_1) = [1, 4]$ and a container $\mathsf{T} \in \mathcal{V}ar_c$ modelled as $i(\mathsf{T}) = \big\{([8, \infty], \mathsf{True})\big\}$ and $d(\mathsf{T}) = \big\{([0, 2], [-2, 1]), ([3, 5], [4, 4]), ([6, 9], [2, 7])\big\}$. The read $v_2 \leftarrow \mathsf{T}[v_1]$ results in $a(v_2) = [-2, 1] \sqcup_i [4, 4] = [-2, 4]$.

$$\frac{\begin{array}{c} k \triangleq \kappa_{A \to K}\big(\delta_a(\mathscr{k} \leftarrow v_1, a{\Uparrow_{\mathscr{k}}})\big) \\ v \triangleq \bigsqcup_V \{w \mid (l, w) \in d(T), k \sqcap_k l \neq \bot_k\} \\ a' \triangleq \delta_a(v_2 \leftarrow \mathit{t}, \kappa_{V \to A}(v) \sqcap_a a{\Uparrow_{\mathit{t}}}) \\ (\_, d', i') \triangleq (a, d, i){\Updownarrow_{v_2}} \end{array}}{\widehat{\delta_c}\big(v_2 \leftarrow T[v_1], (a, d, i)\big) \triangleq (a'{\Downarrow_{\mathit{t}}}, d', i')}$$

Figure 6.3: Transfer rule for $v_2 \leftarrow T[v_1]$

$$\frac{\begin{array}{c} k \triangleq \kappa_{A \to K}\big(\delta_a(\mathscr{k} \leftarrow v_1, a{\Uparrow_{\mathscr{k}}})\big) \\ v \triangleq \kappa_{A \to V}\big(\delta_a(\mathit{t} \leftarrow v_2, a{\Uparrow_{\mathit{t}}})\big) \\ x \triangleq \mathrm{dNorm}\big(d(T) \cup \{(k, v)\}\big) \end{array}}{\widehat{\delta_c}\big(T[v_1] \leftarrow v_2, (a, d, i)\big) \triangleq (a, d[T \mapsto x], i)} \neg\mathfrak{S}_k(k)$$

Figure 6.4: Weak update $T[v_1] \leftarrow v_2$

UPDATES We define both weak and strong dictionary updates. The strong update can be performed only when the update may modify only one element. We formalise this by defining the following unary predicate $\mathfrak{S}$ on the domain K:

$$\mathfrak{S}(k) \triangleq \mathsf{True} \quad \text{iff} \quad \forall_{\sigma_1, \sigma_2 \in \gamma_k(k)} (\sigma_1|_{\mathcal{Var}} = \sigma_2|_{\mathcal{Var}} \Rightarrow \sigma_1 = \sigma_2) \ .$$

This definition is not very practical, thus we require that the domain K is equipped with a domain-specific predicate $\mathfrak{S}_k$ that implies $\mathfrak{S}$, i.e. $\forall_{k \in \mathcal{K}}\, \mathfrak{S}_k(k) \Rightarrow \mathfrak{S}(k)$. If $\mathfrak{S}_k(k) = \mathsf{True}$, then we say that k is a *singleton*.

Let us consider now an update $T[v_1] \leftarrow v_2$. We compute the abstract key $k \in \mathcal{K}$ in the same way as in the read. In a similar way we obtain the abstract value $v \in \mathcal{V}$.

If k is not a singleton (i.e. $\neg\mathfrak{S}_k(k)$) then a weak update is performed as defined in Figure 6.4. We add the new abstract segment (k, v) to $d(T)$ and compute $\mathrm{dNorm}\big(d(T) \cup \{(k, v)\}\big)$. The information about uninitialised elements is not altered, as for no specific key the dictionary must have been initialised by this update.

A weak update clearly over-approximates a concrete update, as it admits both the old and the new value for the modified element.

If k is a singleton, a strong update can be performed. The container $d(T)$ may already contain an abstract segment (l, w) that describes the updated element, i.e. $k \sqcap_k l \neq \bot_k$. The new value should be assigned only to the modified element, all other elements associated with keys

$$\frac{\begin{array}{c} k \triangleq \kappa_{A\to K}\big(\delta_a(\mathcal{k} \leftarrow v_1, a{\upharpoonleft}_{\mathcal{k}})\big) \\ v \triangleq \kappa_{A\to V}\big(\delta_a(t \leftarrow v_2, a{\upharpoonleft}_{t})\big) \\ x \triangleq d(T) \uplus (k, v) \\ y \triangleq i(T) \uplus (k, \mathsf{False}) \end{array}}{\widehat{\delta_c}\big(T[v_1] \leftarrow v_2, (a, d, i)\big) \triangleq \big(a, d[T \mapsto x], i[T \mapsto y]\big)}\ \mathfrak{S}_k(k)$$

Figure 6.5: Strong update $T[v_1] \leftarrow v_2$

abstracted by l should remain unchanged. Intuitively, we need to split the abstract key l into a collection of smaller keys $k, m_1, m_2, \dots m_j$ which represent together the same concrete keys as l. We say that a function $\zeta\colon \mathcal{K} \times \mathcal{K} \to \mathcal{P}(\mathcal{K})$ is a *decomposition* of an abstract key $l \in \mathcal{K}$ with respect to a singleton $k \in \mathcal{K}$ if:

- $\forall_{m_1, m_2 \in \zeta(l,k) \cup \{k\}}\, m_1 \neq m_2 \Rightarrow m_1 \sqcap_k m_2 = \bot_k$,
- $k \notin \zeta(l, k)$,
- $\gamma_k(k) \cup \big(\bigcup_{m \in \zeta(l,k)} \gamma_k(m)\big) = \gamma_k(l)$.

The definition of $\zeta(l, k)$ must be provided together with the domain K. It is sufficient to define the decomposition of the top element $\top_k$, as then the decomposition of an arbitrary $l \in \mathcal{K}$ can be defined as:

$$\zeta(l, k) \triangleq \{n \in \mathcal{K} \mid m \in \zeta(\top_k, k), m \sqcap_k l = n, n \neq \bot_k\}\,.$$

We define an operation $\uplus\colon \mathcal{D}(\mathcal{K}, \mathcal{V}) \times (\mathcal{K} \times \mathcal{V}) \rightharpoonup \mathcal{D}(\mathcal{K}, \mathcal{V})$ so that $\mathcal{d} \uplus (k, v)$ overwrites in $\mathcal{d}$ the elements at keys abstracted by k:

$$\begin{aligned} \mathcal{d} \uplus (k, v) \triangleq \big(\mathcal{d} \setminus \big\{(l, w) \mid (l, w) \in \mathcal{d}, l \sqcap_k k \neq \bot_k\big\}\big) \cup \big\{(k, v)\big\} \\ \cup \big\{(m, w) \mid (l, w) \in \mathcal{d}, l \sqcap_k k \neq \bot_k, m \in \zeta(l, k)\big\}\,. \end{aligned}$$

The operation $\mathcal{d} \uplus (k, v)$ is defined only if k is a singleton.

We are ready to define the strong update $T[v_1] \leftarrow v_2$. Let k and v be as in the weak update and let k be a singleton. The strong update overwrites in $d(T)$ the old value associated with the abstract key k and marks in $i(T)$ that the element at key abstracted by k must be initialised (see Figure 6.5).

For standard dictionaries that do not impose any restrictions on the keys or dictionary elements, the function $\Lambda_c(T[v_1] \leftarrow v_2, a)$ returns always True. For special types of dictionaries some specific checks may be necessary. For instance, an array cannot be written to outside of its bounds, which could be expressed (using the abstract semantics of boolean predicates π_a in the scalar domain A) as:

$$\Lambda_c\big(T[v_1] \leftarrow v_2, (a, d, i)\big) \triangleq \mathsf{True} \quad \text{iff} \quad \pi_a(0 \leqslant v_1, a) = (a, \bot_a) \\ \wedge \pi_a(v_1 < T.\mathtt{length}, a) = (a, \bot_a)\ .$$

The strong update forgets only the old value of the updated element (by the definition of the decomposition) and replaces it with an abstract value that over-approximates the inserted concrete value (as δ_a is a sound over-approximation), thus is a sound abstraction of the concrete update.

Example 6.15 (Weak update)**.** Consider the same container $T \in \mathcal{V}ar_c$ and scalar $v_1 \in \mathcal{V}ar$ as in Example 6.14, with an additional scalar $v_2 \in \mathcal{V}ar$ such that $a(v_2) = [6, 8]$. After the update $T[v_1] \leftarrow v_2$, $d(T)$ becomes $\big\{([0, 5], [-2, 8]), ([6, 9], [2, 7])\big\}$.

Example 6.16 (Strong update)**.** Consider scalars $v_1, v_2 \in \mathcal{V}ar$, such that $a(v_1) = [2, 2]$ and $a(v_2) = [7, 9]$ and a container $T \in \mathcal{V}ar_c$ with $d(T) = \big\{([0, 5], [1, 3])\big\}$, $i(T) = \big\{([0, \infty], \mathsf{True})\big\}$. The strong update $T[v_1] \leftarrow v_2$ modifies T so that $d(T) = \big\{([0, 1], [1, 3]), ([2, 2], [7, 9]), ([3, 5], [1, 3])\big\}$. It also marks that the updated element must be initialised, setting $i(T) = \big\{([0, 1], \mathsf{True}), ([3, \infty], \mathsf{True})\big\}$ (the superfluous abstract segment $([2, 2], \mathsf{False})$ is removed).

6.2.2 *Abstract Semantics of Boolean Predicates*

We admit boolean predicates that operate over a scalar variable and a dictionary access, such as $\phi(T[v_1], v_2)$, where $T \in \mathcal{V}ar_c$ and $v_1, v_2 \in \mathcal{V}ar$. This allows us to restrict the possible values of $T[v_1]$ as shown in Figure 6.6. As usual, k denotes the abstract key for the access $T[v_1]$ and v is the corresponding abstract value. If k is a singleton, then we can restrict v according to $\phi(t, v_2)$. The soundness of π_c follows from the soundness of the strong update, as in fact we perform two strong updates, one in the True and one in the False branch.

