HAL
open science

Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs

Caterina Urban

To cite this version:

Caterina Urban. Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs. Computer Science [cs]. École Normale Supérieure, 2015. English. NNT : . tel-01176641v1

HAL Id: tel-01176641

https://theses.hal.science/tel-01176641v1

Submitted on 15 Jul 2015 (v1), last revised 18 Apr 2018 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT

En vue de l'obtention du grade de

DOCTEUR
DE L'ÉCOLE NORMALE SUPÉRIEURE

École doctorale numéro 386:
Sciences mathématiques de Paris Centre

Discipline ou spécialité : **INFORMATIQUE**

Présentée et soutenue par :

Caterina URBAN

le 9 juillet 2015

Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs

Analyse Statique par Interprétation Abstraite de Propriétés Temporelles Fonctionnelles des Programmes

---

Unité de recherche	UMR8548
Thése dirigée par	Antoine MINÉ, *CNRS & École Normale Supérieure, France*
	†Radhia COUSOT, *CNRS & École Normale Supérieure, France*
Membres du jury	
Président :	Nicolas HALBWACHS, *Verimag, France*
Rapporteurs :	Manuel HERMENEGILDO, *IMDEA Software Institute, Espagne*
	Andreas PODELSKI, *University of Freiburg, Allemagne*
Examinateurs :	Patrick COUSOT, *New York University, USA*
	Mooly SAGIV, *Tel Aviv University, Israël*
Directeur de Thèse :	Antoine MINÉ, *CNRS & École Normale Supérieure, France*

Numéro identifiant de la Thèse : 68165

To Luca,
shekh ma shieraki anni

To Radhia,
who trusted me with this work
and changed my life

Abstract

The overall aim of this thesis is the development of mathematically sound and practically efficient methods for automatically proving the correctness of computer software. More specifically, this thesis is grounded in the theory of Abstract Interpretation, a powerful mathematical framework for approximating the behavior of programs. In particular, this thesis focuses on proving program liveness properties, which represent requirements that must be eventually or repeatedly realized during program execution.

Program termination is the most prominent liveness property. This thesis designs new program approximations, in order to automatically infer sufficient preconditions for program termination and synthesize so called piecewise-defined ranking functions, which provide upper bounds on the waiting time before termination. The approximations are parametric in the choice between the expressivity and the cost of the underlying approximations, which maintain information about the set of possible values of the program variables along with the possible numerical relationships between them.

This thesis also contributes an abstract interpretation framework for proving liveness properties, which comes as a generalization of the framework proposed for termination. In particular, the framework is dedicated to liveness properties expressed in temporal logic, which are used to ensure that some desirable event happens once or infinitely many times during program execution. As for program termination, piecewise-defined ranking functions are used to infer sufficient preconditions for these properties, and to provide upper bounds on the waiting time before a desirable event.

The results presented in this thesis have been implemented into a prototype analyzer. Experimental results show that it performs well on a wide variety of benchmarks, it is competitive with the state of the art, and is able to analyze programs that are out of the reach of existing methods.

Résumé

L'objectif général de cette thèse est le développement de méthodes mathématiques correctes et efficaces en pratique pour prouver automatiquement la correction de logiciels. Plus précisément, cette thèse est fondée sur la théorie de l'Interprétation Abstraite, un cadre mathématique puissant pour l'approximation du comportement des programmes. En particulier, cette thèse se concentre sur la preuve des propriétés de vivacité des programmes, qui représentent des conditions qui doivent être réalisés ultimement ou de manière répétée pendant l'exécution du programme.

La terminaison des programmes est la propriété de vivacité la plus fréquement considérée. Cette thèse conçoit des nouvelles approximations, afin de déduire automatiquement des conditions suffisantes pour la terminaison des programmes et synthétiser des fonctions de rang définies par morceaux, qui fournissent des bornes supérieures sur le temps d'attente avant la terminaison. Les approximations sont paramétriques dans le choix entre l'expressivité et le coût des approximations sous-jacentes, qui maintiennent des informations sur l'ensemble des valeurs possibles des variables du programme ainsi que les relations numériques possibles entre elles.