The function Λ_c is defined for $\phi(T[v_1], v_2)$ in the same manner as for the read access, i.e.

$$\Lambda_c\big(\phi(T[v_1], v_2), (a, d, i)\big) \triangleq \mathsf{True} \quad \text{iff} \\ \big\{l \mid (l, \mathsf{True}) \in i(T), k \sqcap_k l \neq \bot_k\big\} = \emptyset\ .$$

Example 6.17. Consider scalars $v_1, v_2 \in \mathcal{V}ar$ such that $a(v_1) = [4, 4]$ and $a(v_2) = [0, 0]$ and a container $T \in \mathcal{V}ar_c$ with $d(T) = \big\{([0, 4], [-2, 5])\big\}$.

$$\frac{\begin{array}{c} k \triangleq \kappa_{A\to K}\big(\delta_a(\mathcal{k} \leftarrow v_1, a{\upharpoonright}_{\mathcal{k}})\big) \\ v \triangleq \bigsqcup_v \{w \mid (l,w) \in d(T), k \sqcap_k l \neq \bot_k\} \\ (v_t, v_f) \triangleq \pi_v\big(\phi(t, v_2), v\big) \\ x_t \triangleq d(T) \uplus (k, v_t) \\ x_f \triangleq d(T) \uplus (k, v_f) \end{array}}{\widehat{\pi}_c\big(\phi(T[v_1], v_2), (a, d, i)\big) \triangleq (a, d[T \mapsto x_t], i), (a, d[T \mapsto x_f], i)}\ \mathfrak{S}_k(k)$$

Figure 6.6: Boolean predicate $\phi(T[v_1], v_2)$

A test $T[v_1] \leqslant v_2$ evaluates to two abstract states $(c_{\mathsf{True}}, c_{\mathsf{False}})$. In the True branch, the content of the container is given by

$$d_{\mathsf{True}}(T) = \big\{([0,3],[-2,5]),([4,4],[-2,0])\big\}$$

while in the False branch it is

$$d_{\mathsf{False}}(T) = \big\{([0,3],[-2,5]),([4,4],[1,5])\big\}\ .$$

Theorem 6.18. *The functions δ_c and π_c are sound abstractions of the concrete transfer functions t and p.*

Proof. This theorem is justified by the arguments presented for each of the transfer rules above. $\square$

6.3 EXAMPLES

We present now some sample instances of the generic domain described above.

6.3.1 *Arrays: Non-relational Abstraction*

We start with a very simple example, in which the domain is used to model arrays. We use here the domain of intervals I (see Section 2.1) and abstraction by parity P that is sketched below.

The abstraction by parity is a simple abstract domain $P(X)$ (over a finite set of variables X) with the set of abstract elements $\mathcal{P}$ equal to $X \to \mathcal{P}(\{0,1\})$, join $\sqcup_p$ and meet $\sqcap_p$ given as point-wise set union and intersection (hence $\top_p = \lambda x.\{0,1\}$ and $\bot_p = \lambda x.\emptyset$) and concretisation γ_p defined as

$$\gamma_p(p) \triangleq \{\rho \mid \forall_{x\in X}\, \exists_{c\in p(x)}\, \rho(x) \equiv c \mod 2\}\ .$$

Thus, $\{0\}$ and $\{1\}$ abstract even and odd integers, respectively.

We instantiate the domain C by fixing $A = P(\mathcal{V}ar)$, $K = P(\mathcal{V}ar \cup \{k\})$ and $V = I(\mathcal{V}ar \cup \{t\})$.

There are no singletons in the abstraction by parity, thus $\mathfrak{S}_p(p) =$ False for each $p \in \mathcal{P}$ and the decomposition is an empty partial function.

Example 6.19. Let us focus on a code snippet, where only each fourth element of an array is initialised:

```
1: $j \leftarrow 0$
2: $T \leftarrow$ new array$(n)$
3: while $j < n$ do
4:     $T[j] \leftarrow 5, j \leftarrow j + 4$
5: end while
```

Before the first loop iteration, no element of the array is initialised, j is even and n is either odd or even, i.e. $d(T) = \emptyset$, $i(T) = \{(\{0,1\}, \mathsf{True})\}$[1], $a(j) = \{0\}$ and $a(n) = \{0,1\}$. Inside the loop, 5 is assigned to an even element $T[j]$. This is a weak update, hence the initialisation information i is not modified. The instruction $j \leftarrow j+4$ does not change the parity of j. After the loop we get $d(T) = \{(\{0\}, [5,5])\}$, $i(t) = \{(\{0,1\}, \mathsf{True})\}$, $a(j) = \{0\}$ and $a(n) = \{0,1\}$. This invariant guarantees that no odd element in T is initialised (as $d(T)$ does not contain any abstract segment with key k such that $1 \in k(k)$). On the other hand, $i(T)$ states that each element of T may still be uninitialised. Note also that the only one abstract key in $d(T)$ describes a set of non-contiguous concrete array elements.

6.3.2 *Arrays: Relational Abstraction*

We instantiate now the generic domain C using a very simple relational domain of upper bounds, which is an enriched version of the domain of strict upper bounds described in Section 2.3.

We demonstrate the analysis on examples of partial array initialisation (Figure 6.7) and quicksort partition procedure (Figure 6.8).

In the domain of upper bounds B over a finite set of variables X, the set of abstract states $\mathcal{B}$ is a map $X \to \mathcal{P}(X \times \{<, \leqslant\})$. Intuitively, each variable x is mapped to a set of variables greater than x (with an indicator, whether the constraint is strict). The concretisation γ_b is given by:

$$\gamma_b(b) \triangleq \{\rho \mid \forall_{x,y \in X}\, (y, \lhd) \in b(x) \Rightarrow \rho(x) \lhd \rho(y)\}\,.$$

1 In the abstract segments we show only the abstract values of the key and value-tracking variables

The meet $\sqcap_b$ can be defined just as a point-wise set union. The result of a join $b_1 \sqcup_b b_2$ contains for each variable x the less restrictive (in the obvious sense) constraints from $b_1(x)$ and $b_2(x)$. The domain of upper bounds B is finite, hence the widening can be defined as the join.

We instantiate now the domain C by fixing $A = B(\mathcal{V}ar)$, $K = B(\mathcal{V}ar \cup \{\mathcal{k}\})$ and $V = B(\mathcal{V}ar \cup \{t\})$. We additionally assume that there is a special variable $v_0 \in \mathcal{V}ar$ that is always equal to zero.

SINGLETON AND DECOMPOSITION The abstract key is a singleton, if the key variable $\mathcal{k}$ is equal to some numerical variable x:

$$\mathfrak{S}_b(k) = \mathsf{True} \quad \text{iff} \quad \exists_{x \in \mathcal{V}ar}\, (x, \leqslant) \in k(\mathcal{k}) \wedge (\mathcal{k}, \leqslant) \in k(x)\ .$$

We define now a decomposition function ζ_b of the top element $\top_b$. Each element of $\zeta_b(\top_b, k)$ originates from k by negating any (non-empty) subset of constraints:

$$\zeta_b(\top_b, k) \triangleq \Big\{ b \in \mathcal{K} \mid (y, \lhd) \in b(x) \Leftrightarrow \Big((y, \lhd) \in k(x) \\ \vee \big((x, \blacktriangleleft) \in k(y) \wedge \{\lhd, \blacktriangleleft\} = \{<, \leqslant\} \big) \Big) \Big\} \setminus \{k\}\ .$$

Fact. $\zeta_b(\top_b, k)$ is a decomposition of the top element $\top_b$ and it can be used to instantiate the generic definition of ζ.

TRANSFER FUNCTION We demonstrate only one example of the transfer function δ_b, namely $\delta_b(j \leftarrow j+1, b)$. We choose this statement, as it occurs in the examples presented later in this section. Let us assume that the set of scalar values $\mathbb{V}$ is fixed as $\mathbb{Z}$. Basically, if there is in b an inequality $x \leqslant j$, then after the increment $j \leftarrow j+1$, it is transformed to $x < j$. Similarly, $j < y$ is transformed to $j \leqslant y$. On the other hand, an inequality $j \leqslant z$ is lost. More formally, $\delta_b(j \leftarrow j+1, b)$ is given by:

$$\lambda v. \begin{cases} \big\{(x, \leqslant) \mid (x, <) \in b(j)\big\} & \text{if } v = j, \\ \Big(b(v) \cup \big\{(j, <) \mid (j, \leqslant) \in b(v)\big\} \Big) \setminus \big\{(j, \leqslant)\big\} & \text{otherwise.} \end{cases} \tag{6.20}$$

EXAMPLES In the examples below we do not write all bounds explicitly. Instead, for each abstract key k we show only constraints that involve the key variable $\mathcal{k}$. And so, if $(x, \lhd) \in k(\mathcal{k})$, we write "$\lhd x$". If $(\mathcal{k}, \lhd) \in k(y)$, then we write "$\rhd y$". If $(z, \leqslant) \in k(\mathcal{k})$ and $(\mathcal{k}, \leqslant) \in k(z)$, then we use an abbreviation "$=z$". Similarly, for each abstract value v we show only the constraints involving the value-tracking variable t.

```
1: $x \leftarrow 0$, $j \leftarrow 0$, $T \leftarrow$ new array$(n)$
2: while $x < n$ do
3:     $x \leftarrow x + 1$
4:     if $\phi(x)$ then
5:         $T[j] \leftarrow x$, $j \leftarrow j + 1$
6:     end if
7: end while
```

Figure 6.7: Partial array initialisation

Example 6.21 (Array Initialisation)**.** Let us start with a standard partial array initialisation code presented in Figure 6.7. Our analysis successfully detects that after this code fragment first j elements of T are initialised to values smaller or equal to x. The analysis also ensures that all array accesses are correct.

The statement $T \leftarrow$ **new array**(n) creates a new, empty array. The possible indices of T range over $[0, n)$. Note that at this program point $j = 0$ and $T.\mathit{length} = n$, thus the range of indices i of uninitialised array elements can be written as $v_0 = j \leqslant i < n = T.\mathit{length}$. This observation justifies the abstract state just before the loop, which is $d(T) = \emptyset$ and $i(T) = \{(\{<n, <T.\mathit{length}, \geqslant j, \geqslant v_0\}, \mathsf{True})\}$.

We assume that nothing can be statically determined about the test $\phi(x)$, thus $\pi_c(\phi(x), c) = (c, c)$. Let us focus now on the array modification $T[j] \leftarrow x$ in line 4. First observe that the access is correct, i.e. the analysis of scalars captures that the tests $0 \leqslant j$ and $j < T.\mathit{length}$ are fulfilled (see the definition of Λ_c for an array update). The abstract key $k \in \mathcal{K}$ for the array access $T[j]$ contains the constraints $\mathcal{k} \leqslant j$ and $j \leqslant \mathcal{k}$, thus k is a singleton and the strong update is performed. The inserted abstract segment is $(k, v) = (\{=j, <n, \geqslant v_0\}, \{=x\})$ (we drop here constraints not important in the analysis). After this update, the content of the array is modelled as $d(T) = \{(\{=j, <n, \geqslant v_0\}, \{=x\})\}$. The initialisation information is $i(T) = \{(\{<n, >j\}, \mathsf{True})\}$.

The increment of the scalar $j \leftarrow j + 1$ modifies the array so that

$$
\begin{aligned}
d(T) &= \Big\{ \big(\{<j, <n, \geqslant v_0\}, \{=x\}\big) \Big\} \\
i(T) &= \Big\{ \big(\{<n, \geqslant j\}, \mathsf{True}\big) \Big\} .
\end{aligned}
$$

Let us explain the above equalities in more detail. We focus on $i(T)$, as the case of $d(T)$ is similar. Recall that each scalar statement is interpreted in all abstract segments in $d(T)$ and $i(T)$. Let us focus on the statement $j \leftarrow j + 1$ and the abstract segment $(\{>j, <n\}, \mathsf{True}) \in i(T)$

in the state before this increment. The shown (fragment of the) abstract key encodes the following inequalities: $j < k$ and $k < n$. As we have discussed when presenting the transfer function $\delta_b(j \leftarrow j+1, b)$, the inequality $j < k$ is transformed into $j \leqslant k$. The remaining inequalities are unaffected. This explains the abstract segment $(\{{<}n, {\geqslant}j\}, \mathsf{True})$.

To understand the example even better, one could imagine modifying the code snippet by replacing $j \leftarrow j+1$ with $j \leftarrow j+2$. In the considered abstract segment $(\{{>}j, {<}n\}, \mathsf{True})$ the constraint $j < k$ is lost, resulting in an abstract segment $(\{{<}n\}, \mathsf{True})$. In this case the information about initialised elements is lost.

During the second loop iteration, the increment $x \leftarrow x+1$ modifies $d(T)$ so that $d(T) = \{(\{{<}j, {<}n, {\geqslant}v_0\}, \{{<}x\})\}$ (while $i(T)$ remains unchanged). The assignment $T[j] \leftarrow x$ results in

$$d(T) = \Big\{ \big(\{{<}j, {<}n, {\geqslant}v_0\}, \{{<}x\}\big), \big(\{{=}j, {<}n, {\geqslant}v_0\}, \{{=}x\}\big) \Big\}$$

and

$$i(T) = \Big\{ \big(\{{<}n, {>}j\}, \mathsf{True}\big) \Big\} \,.$$

Additionally, $\Lambda_c(T[j] \leftarrow x, (a, d, i))$ evaluates to True, as the analysis of scalars ensures that $0 = v_0 \leqslant j$ and $j < T.\mathtt{length}$. Finally, using the same arguments as in the first loop iteration, after $j \leftarrow j+1$ we get:

$$\begin{aligned} d(T) &= \Big\{ \big(\{{<}j, {<}n, {\geqslant}v_0\}, \{{\leqslant}x\}\big) \Big\} \\ i(T) &= \Big\{ \big(\{{<}n, {\geqslant}j\}, \mathsf{True}\big) \Big\} \,. \end{aligned}$$

When analysing the next loop iteration, it turns out that this is already the loop invariant. The first abstract segment in $i(T)$ guarantees that all elements at indices smaller than j were initialised, while $d(T)$ ensures that all values of these elements are smaller or equal than x.

Example 6.22 (Partition). In the next example, we analyse the partition procedure from Figure 6.8. The array T is modified so that elements not greater than x are moved to the front, i.e. there exists an index j such that $\forall_{0 \leqslant m < j} T[m] \leqslant x$ and $\forall_{j < n < T.\mathtt{length}} T[n] \geqslant x$ (we assume that the array is fully initialised). For brevity, we omit the initialisation analysis i and focus on the array content d. We omit the discussion about Λ_c, as the correctness of all array accesses in this example can be shown in a straightforward way: we have assumed that the array is fully initialised and at each time when an array is accessed $0 \leqslant l < T.\mathtt{length}$ and $0 \leqslant r < T.\mathtt{length}$.

At the beginning of the procedure the array T may contain arbitrary values, i.e. $d_1(T) = \{(\{{<}T.\mathtt{length}, {\geqslant}v_0\}, \top_b)\}$.

```
procedure PARTITION($T, x$)
$l \leftarrow 0$, $r \leftarrow T.length - 1$
while $l < r$ do
    if $T[l] \leqslant x$ then
        $l \leftarrow l + 1$
    else if $T[r] \geqslant x$ then
        $r \leftarrow r - 1$
    else
        $y \leftarrow T[r]$, $z \leftarrow T[l]$, $T[r] \leftarrow z$, $T[l] \leftarrow y$
    end if
end while
```

Figure 6.8: Partition procedure

After the two assignments to scalars l and r in line 2, we get $d_2(T) = \{(\{\leqslant r, < T.length, \geqslant l, \geqslant v_0\}, \top_b)\}$. Below we omit the obvious constraints "$\geqslant v_0$" and "$<T.length$". Consider the program point before the instruction in line 5. As we have $l < r$, the test $T[l] \leqslant x$ ensures

$$d_5(T) = \big\{\big(\{=l, <r\}, \{\leqslant x\}\big),\ \big(\{\leqslant r, >l\}, \top_b\big)\big\}\ .$$

After $l \leftarrow l + 1$ the array gets $d_5'(T) = \big\{\big(\{<l, <r\}, \{\leqslant x\}\big),\ \big(\{\leqslant r, \geqslant l\}, \top_b\big)\big\}$, which can be justified in the same fashion as we have done it in Example 6.21. In the second branch of the if statement both $T[l] > x$ and $T[r] \geqslant x$:

$$d_7(T) = \big\{\big(\{=l, <r\}, \{>x\}\big),\ \big(\{<r, >l\}, \top_b\big),\ \big(\{=r, >l\}, \{\geqslant x\}\big)\big\}$$

and after decreasing the scalar r:

$$d_7'(T) = \big\{\big(\{=l, \leqslant r\}, \{>x\}\big),\ \big(\{\leqslant r, >l\}, \top_b\big),\ \big(\{>r, >l\}, \{\geqslant x\}\big)\big\}\ .$$

In line 9, $T[l] > x$ and $T[r] < x$. After the two strong updates $T[l] \leftarrow y$ and $T[r] \leftarrow z$ we get:

$$d_9'(T) = \big\{\big(\{=l, <r\}, \{<x\}\big),\ \big(\{<r, >l\}, \top_b\big),\ \big(\{=r, >l\}, \{>x\}\big)\big\}\ .$$

After joining the states d_5', d_7' and d_9', at the end of the first iteration we get

$$d_{11}(T) = \big\{\big(\{<l, <r\}, \{\leqslant x\}\big), \big(\{\leqslant r, \geqslant l\}, \top_b\big),\ \big(\{>r, >l\}, \{\geqslant x\}\big)\big\}\ .$$

The domain of bounds is finite, hence we apply the join instead of the widening before the next loop iteration. Note that $d_2(T) \sqcup_d d_{11}(T) =$

```
j ← 0, T ← new array(n)
while j < n do
    if j ≡ 0 mod 2 then
        T[j] ← 0
    else
        T[j] ← 1
    end if
    j ← j + 1
end while
```

Figure 6.9: Array initialisation depending on index parity

$d_{11}(T)$ ($d_2(T)$ is a subset of $d_{11}(T)$). It turns out that $d_{11}(T)$ is a loop invariant. After the loop, $l \geqslant r$, hence the invariant describes the desired property (note that the abstract segment $\big(\{\leqslant r, \geqslant l\}, \top_b\big)$ combined with $l \geqslant r$ is equivalent to $\big(\{=r, =l\}, \top_b\big)$ and contains at most one element), i.e. for $0 \leqslant m < j$, $T[m] \leqslant x$ and for $j < n < T.length$, $T[n] \geqslant x$.

6.3.3 *Arrays: Abstraction Using a Product*

We use now both the non-relational and relational analysis in one example, i.e. we instantiate the domain C by fixing abstraction of scalars and keys as the product of abstractions by parity and upper bounds, i.e. $A = P \times B$, $K = P \times B$ and $V = I$ (the domain of intervals) over the sets of variables as usual. In the product domain $P \times B$ all domain operations are given point-wise. An element $(p, b) \in \mathcal{P} \times \mathcal{B}$ is a singleton if either $p \in \mathcal{P}$ is a singleton (which never happens) or $b \in \mathcal{B}$ is a singleton. The decomposition $\zeta\big((l_b, l_p), (k_b, k_p)\big)$ is given by

$$\big\{(m, \top_p) \mid m \in \zeta_b(l_b, k_b)\big\} .$$

Using such abstraction we can show that in the code in Figure 6.9, all array elements at odd indices are initialised to 0, while all elements at even indices are equal to 1.

In the first loop iteration $a_1(j) = (\{\geqslant v_0, <n\}, e)$. Let $c = (a, d, i)$ denote the state before the loop. The test $j = 0 \mod 2$ results in $c_t = c$ and $c_f = \bot_c$, thus the array before the increment $j \leftarrow j + 1$ is abstracted as:

$$d_8(T) = \Big\{\big((\{=j\}, e), [0, 0]\big)\Big\} \quad \text{and} \quad i_8(T) = \Big\{\big((\{>j, <n\}, \top_p), \mathsf{True}\big)\Big\} .$$

As the analysis of scalars captures that $0 \leqslant j$ and $j < n$, Λ_c returns True for the array access. After the increment $j \leftarrow j + 1$ the array is abstracted by:

$$d_9(T) = \Big\{\big((\{<j\}, e), [0, 0]\big)\Big\} \quad \text{and} \quad i_9(T) = \Big\{\big((\{\geqslant j, <n\}, \top_p), \mathsf{True}\big)\Big\}$$

while the scalar j is $a_9(j) = (\{>v_0, \leqslant n\}, o)$.

In the next loop iteration j is either odd or even (as we join a_1 with a_9). It is easy to see that Λ_c returns True for array accesses in both branches of the if statement. After the whole if statement we get:

$$d_8(T) = \Big\{\big((\{<j\}, e), [0, 0]\big), \big((\{=j\}, e), [0, 0]\big), \big((\{=j\}, o), [1, 1]\big)\Big\}$$

and

$$i_8(T) = \Big\{\big((\{>j, <n\}, \top_p), \mathsf{True}\big)\Big\} .$$

After incrementing $j \leftarrow j + 1$ we obtain

$$d_9(T) = \Big\{\big((\{<j\}, e), [0, 0]\big), \big((\{<j\}, o), [1, 1]\big)\Big\}$$

and

$$i_9(T) = \Big\{\big((\{\geqslant j, <n\}, \top_p), \mathsf{True}\big)\Big\} .$$

It is now easy to check that the state (a_9, d_9, i_9) is already a loop fixpoint. After the loop $j \geqslant n$ and $i_{9'}(T) = \emptyset$ (as the abstract key would contain contradictory constraints $k \geqslant n$ and $k < n$), hence all array elements at even and odd indices smaller than j are initialised to zero and one, respectively.

6.3.4 *Dictionaries*

In the next example our technique is used to model a string-keyed dictionary. This example is inspired by the way in which objects are represented in dynamic programming languages (such as Python).

In dynamic languages an object is stored as a dictionary, where each entry represents an object attribute. Attributes can be added and removed during program execution. Moreover, different types of values may be assigned to the same attribute at different points of the program execution. For instance `obj.attr` may be at some point an integer, at another point a string, while somewhere else it may refer to a function. This flexibility, although sometimes useful and convenient, leads often to serious errors. When a missing attribute is accessed, the program fails with an `AttributeError`. When an attribute does not match the expected type (for example a string is encountered in an arithmetic operation), a runtime `TypeError` is raised. In statically typed languages these types of errors are eliminated during compilation. In dynamic languages they may arise during program execution, thus to avoid them, a significant testing effort is required.

We use now our dictionary analysis technique to statically detect these types of problems.

STRING ANALYSIS An object is represented as a string-keyed dictionary, thus we need to develop an abstract domain for modelling string values. There already exist such domains, both non-relational [48] and relational [44, 67].

We present here our own tiny abstract domain, in which each string variable is represented using a generalised regular expression [59]. A regular expression $r \in \mathcal{r}(\Sigma)$ over a finite alphabet Σ is defined by the following grammar:

$$r ::= \emptyset \mid \epsilon \mid a \in \Sigma \mid r_1 \cdot r_2 \mid r_1 \vee r_2 \mid r_1 \wedge r_2 \mid r^* \mid \neg r \ .$$

Note that in addition to the standard concatenation, alternative and Kleene star, we allow conjunction and complement operations. These extensions are purely syntactical, i.e. each such generalised regular expression can be rewritten using only the standard operations. We write $L(r)$ to denote the language recognised by the regular expression r.

Given some set of variables X, in the domain $R(X)$ each variable $v \in X$ is mapped to a regular expression, i.e. $\mathcal{R} = X \to \mathcal{r}(\Sigma)$. The meet $\sqcap_r$ and join $\sqcup_r$ are given as point-wise conjunction and alternative. The concretisation is given by:

$$\gamma_r(a) \triangleq \{\rho \colon X \to \mathit{String} \mid \forall_{v \in X} \rho(v) \in L(a(v))\} \ .$$

The top $\top_r$ is equal to $\lambda v.\Sigma^*$, while the bottom $\perp_r$ is $\lambda v.\emptyset$.

WIDENING The most complicated operation is the widening. It is defined in terms of non-deterministic finite state automata (which are equivalent to regular expressions [40]). Roughly, given two regular expressions $r_1, r_2 \in \mathcal{r}(\Sigma)$ and the corresponding automata $A(r_1)$ and $A(r_2)$, we construct an automaton A' by adding transitions to $A(r_1)$ so that $L(r_1) \cup L(r_2) \subseteq L(A')$. As we never add new states and the size of the alphabet Σ is finite, our widening is guaranteed to stabilise.

In the algorithm for computing the widening of two automata A_1 and A_2 we map states of A_2 to states of A_1 in the following way:

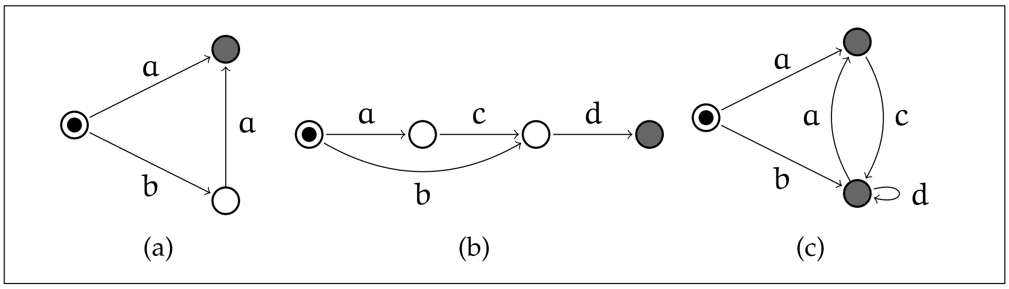

Figure 6.10: Two automata (a) and (b) and their widening (c). Input and accepting states are marked as circles with dots and filled circles, respectively.

1. the (unique) input state of A_2 is mapped to the input state of A_1,
2. consider a state $s_2 \in States(A_2)$ that is already mapped to some $s_1 \in States(A_1)$. Consider a transition (in A_2) from s_2 to some s_2' labelled with a letter $l \in \Sigma$. There are three possibilities:
   - s_2' is already mapped to some $s_1' \in States(A_1)$. In this case, we add (in A_1) a transition from s_1 to s_1' labelled with l (if this transition already exists, nothing is added),
   - s_2' is not mapped and there exits in A_1 a transition from s_1 to some s_1' labelled with l. In this case s_2' is mapped to s_1' (if there are many such transitions in A_1, any of them can be taken),
   - s_2' is not mapped and there is no transition form s_1 labelled with l in A_1. In this case we map s_2' to s_1 and add a loop on s_1 with label l,
3. it may happen that we map an accepting state $s_2 \in States(A_2)$ to a non-accepting state $s_1 \in States(A_1)$. In this case, we turn s_1 into an accepting state as well.

We proceed with steps 2 and 3 until all states in A_2 are mapped. The modified automaton A_1 obtained at the end of the procedure is considered as the result of the widening.

Finally, the widening $\mathcal{a} \nabla_r \mathcal{b}$ is for each variable $v \in \mathcal{Var}$ defined as the regular expression equivalent to the automaton constructed by the above algorithm applied to $\mathcal{a}(v)$ and $\mathcal{b}(v)$.

TRANSFER FUNCTION The only operation on strings that we need, is the string concatenation. The transfer function $\delta_r(z \leftarrow x + y, \mathcal{a})$ is just given by

$$\delta_r(z \leftarrow x + y, \mathcal{a}) \triangleq \mathcal{a}[z \mapsto \mathcal{a}(x) \cdot \mathcal{a}(y)] \ .$$

```
1:  at ← "b"
2:  repeat
3:      setattr(obj, at, 6)
4:      at ← at + "c"
5:  until random() = False
6:  if random() = True then
7:      obj.x ← 8
8:  else
9:      obj.x ← "text"
10: end if
11: print obj.b - 1
12: print obj.bcc - 1
13: print obj.x - 1
```

Figure 6.11: Dynamically added attributes

ATTRIBUTE ANALYSIS In our example, we will abstract each attribute of an object by its possible type. For simplicity, we consider only integer and string attributes. The types can be abstracted in a tiny domain $\mathsf{T}(X)$, where the set of abstract states is $\mathcal{T} = X \to \mathcal{P}(\{\mathsf{Int}, \mathsf{Str}\})$, with the domain operations given as point-wise set union and intersection.

OBJECT ABSTRACTION As already mentioned, we use our dictionary analysis technique to model possible attributes of an object. We instantiate the generic domain C by fixing the abstraction of scalars A as $\mathsf{R}(\mathcal{Var})$, key abstraction as $\mathsf{K} = \mathsf{R}(\mathcal{Var} \cup \{\mathcal{k}\})$ and value abstraction as $\mathsf{V} = \mathsf{T}(\mathcal{Var} \cup \{t\})$.

An abstract key $k \in \mathcal{K}$ is a singleton, if the regular expression $k(\mathcal{k})$ can be generated using only the productions $r ::= \epsilon \mid a \in \Sigma \mid r_1 \cdot r_2$. The decomposition is given by $\zeta_r(l, k) = \{\neg k \wedge l\}$.

Example 6.23. We may now demonstrate the analysis on the code fragment shown in Figure 6.11. We write attribute accesses in a Python-like style, however in fact they are just dictionary accesses. And so `obj.x` is equivalent to `obj['x']` and `setattr(obj,v,u)` can be written as `obj[v] ← u`.

For brevity, in the abstract segments we show only the abstract values of the key and value-tracking variables $\mathcal{k}$ and t. In the first loop iteration, the scalar variable at is abstracted as $a(\mathsf{at}) = \mathsf{b}$. Thus, after setting the attribute in line 3, the object is modelled as $d_3(\mathsf{obj}) = \big\{(\mathsf{b}, \{\mathsf{Int}\})\big\}$ and $i_3(\mathsf{obj}) = \big\{(\neg\mathsf{b}, \mathsf{True})\big\}$. In the next loop iteration at is widened (using our widening on automata for regular expressions) to $\mathtt{a}(\mathtt{at}) = \mathtt{bc}^*$. Since $\mathtt{bc}^*$ is not a singleton, the `setattr` in line 3 results in a weak update and after the loop we get $\mathtt{d}_5(\mathtt{obj}) = \big\{(\mathtt{bc}^*, \{\mathtt{Int}\})\big\}$ and $\mathtt{i}_5(\mathtt{obj}) = \big\{(\neg \mathtt{b}, \mathsf{True})\big\}$. After the (strong) update in line 7, the object is modelled as

$$\mathtt{d}_7(\mathtt{obj}) = \big\{(\mathtt{bc}^*, \{\mathtt{Int}\}), (\mathtt{x}, \{\mathtt{Int}\})\big\} \quad \mathtt{i}_7(\mathtt{obj}) = \big\{(\neg \mathtt{b} \wedge \neg \mathtt{x}, \mathsf{True})\big\} .$$

Similarly, after the assignment in the second branch we get:

$$\mathtt{d}_9(\mathtt{obj}) = \big\{(\mathtt{bc}^*, \{\mathtt{Int}\}), (\mathtt{x}, \{\mathtt{Str}\})\big\} \quad \mathtt{i}_9(\mathtt{obj}) = \big\{(\neg \mathtt{b} \wedge \neg \mathtt{x}, \mathsf{True})\big\} .$$

Thus, joining the above two states gives

$$\mathtt{d}_{10}(\mathtt{obj}) = \big\{(\mathtt{bc}^*, \{\mathtt{Int}\}), (\mathtt{x}, \{\mathtt{Int}, \mathtt{Str}\})\big\} \ \mathtt{i}_{10}(\mathtt{obj}) = \big\{(\neg \mathtt{b} \wedge \neg \mathtt{x}, \mathsf{True})\big\} .$$

We can now prove that the attribute usage in line 11 is correct. The attribute $\mathtt{b}$ must be present in $\mathtt{obj}$ and it is an integer. The access `obj.bcc` is detected as unsafe (possible `AttributeError`). The analysis captured that $\mathtt{obj}$ may contain any attribute $\mathtt{bc}^*$, each of them of type $\mathtt{Int}$, but it does not guarantee that $\mathtt{bcc}$ (or any $\mathtt{bc}^+$) is present in $\mathtt{obj}$. Finally, the last instruction is signalled as unsafe, because `obj.x` does not need to be an integer at this program point (possible `TypeError`).

Our technique is flexible enough to capture non-trivial properties of both dictionary keys and values. The abstract segment $(\mathtt{bc}^*, \{\mathtt{Int}\})$ means that elements at keys matching the regular expression $\mathtt{bc}^*$ may be only of type $\mathtt{Int}$. This type of properties cannot be modelled by any of the approaches sketched in the previous chapter.

6.4 POSSIBLE EXTENSIONS

In our domain it is possible to express constraints between dictionary keys and scalars and between dictionary values and scalars, but it is not possible to model relationships between dictionary keys and dictionary values. However, our domain can be easily modified to capture also this kind of relationships. This can be achieved by abstracting dictionary values in a domain V over the set of variables $\mathcal{V}ar \cup \{k, t\}$, where k and t are the key and value-tracking variables, respectively. Clearly, such extension would require also some (rather straightforward) modifications to the concretisation and transfer function. After this extension the domain would be powerful enough to express properties such as "each value in the dictionary is equal to the corresponding key".

After the proposed extension, the dictionary keys are modelled both by abstract keys and abstract dictionary values. One could try to eliminate this redundancy by removing the key abstraction from the abstract segments. In this case an abstract dictionary would just be defined as a finite set of abstract values. However, we prefer to keep the abstract keys, as this allows us to maintain the property that each abstract segment models disjoint sets of concrete dictionary elements.

Another possible modification is to get rid of the abstraction of scalars. As we keep the relations to scalars within the abstract segments, the additional element $a \in \mathcal{A}$ is sometimes superfluous. However, we have decided to keep it for two reasons:

- it makes the presentation clearer,
- it is useful when the domain is instantiated with non-relational domains: the abstraction of keys and dictionary values can be chosen to model only the special variables k and v, respectively.

6.5 PROOFS

In this section we present proofs of theorems and lemmas stated in this chapter.

6.5.1 *Proof of Lemma 6.2*

Let us recall the proved lemma:

Lemma (6.2; recalled). *The meet operator $\sqcap_d$ in the lattice $\langle \mathcal{D}, \sqcup_d, \sqcap_d \rangle$ is well defined, i.e. $a \sqcap_d b \in \mathcal{D}$.*

Proof. Clearly $a \sqcap_d b \in \mathcal{P}_{fin}(\mathcal{K} \times \mathcal{V})$. Directly from the definition it also follows that for each $(k, v) \in a \sqcap_d b$, $k \neq \bot_k$ and $v \neq \bot_v$.

Now we show that no two distinct abstract keys in $a \sqcap_d b$ may overlap. Let $(k_{a_1} \sqcap_k k_{b_1}, v), (k_{a_2} \sqcap_k k_{b_2}, w) \in a \sqcap_d b$ where $(k_{a_1}, v_{a_1}), (k_{a_2}, v_{a_2}) \in a$ and $(k_{b_1}, v_{b_1}), (k_{b_2}, v_{b_2}) \in b$. Since $a, b \in \mathcal{D}$, we have following possibilities:

1. $(k_{a_1}, v_{a_1}) = (k_{a_2}, v_{a_2})$ and $(k_{b_1}, v_{b_1}) = (k_{b_2}, v_{b_2})$. In this case we immediately get $(k_{a_1} \sqcap_k k_{b_1}, v) = (k_{a_2} \sqcap_k k_{b_2}, w)$,
2. $k_{b_1} \sqcap_k k_{b_2} = \bot_k$ (the case when $k_{a_1} \sqcap_k k_{a_2} = \bot_k$ is symmetric). We are to show that

$$(k_{a_1} \sqcap_k k_{b_1}) \sqcap_k (k_{a_2} \sqcap_k k_{b_2}) = \bot_k \,.$$

Using the commutativity and associativity of $\sqcap_k$ we get

$$\begin{aligned}(k_{a_1} \sqcap_k k_{b_1}) \sqcap_k (k_{a_2} \sqcap_k k_{b_2}) &= (k_{a_1} \sqcap_k k_{b_1}) \sqcap_k (k_{b_2} \sqcap_k k_{a_2}) \\ &= k_{a_1} \sqcap_k \big((k_{b_1} \sqcap_k k_{b_2}) \sqcap_k k_{a_2}\big) \\ &= k_{a_1} \sqcap_k (\bot_k \sqcap_k k_{a_2}) = \bot_k \,.\end{aligned}$$

□

6.5.2 *Proof of Lemma 6.5*

Again, we start with recalling the statement:

Lemma (6.5; recalled). *Let $\langle \mathcal{A}, \sqcup_a, \bot_a, \top_a\rangle$ be a complete lattice and let S be a finite subset of $\mathcal{A}$. The least disjoint partition of S exists and is uniquely defined.*

Proof. We define a sequence of minimally defined partial functions $f_i \colon \mathcal{P}(S) \rightharpoonup S$ such that

- $f_0(\{a\}) = a$ for each $a \in S$,
- $f_i(\bigcup \mathcal{Y}) = \bigsqcup_{a Y \in \mathcal{Y}} f_{i-1}(Y)$, for every maximal family of sets $\mathcal{Y}$ (with respect to inclusion of family sets) such that for each $Y_0, Y_k \in \mathcal{Y}$ there exist $Y_1, \dots Y_{k-1} \in \mathcal{Y}$ such that $f_{i-1}(Y_j) \sqcap_a f_{i-1}(Y_{j+1}) \neq \bot_a$,
- for $i \geqslant 0$, f_i is undefined on arguments other than mentioned above.

We show that the domain of $f_{|S|}$ is equal to the least disjoint partition of S. We start with the following auxiliary property:

Lemma 6.24. *For each $i \geqslant 0$ and for each $Y \in Dom(f_i)$, all $a \in Y$ must belong to the same set in any disjoint partition of S.*

We prove this property by induction over i:

1. $i = 0$: each $Y \in Dom(f_0)$ is a singleton. As the least disjoint partition is a partition, each element of S must belong to some set. So Lemma 6.24 holds,
2. assume that Lemma 6.24 holds for $i - 1$. Let us consider $\mathcal{Y}$ as in the definition of f_i:
   - if $\mathcal{Y} = \{Y\}$, then $f_i(\bigcup \mathcal{Y}) = f_{i-1}(Y)$, so the property holds

- otherwise consider any two $Y_j, Y_k \in \mathcal{Y}$ and a sequence $Y_j, Y_{j+1}, \ldots, Y_k$ such that $f_{i-1}(Y_r) \sqcap_a f_{i-1}(Y_{r+1}) \neq \bot_a$. We show that all elements from Y_j and Y_k must belong to the same set in any disjoint partition. We prove this by induction over the length of sequence $Y_j, \ldots, Y_k$:
  a) if the length is 2, then $f_{i-1}(Y_j) \sqcap_a f_{i-1}(Y_k) \neq \bot_a$. Using the inductive hypothesis for $i-1$ and the definition of a disjoint partition, we immediately get that all elements from Y_j and Y_k must belong to the same set in the disjoint partition.
  b) assume that the length of the path between Y_j and Y_k is n and that the examined property holds for paths of length $n-1$. Consider the element Y_{k-1} (which is just before Y_k in the sequence). The distance between Y_j and Y_{k-1} is $n-1$, so all elements from Y_j and Y_{k-1} must belong to the same set in any disjoint partition (inductive hypothesis). But $f_{i-1}(Y_{k-1}) \sqcap_a f_{i-1}(Y_k) \neq \bot_a$, so as in the previous case, from the inductive hypothesis for $i-1$ and definition of a disjoint partition, all elements from Y_{k-1} and Y_k must belong to one set, which implies that also elements from Y_j and Y_k belong to the same set in each disjoint partition.

This ends the proof of Lemma 6.24.

The sequence $\langle f_i \rangle$ stabilises after at most $|S|$ steps. Note that in each iteration, either $f_i = f_{i-1}$ or $|Dom(f_i)| < |Dom(f_{i-1})|$. Since $|Dom(f_0)| = |S|$, $f_{|S|}$ must be a fixpoint of the sequence.

Let us consider now $f_{|S|}$. For each $X, Y \in Dom(f_{|S|})$, either $X = Y$ or $f_{|S|}(X) \sqcap_a f_{|S|}(Y) = \bot_a$. Since $f_{|S|}(X) = \bigsqcup_a X$, $Dom(f_{|S|})$ is a disjoint partition of S. This, together with Lemma 6.24, means that $Dom(f_{|S|})$ is the least disjoint partition of S.

The least disjoint partition is clearly unique. Suppose that there exist two distinct least disjoint partitions $\mathcal{X}$ and $\mathcal{Y}$. From the minimality of $\mathcal{X}$ it follows that for each $X \in \mathcal{X}$ there exists $Y \in \mathcal{Y}$ such that $X \subseteq Y$. Since $\mathcal{X}$ and $\mathcal{Y}$ are distinct, for some $X_0 \in \mathcal{X}$ we have $Y_0 \in \mathcal{Y}$ such that $X_0 \subsetneq Y_0$. But, from the minimality of $\mathcal{Y}$, $Y_0 \subseteq X_1 \in \mathcal{X}$. This cannot happen (since $\mathcal{X}$ is a partition). This completes the proof that a least disjoint partition is unique. □

6.5.3 *Proof of Theorem 6.7*

As always, let us first restate the theorem:

Theorem (6.7; recalled). *Set $\mathcal{D}$ together with the meet and join operators given by* (6.1) *and* (6.6) *forms a lattice.*

Proof. We start with an auxiliary lemma:

Lemma 6.25. *Let $A \subseteq B$ be two finite subsets of $\mathcal{L}$. The least disjoint partition $\mathcal{Y}$ of the superset B preserves the least disjoint partition $\mathcal{X}$ of A, i.e.*

$$\forall_{X \in \mathcal{X}} \exists_{Y \in \mathcal{Y}} X \subseteq Y \tag{6.26}$$

Proof. By contradiction: assume that there exists $X \in \mathcal{X}$ which is not a subset of some $Y \in \mathcal{Y}$. Let $\mathcal{Z}$ be obtained from $\mathcal{Y}$ by removing all elements from $B \setminus A$. $\mathcal{Z}$ is a disjoint partition of A, but X is not a subset of any $Z \in \mathcal{Z}$. This would mean that $\mathcal{X}$ is not the least disjoint partition of A. Contradiction. □

We proceed now with the proof of Theorem 6.7. The commutativity of $\sqcap_d$ and $\sqcup_d$ is straightforward. The associativity of the meet follows directly from the associativity of $\sqcap_k$.

We focus now on the associativity of the join. We have to show that for any a, b, c

$$\mathrm{dNorm}(\mathrm{dNorm}(a \cup b) \cup c) = \mathrm{dNorm}(a \cup \mathrm{dNorm}(b \cup c)) .$$

We start by showing that

$$\mathrm{dNorm}(\mathrm{dNorm}(a \cup b) \cup c) = \mathrm{dNorm}(a \cup b \cup c) . \tag{6.27}$$

Let $\mathcal{X}, \mathcal{Y}$ denote the least disjoint partitions of keys from $a \cup b$ and $a \cup b \cup c$ respectively. Let $\mathcal{Z}$ be obtained from $\mathcal{Y}$ by replacing in each $Y_j \in \mathcal{Y}$ each key i from $a \cup b$ with the corresponding key from $\mathrm{dNorm}(a \cup b)$, i.e. with $\bigsqcup_k X$, where $i \in X$ for some $X \in \mathcal{X}$. Thanks to (6.26), $X \subseteq Y_j$, hence $\bigsqcup_k Z_j = \bigsqcup_k Y_j$ and $\mathcal{Z}$ is a correct disjoint partition of $\mathrm{dNorm}(a \cup b) \cup c$.

Each disjoint partition $\mathcal{W}$ of $\mathrm{dNorm}(a \cup b) \cup c$ can be obtained from some disjoint partition of $a \cup b \cup c$ in the same way as we have defined $\mathcal{Z}$. Since $\mathcal{Z}$ was obtained from the least partition, it must be preserved by all $\mathcal{W}$, hence it is a least disjoint partition of $\mathrm{dNorm}(a \cup b) \cup c$. From the definition of dNorm and the fact that $\bigsqcup_k Z = \bigsqcup_k Y$ for corresponding Z and Y, we immediately get (6.27).

The same arguments can be used to show the equality $\mathrm{dNorm}(a \cup \mathrm{dNorm}(b \cup c)) = \mathrm{dNorm}(a \cup b \cup c)$. These two facts together end the proof of the associativity of the join.

It remains to prove the absorption laws. Let us start with showing that $a \sqcap_d (a \sqcup_d b) = a$. Consider any abstract segment $(i, v) \in a$. There exists an abstract segment $(j, w) \in a \sqcup_d b$ such that $i \sqcap_k j = i$ and $v \sqcap_v w = v$, hence $(i, v) \in a \sqcap_a (a \sqcup_a b)$. Moreover, for any other $(j', w') \in a \sqcup_d b$, $i \sqcap_k j' = \bot_k$, therefore if $(i', v') \notin a$, then $(i', v') \notin a \sqcap_d (a \sqcup_d b)$.

We focus now on the absorption law for $\sqcup_d$, i.e. $a \sqcup_d (a \sqcap_d b) = a$. For any $(i, v) \in a$, let $(j_1, w_1), \ldots, (j_k, w_k) \in a \sqcap_d b$ denote elements, for which $i \sqcap_k j_l \neq \bot_k$ (note that in this means $i \sqcap_k j_l = j_l$). All $i, j_1, \ldots, j_k$ must belong to the same set in the least disjoint partition of the keys from the union $a \cup (a \sqcap_d b)$. Note that $s_i = i \sqcup_k j_1 \cdots \sqcup_k j_k$ is equal to i. This means that $s_i \sqcap_k s_{i'} = \bot_k$ for $i \neq i'$ and finally gives us $(i, v) \in a \sqcup_d (a \sqcap_d b)$. Note also that (by the definition of $\sqcap_d$) for each $(j, w) \in a \sqcap_d b$ there exists $(i, v) \in a$ such that $j \sqcap_k i = j$. This means that in the discussion above we have considered all elements from $a \sqcap_d b$, hence no more elements can belong to $a \sqcup_d (a \sqcap_d b)$.

This ends the proof that $\mathcal{D}$ together with $\sqcap_d$ and $\sqcup_d$ forms a lattice structure. $\square$

6.5.4 *Proof of Theorem 6.9*

Let us restate the theorem:

Theorem (6.9; recalled)**.** *The operator ∇_d given by:*

$$\bot_d \nabla_d a \triangleq a \qquad a \nabla_d \bot_d \triangleq a \qquad a \nabla_d b \triangleq \mathrm{dNorm}(a \tilde{\nabla} b),$$

where $\tilde{\nabla}$ is defined as:

$$\begin{aligned} a \tilde{\nabla} b \triangleq\ & \{(k \nabla_k l, v \nabla_v w) \mid (k, v) \in a, (l, w) \in b, k \sqcap_k l \neq \bot_k\} \\ & \cup \{(k, v) \mid (k, v) \in a, \forall_{(l,w) \in b}\ k \sqcap_k l = \bot_k\} \\ & \cup \{(\top_k, w) \mid (l, w) \in b, \forall_{(k,v) \in a}\ k \sqcap_k l = \bot_k\}\ . \end{aligned}$$

is a widening operator.

Proof. Recall that we say that ∇_d is a widening if and only if

1. for all $a, b \in \mathcal{D}$, $a \sqcup_d b \sqsubseteq_d a \nabla_d b$
2. for every infinite sequence $c_0, c_1, \ldots$, sequence $s_0, s_1, \ldots$ defined as

$$\begin{cases} s_0 = c_0 \\ s_i = s_{i-1} \nabla_d c_i & \text{for } i > 0 \end{cases}$$

is not strictly increasing.

We start with showing that our widening operator always over-approximates the join. First note that each pair $(i, v) \in a \cup b$ is replaced in $a \tilde{\nabla} b$ by some greater pair (i', v'), i.e. $i \sqcap_k i' = i$ and $v \sqcap_v v' = v$. Now, if some keys $i_1, i_2, \ldots, i_k$ belong to the same set in the least disjoint partition of keys from $a \cup b$, then $i'_1, i'_2, \ldots, i'_k$ must be in one set in the partition of keys from $a \tilde{\nabla} b$.

The condition $a \sqcup_d b \sqsubseteq_d a \nabla_d b$ can be rewritten as $(a \sqcup_d b) \sqcap_d (a \nabla_d b) = a \sqcup_d b$. Consider any segment $(i_1 \sqcup_k i_2 \sqcup_k \cdots \sqcup_k i_k, v_1 \sqcup_v v_2 \sqcup_v \cdots \sqcup_v v_k) \in a \sqcup_d b$. First we focus on the keys. Using the observation above, we conclude that in $a \nabla_d b$ there is a key $i'_1 \sqcup_k i'_2 \sqcup_k \cdots \sqcup_k i'_k \sqcup_k j_1 \cdots \sqcup_k j_m$ (as $i'_1, \ldots, i'_k$ must be in the same set in the partition). We need to show that

$$(i_1 \sqcup_k i_2 \sqcup_k \cdots \sqcup_k i_k) \sqcap_k (i'_1 \sqcup_k i'_2 \sqcup_k \cdots \sqcup_k i'_k \sqcup_k j_1 \cdots \sqcup_k j_m) \\ = i_1 \sqcup_k i_2 \sqcup_k \cdots \sqcup_k i_k$$

but this is clear, since $i_l \sqcap_k i'_l = i_l$. Identical reasoning can be repeated for the values. This means that each pair $(i, v) \in a \sqcup_d b$ belongs also to $(a \sqcup_d b) \sqcap_d (a \nabla_d b)$, which completes the proof that the widening always over-approximates the join.

Now we focus on the finite sequence property. If $s_1 = \bot_d$, then the sequence is already stabilised. If $s_1 \neq \bot_d$, we proceed with the proof by contradiction.

Note that the cardinality of the sequence elements may never increase, i.e. $|s_{i+1}| \leqslant |s_i|$. To see this, consider the three parts of the definition of $a \tilde{\nabla} b$. Only the last part potentially adds a new segment $(\top_k, w)$. However, if it was added, the normalisation dNorm would smash everything to just one segment (as $\top_k$ overlaps with all other keys).

Assume now that the sequence $s_0, s_1, \ldots$ is strictly increasing. This means that there must exist an infinite sub-sequence starting at some s_i, such that all elements have the same cardinality: $|s_i| = |s_{i+1}| = |s_{i+2}| = \ldots$. As $|s_i|$ is finite, some pair $(k, v) \in s_i$ must be modified infinitely many times.

This pair can be replaced only by (k', v') such that k' and v' over-approximate $k \nabla_k l$ and $v \nabla_v w$ for some pair (l, w). If either of the sequences $k, k', \ldots$ or $v, v', \ldots$ was strictly increasing, then one of the operators ∇_k or ∇_v would not satisfy the finite sequence property. But these are correct widening operators. $\square$

6.5.5 *Proof of Lemma 6.12*

Let us start with recalling the lemma:

Lemma (6.12; recalled)**.** *If the domains* A, K *and* V *have exact meets, i.e.*

- $\gamma_a(a_1) \cap \gamma_a(a_2) = \gamma_a(a_1 \sqcap_a a_2)$,
- $\gamma_k(k_1) \cap \gamma_k(k_2) = \gamma_k(k_1 \sqcap_k k_2)$ *and*
- $\gamma_v(v_1) \cap \gamma_v(v_2) = \gamma_v(v_1 \sqcap_v v_2)$

then the meet $\sqcap_c$ *in the domain* C *is also exact:*

$$\gamma_c\big((a_1, d_1, i_1)\big) \cap \gamma_c\big((a_2, d_2, i_2)\big) = \gamma_c\big((a_1, d_1, i_1) \sqcap_c (a_2, d_2, i_2)\big) .$$

Proof. Let us start with the inclusion

$$\gamma_c\big((a_1, d_1, i_1)\big) \cap \gamma_c\big((a_2, d_2, i_2)\big) \subseteq \gamma_c\big((a_1, d_1, i_1) \sqcap_c (a_2, d_2, i_2)\big) .$$

Let $(\rho, \tau) \in \gamma_c\big((a_1, d_1, i_1)\big) \cap \gamma_c\big((a_2, d_2, i_2)\big)$. We will show that $(\rho, \tau) \in \gamma_c\big((a_1, d_1, i_1) \sqcap_c (a_2, d_2, i_2)\big)$. As the meet in A is exact, it is easy to see that $\rho \in \gamma_a(a_1 \sqcap_a a_2)$.

Let $T \in \mathcal{V}ar_c$ and $n \in \mathbb{K}$ be arbitrary container and concrete key, respectively. In the concrete state (ρ, τ), $T[n]$ may be either uninitialised or equal to some $m \in \mathbb{E}$. We present the proof only in the first case.

By the definition of predicate $\mathfrak{I}$, given by (6.10), we have for some $(k_1, \mathsf{True}) \in i_1(T)$ and $\sigma_1 \in \gamma_k(k_1)$

$$\sigma_1|_{\mathcal{V}ar} = \rho, \sigma_1(\mathcal{k}) = n$$

and for some $(k_2, \mathsf{True}) \in i_2(T)$ and $\sigma_2 \in \gamma_k(k_2)$:

$$\sigma_2|_{\mathcal{V}ar} = \rho, \sigma_2(\mathcal{k}) = n .$$

This immediately gives that $\sigma_1 = \sigma_2$. By the properties of the meet $\sqcap_k$ in the domain K, we get $\sigma_1 \in \gamma_k(k_1 \sqcap_k k_2)$. By the definition of the meed $\sqcap_d$ given by (6.1) in the generic dictionary abstraction, we get that $(k_1 \sqcap_k k_2, \mathsf{True}) \in (i_1 \sqcap_d i_2)(T)$, thus $T[n]$ may be uninitialised in $(a_1, d_1, i_1) \sqcap_c (a_2, d_2, i_2)$.

In the same way (but using the predicate $\mathfrak{V}$ given by (6.11) instead of $\mathfrak{I}$), we can show the case when $\tau(T)(n) = m$. This completes the proof that $(\rho, \tau) \in \gamma_c\big((a_1, d_1, i_1) \sqcap_c (a_2, d_2, i_2)\big)$.

We focus now on the opposite inclusion:

$$\gamma_c\big((a_1, d_1, i_1)\big) \cap \gamma_c\big((a_2, d_2, i_2)\big) \supseteq \gamma_c\big((a_1, d_1, i_1) \sqcap_c (a_2, d_2, i_2)\big) .$$

Let us assume that $(\rho, \tau) \in \gamma_c\big((a_1, d_1, i_1) \sqcap_c (a_2, d_2, i_2)\big)$. This means that $\rho \in \gamma_a(a_1 \sqcap_a a_2)$. As $\sqcap_a$ is exact, it follows that $\rho \in \gamma_a(a_1)$ and $\rho \in \gamma_a(a_2)$.

Consider an arbitrary container $T \in \mathcal{V}ar_c$ and an arbitrary concrete key $n \in \mathbb{K}$. We show in detail only the case, when $T[n]$ is uninitialised in (ρ, τ), as the other case is similar.

Using the definition of the predicate $\mathfrak{I}$ we get for some $(k, \mathsf{True}) \in (i_1 \sqcap_d i_2)(T)$ and $\sigma_1 \in \gamma_k(k)$

$$\sigma_1|_{\mathcal{V}ar} = \rho, \sigma_1(\mathcal{k}) = n \; .$$

From the definition of $\sqcap_d$ given by (6.1) it follows that (k, True) was computed as $(k_1 \sqcap_k k_2, \mathsf{True})$, where $(k_1, \mathsf{True}) \in i_1(T)$ and $(k_2, \mathsf{True}) \in i_2(T)$. This means that $\sigma \in \gamma_k(k_1 \sqcap_k k_2)$ and, using the exactness of $\sqcap_k$, $\sigma \in \gamma_k(k_1)$ and $\sigma \in \gamma_k(k_2)$, hence, given the scalar valuation ρ, $T[n]$ may be uninitialised both in (a_1, d_1, i_1) and (a_2, d_2, i_2).

In the same way (but using $\mathfrak{V}$ instead of $\mathfrak{I}$), we can show the case when $\tau(T)(n) = m$. This completes the proof that $(\rho, \tau) \in \gamma_c((a_1, d_1, i_1))$ and $(\rho, \tau) \in \gamma_c((a_2, d_2, i_2))$, as well as the whole proof of Lemma 6.12. $\square$

6.5.6 *Proof of Lemma 6.13*

As usual, let us first recall the lemma:

Lemma (6.13; recalled)**.** *If the domains* A, K *and* V *have over-approximating joins, i.e.*

- $\gamma_a(a_1) \cup \gamma_a(a_2) \subseteq \gamma_a(a_1 \sqcup_a a_2)$,
- $\gamma_k(k_1) \cup \gamma_k(k_2) \subseteq \gamma_k(k_1 \sqcup_k k_2)$ *and*
- $\gamma_v(v_1) \cup \gamma_v(v_2) \subseteq \gamma_v(v_1 \sqcup_v v_2)$

then the join $\sqcup_c$ *in the domain* C *is also over-approximating:*

$$\gamma_c\big((a_1, d_1, i_1)\big) \cup \gamma_c\big((a_2, d_2, i_2)\big) \subseteq \gamma_c\big((a_1, d_1, i_1) \sqcup_c (a_2, d_2, i_2)\big) \; .$$

Proof. Let $(\rho, \tau) \in \gamma_c\big((a_1, d_1, i_1)\big) \cup \gamma_c\big((a_2, d_2, i_2)\big)$. Without loss of generality we may assume that $(\rho, \tau) \in \gamma_c\big((a_1, d_1, i_1)\big)$. This means that $\rho \in \gamma_a(a_1)$. As the join $\sqcup_a$ in A is over-approximating, we get that $\rho \in \gamma_a(a_1 \sqcup_a a_2)$.

Now we consider an arbitrary container $T \in \mathcal{V}ar_c$ and an arbitrary concrete key $n \in \mathbb{K}$. In the concrete state (ρ, τ), $T[n]$ may be either uninitialised or equal to some $m \in \mathbb{E}$. We show here only the first case. Using the definition of the predicate $\mathfrak{I}$ we get for some $(k_1, \mathsf{True}) \in (i_1)(T)$ and $\sigma_1 \in \gamma_{\mathsf{k}}(k_1)$

$$\sigma_1|_{\mathcal{V}ar} = \rho, \sigma_1(\mathcal{k}) = n\ .$$

From the definition of the join $\sqcup_{\mathsf{d}}$ given by (6.6), it follows that there exists $(k, \mathsf{True}) \in (i_1 \sqcup_{\mathsf{d}} i_2)(T)$ such that $k = k_1 \sqcup_{\mathsf{k}} l_1 \sqcup_{\mathsf{k}} \ldots \sqcup_{\mathsf{k}} l_m$ for some $m \geqslant 0$ and $(l_j, \mathsf{True}) \in i_1(T) \cup i_2(T)$. As $\sigma_1 \in \gamma_{\mathsf{k}}(k_1)$ and $\sqcup_{\mathsf{k}}$ is over-approximating, we get $\sigma_1 \in \gamma_{\mathsf{k}}(k)$. This means that the predicate $\mathfrak{I}(\rho, T, n, i_1 \sqcup_{\mathsf{d}} i_2)$ holds.

In the same way we justify the case, when $\tau(T)(n) = m$. As T and n were chosen arbitrarily, this ends the proof that $(\rho, \tau) \in \gamma_{\mathsf{c}}\big(a_1 \sqcup_{\mathsf{a}} a_2, d_1 \sqcup_{\mathsf{d}} d_2 i_1 \sqcup_{\mathsf{d}} i_2)\big) = \gamma_{\mathsf{c}}\big((a_1, d_1, i_1) \sqcup_{\mathsf{c}} (a_2, d_2, i_2)\big)$. $\square$

7 SUMMARY

In this thesis we have described our contribution to the static analysis by abstract interpretation of programs that manipulate over numerical and container variables.

The first contribution is a design of two new relational numerical domains. The domain of weighted hexagons (Chapter 3) is capable of representing systems of hexagonal constraints of the form $x \leqslant a \cdot y$, where x and y are program variables and a denotes a non-negative constant. The domain elements have been formalised as pairs of functions that represent the smallest and the largest, respectively, coefficient among all constraints $x \leqslant a \cdot y$ in the system. We have also shown that a domain element can be represented using a pair of graphs, and thus it can be efficiently encoded using adjacency matrices. We have developed a cubic time satisfiability checking and normalisation algorithm that infers the tightest constraints entailed by the given weighted hexagon.

We have defined (and formally proved the correctness of) all domain operations, including meet, join, widening and satisfiability test. We have also provided the transfer function for a simple programming language.

The second domain proposed in this thesis is the domain of strict weighted hexagons (Chapter 4). It is more expressive than the weighted hexagons, as it can maintain both strict and non-strict hexagonal constraints, such as $x \leqslant a \cdot y$ and $u < b \cdot v$, where x, y, u and v are program variables and a, b denote non-negative constants. The additional expressiveness of the domain is achieved without increasing the computational complexity. In case of both standard and strict weighted hexagons, the most complicated operation is the normalisation, which works in a cubic time (with respect to the number of variables). Elements of both these domains can be encoded in a quadratic memory. The domain of strict weighted hexagons lies between pentagons (Section 2.3) and TVPI (Section 2.4) in terms of expressiveness and efficiency.

The major contribution of this thesis is an abstract domain for analysis of programs that use scalar variables together with dictionaries or arrays of scalar values (Chapter 6). The domain is fully generic and can be parametrised with abstractions of dictionary keys, dictionary values and scalar variables. It is also relational, i.e. it can automatically discover relationships between scalars and dictionary keys and values.

Domain elements are constructed in a way that makes it possible to simultaneously find the over-approximation of the set of possible values stored in a dictionary at any key and to determine the set of potentially uninitialised dictionary elements. The domain is equipped with strong and weak dictionary updates, making the analysis as precise as possible.

Our technique can be applied for instance to analysis of arrays and dictionaries. As it is parametrised by the abstractions of keys and dictionary values, it is not limited to analysis of containers containing only numerical values. We have presented examples in which the domain is used to analyse numerical arrays (both in relational and non-relational manner) and string-keyed dictionaries. For purposes of the last example, we have also developed a simple domain for analysis of string variables, with domain elements represented using regular expressions and an automata-based widening.

To the best of our knowledge, this is the first abstract domain that can be used to model arbitrary dictionaries.

As a part of the work on the dictionary abstraction, we have developed a simple prototype of an abstract interpreter for Java source code. The goal of the experiment was to roughly check the performance impact of our approach applied to array analysis. We did not perform any analysis of string-keyed dictionaries.

The prototype works on inter-procedural level. It does not analyse method calls. Instead, we assume that each method may return an arbitrary value and that it can modify all variables available in this call.

In the experiment we have instantiated our technique by fixing the abstractions of scalars, dictionary keys and values as the domain of pentagons.

We have launched the analyser on various open-source projects, including image processing application (ImageJ), ssh client for mobile devices (MidpSSH), Apache Commons Math library and the Oracle Berkeley DB database implementation. In the first stage of the experiment (denoted in Table 7.1 as "off"), the array analysis was turned off, i.e. only analysis of numeric variables in the domain of pentagons was performed. In the second phase, we switched the array analysis on and checked the number of detected non-trivial array invariants at the

PROJECT	SIZE	# METH.	OFF	ON	# INV.
ImageJ	84	4 372	2:08	2:29	265
Apache Commons Math	42	2 896	1:53	1:57	176
MidpSSH	12	561	0:10	0:11	56
Berkeley DB	103	6 196	5:08	5:12	45

Table 7.1: Array analysis statistics on open-source projects

method exit points. We treat an array invariant as non-trivial, if it is more precise than "arbitrary value may be at arbitrary index".

For each project, we report in Table 7.1 its size (in Kilo Lines Of Code), the number of methods, execution times (in minutes) of the analyser when running with and without array analysis as well as the number of detected array invariants.

The measured slowdown turned out to be negligible (about 5%). The results of our experiment are similar to those reported for Fun-Array [17], in terms of both performance impact and number of new invariants.

The partition procedure analysed in Example 6.22, together with the computed invariant, was found (in a slightly more complicated form) in the Apache Commons Math project.

BIBLIOGRAPHY

[1] A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[2] M. Anderson, M. Barnett, M. Fähndrich, B. Grunkemeyer, K. King, and F. Logozzo. *Code Contracts User Manual*. Microsoft Cooperation, 2009.

[3] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. *SIGPLAN Not.*, 24(7):41–53, 1989.

[4] G. Barthe, L. Beringer, P. Crégut, B. Grégoire, M. Hofmann, P. Müller, E. Poll, G. Puebla, I. Stark, and E. Vétillard. Mobius: mobility, ubiquity, security objectives and progress report. In *Proceedings of the 2nd International Conference on Trustworthy Global Computing*, volume 4912 of *LNCS*, pages 10–29, Berlin, Heidelberg, 2007. Springer-Verlag.

[5] N. Blanc, A. Groce, and D. Kroening. Verifying c++ with stl containers via predicate abstraction. In *22nd IEEE/ACM International Conference on Automated Software Engineering, ASE'07*, pages 521–524. ACM, 2007.

[6] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In T. A. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation*, pages 85–108. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[7] O. Bouissou, E. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, E. Goubault, D. Lesens, L. Mauborgne, A. Mine, S. Putot, and X. Rival. Space Software Validation using Abstract Interpretation. In *Int. Space System Engineering Conference, Data Systems in Aerospace DASIA'09*, 2009.

[8] N. Chernikova. Algorithm for discovering the set of all the solutions of a linear programming problem. *U.S.S.R Computational Mathematics and Mathematical Physics*, pages 282–293, 1968.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[10] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277:47–103, April 2002.

[11] P. Cousot. Proving the absence of run-time errors in safety-critical avionics code. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 7–9, New York, NY, USA, 2007. ACM.

[12] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL'77, pages 238–252, New York, NY, USA, 1977. ACM.

[13] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA*, pages 237–277. North-Holland, 1977.

[14] P. Cousot and R. Cousot. Constructive versions of tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82:43–57, 1979.

[15] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[16] P. Cousot and R. Cousot. 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*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295, London, UK, UK, 1992. Springer-Verlag.

[17] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 105–118, New York, NY, USA, 2011. ACM.

[18] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[19] A. Deutsch. Semantic models and abstract interpretation techniques for inductive data structures and pointers. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '95, pages 226–229, New York, NY, USA, 1995. ACM.

[20] E. W. Dijkstra. The humble programmer. *Commun. ACM*, 15:859–866, October 1972.

[21] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *Proceedings of the 19th European Symposium on Programming*, volume 6012 of *Lecture Notes in Computer Science*, pages 246–266. Springer, 2010.

[22] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 187–200, New York, NY, USA, 2011. ACM.

[23] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software*, volume 6528 of *Lecture Notes in Computer Science*, pages 10–30, Berlin, Heidelberg, 2011. Springer-Verlag.

[24] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 191–202, New York, NY, USA, 2002. ACM.

[25] R. W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[26] J. Fulara. Generic abstraction of dictionaries and arrays. *Electron. Notes Theor. Comput. Sci.*, to appear, 2012.

[27] J. Fulara, K. Durnoga, K. Jakubczyk, and A. Schubert. Relational abstract domain of weighted hexagons. *Electron. Notes Theor. Comput. Sci.*, 267:59–72, October 2010.

[28] J. Fulara and K. Jakubczyk. Practically applicable formal methods. In *Proceedings of the 36th Conference on Current Trends in Theory and Practice of Computer Science*, volume 5901 of *Lecture Notes in Computer Science*, pages 407–418, Berlin, Heidelberg, 2010. Springer-Verlag.

[29] J. Fulara and T. Sznuk. Fluid updates in arbitrary abstract domains. *technical report*.

[30] R. Giacobazzi and F. Ranzato. Optimal domains for disjunctive abstract interpretation. *Sci. Comput. Program.*, 32:177–210, September 1998.

[31] D. Gopan, F. DiMaio, N. Dor, T. W. Reps, and S. Sagiv. Numeric domains with summarized dimensions. In *TACAS'04 : Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pages 512–529, 2004.

[32] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 338–350, New York, NY, USA, 2005. ACM.

[33] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, London, UK, 1997. Springer-Verlag.

[34] R. L. Graham. An efficient algorithm for determining the convex hull of a planar set. In *Information Processing Letters*, pages 132–133, 1972.

[35] P. Granger. 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*, volume 493 of *Lecture Notes in Computer Science*, pages 169–192, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[36] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 235–246, New York, NY, USA, 2008. ACM.

[37] A. Gurfinkel and S. Chaki. Boxes: a symbolic abstract domain of boxes. In *Proceedings of the 17th International Conference on Static analysis*, volume 6337 of *Lecture Notes in Computer Science*, pages 287–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[38] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 339–348, New York, NY, USA, 2008. ACM.

[39] T. A. Henzinger, T. Hottelier, L. Kovacs, and A. Voronkov. Invariant and type inference for matrices. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5944 of *Lecture Notes in Computer Science*, pages 163–179. Springer-Verlag, 2010.

[40] J. E. Hopcroft and J. D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1990.

[41] K. Jakubczyk. Sweeping in abstract interpretation. *Electron. Notes Theor. Comput. Sci.*, 2011, to appear.

[42] N. D. Jones. Combining abstract interpretation and partial evaluation (brief overview). In *Proceedings of the 4th International Symposium on Static Analysis*, volume 1302 of *Lecture Notes in Computer Science*, pages 396–405, London, UK, 1997. Springer-Verlag.

[43] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[44] S.-W. Kim and K.-M. Choe. String analysis as an abstract interpretation. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 294–308, Berlin, Heidelberg, 2011.

[45] S. C. Kleene. *Representation of Events in Nerve Nets and Finite Automata*, pages 3–42. Princeton University Press, Princeton, N.J., 1956.

[46] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software*

Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, volume 5503 of *Lecture Notes in Computer Science*, pages 470–485, Berlin, Heidelberg, 2009. Springer-Verlag.

[47] J. C. Lagarias. The computational complexity of simultaneous diophantine approximation problems. *SIAM J. Comput.*, 14:196–209, February 1985.

[48] F. Logozzo. Separate compositional analysis of class-based object-oriented languages. In *Proceedings of the 10th International Conference on Algebraic Methodology and Software Technology*, AMAST'04, pages 332–346. Springer-Verlag, 2004.

[49] F. Logozzo and M. Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 184–188, New York, NY, USA, 2008. ACM.

[50] M. Marron, D. Stefanovic, M. Hermenegildo, and D. Kapur. Heap analysis in the presence of collection libraries. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE'07, pages 31–36, New York, NY, USA, 2007. ACM.

[51] R. McNaughton and H. Yamada. Regular Expressions and State Graphs for Automata. *IEEE Transactions on Electronic Computers*, EC-9(1):39–47, Mar. 1960.

[52] A. Melton, D. A. Schmidt, and G. E. Strecker. Galois connections and computer science applications. In *Proceedings of a Tutorial and Workshop on Category Theory and Computer Programming, CTCP'86*, volume 240 of *Lecture Notes in Computer Science*, pages 299–312, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[53] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Proceedings of the Second Symposium on Programs as Data Objects*, PADO '01, pages 155–172, London, UK, 2001. Springer-Verlag.

[54] A. Miné. The octagon abstract domain. *Higher Order Symbolic Computation*, 19(1):31–100, 2006.

[55] C. G. Nelson. An $n^{\log n}$ algorithm for the two-variable-per-constraint linear programming satisfiability problem. Technical report, Stanford, CA, USA, 1978.

[56] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '02, pages 83–94. ACM, 2002.

[57] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 296–309, New York, NY, USA, 2005. ACM.

[58] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM.

[59] A. Salomaa. *Formal languages*. Academic Press Professional, Inc., San Diego, CA, USA, 1987.

[60] M. N. Seghir, A. Podelski, and T. Wies. Abstraction refinement for quantified array assertions. In *Proceedings of the 16th International Symposium on Static Analysis, SAS'09*, volume 5673 of *Lecture Notes in Computer Science*, pages 3–18, Berlin, Heidelberg, 2009. Springer-Verlag.

[61] A. Simon, A. King, and J. M. Howe. Two variables per linear inequality as an abstract domain. In *Proceedings of the 12th International Conference on Logic Based Program Synthesis and Transformation, LOPSTR'02*, volume 2664 of *Lecture Notes in Computer Science*, pages 71–89, Berlin, Heidelberg, 2003. Springer-Verlag.

[62] M. Sintzoff. Calculating properties of programs by valuations on specific models. *SIGACT News*, (14):203–207, 1972.

[63] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In *Proceedings of the 2nd World Congress on Formal Methods, FM '09*, volume 5850 of *Lecture Notes in Computer Science*, pages 532–546, Berlin, Heidelberg, 2009. Springer-Verlag.

[64] T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Inf. Process. Lett.*, 43:195–199, September 1992.

[65] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[66] J. Yoo, S. Cha, and E. Jee. A verification framework for fbd based software in nuclear power plants. In *Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*, pages 385–392, Washington, DC, USA, 2008. IEEE Computer Society.

[67] F. Yu, T. Bultan, and O. H. Ibarra. Relational string verification using multi-track automata. In *Proceedings of the 15th International Conference on Implementation and Application of Automata*, volume 6482 of *Lecture Notes in Computer Science*, pages 290–299, Berlin, Heidelberg, 2011. Springer-Verlag.