Cette thèse développe également un cadre d'interprétation abstraite pour prouver des propriétés de vivacité, qui vient comme une généralisation du cadre proposé pour la terminaison. En particulier, le cadre est dédié à des propriétés de vivacité exprimées dans la logique temporelle, qui sont utilisées pour s'assurer qu'un événement souhaitable se produit une fois ou une infinité de fois au cours de l'exécution du programme. Comme pour la terminaison, des fonctions de rang définies par morceaux sont utilisées pour déduire des préconditions suffisantes pour ces propriétés, et fournir des bornes supérieures sur le temps d'attente avant un événement souhaitable.

Les résultats présentés dans cette thèse ont été mis en œuvre dans un prototype d'analyseur. Les résultats expérimentaux montrent qu'il donne de bons résultats sur une grande variété de programmes, il est compétitif avec l'état de l'art, et il est capable d'analyser des programmes qui sont hors de la portée des méthodes existantes.

Contents

Abstract **i**

I Introduction **1**

1 Introduction **3**
1.1 Software Verification 3
1.1.1 Testing 3
1.1.2 Formal Methods 4
1.2 Program Properties 5
1.2.1 Safety Properties 5
1.2.2 Liveness Properties 6
1.3 Termination 6
1.4 Liveness Properties 7

II Safety **9**

2 Abstract Interpretation **11**
2.1 Basic Notions and Notations 11
2.2 Abstract Interpretation 18
2.2.1 Maximal Trace Semantics 19
2.2.2 Galois Connections 22
2.2.3 Widening and Narrowing 27

3 A Small Imperative Language **31**
3.1 A Small Imperative Language 31
3.2 Maximal Trace Semantics 32
3.3 Invariance Semantics 40
3.4 Numerical Abstract Domains 43
3.4.1 Intervals Abstract Domain 46
3.4.2 Polyhedra Abstract Domain 47
3.4.3 Octagons Abstract Domain 47

III Termination **49**

4 An Abstract Interpretation Framework for Termination **51**
4.1 Ranking Functions 51
4.2 Termination Semantics 53
4.2.1 Termination Trace Semantics 54
4.2.2 Termination Semantics 58
4.3 Denotational Definite Termination Semantics 64

5 Piecewise-Defined Ranking Functions **69**
5.1 Piecewise-Defined Ranking Functions 70
5.2 Decision Trees Abstract Domain 73
5.2.1 Decision Trees 74
5.2.2 Binary Operators 79
5.2.3 Unary Operators 87
5.2.4 Widening 97
5.3 Abstract Definite Termination Semantics 108
5.4 Related Work 113

6 Ordinal-Valued Ranking Functions **115**
6.1 Ordinal-Valued Ranking Functions 115
6.2 Ordinal Arithmetic 117
6.3 Decision Trees Abstract Domain 119
6.3.1 Decision Trees 119
6.3.2 Binary Operators 121
6.3.3 Unary Operators 123
6.3.4 Widening 125
6.4 Abstract Definite Termination Semantics 126
6.5 Related Work 131

7 Recursive Programs **133**
7.1 A Small Procedural Language 133
7.2 Maximal Trace Semantics 134
7.3 Definite Termination Semantics 140
7.3.1 Definite Termination Semantics 140
7.3.2 Abstract Definite Termination Semantics 142

8 Implementation **149**
8.1 FuncTion 149
8.2 Experimental Evaluation 150

IV Liveness **159**

9 A Hierarchy of Temporal Properties **161**
9.1 A Hierarchy of Temporal Properties 162
9.1.1 Safety Properties 163
9.1.2 Guarantee Properties 165
9.1.3 Obligation Properties 166
9.1.4 Recurrence Properties 166
9.1.5 Persistence Properties 167
9.1.6 Reactivity Properties 168
9.2 Guarantee Semantics 169
9.2.1 Guarantee Semantics 169
9.2.2 Denotational Guarantee Semantics 172
9.2.3 Abstract Guarantee Semantics 177
9.3 Recurrence Semantics 182
9.3.1 Recurrence Semantics 182
9.3.2 Denotational Recurrence Semantics 185
9.3.3 Abstract Recurrence Semantics 190
9.4 Implementation 194
9.5 Related Work 198

V Conclusion **199**

10 Future Directions **201**

Bibliography **214**

A Proofs **215**
A.1 Missing Proofs from Chapter 1 215
A.2 Missing Proofs from Chapter 4 217
A.3 Missing Proofs from Chapter 5 217
A.4 Missing Proofs from Chapter 6 222
A.5 Missing Proofs from Chapter 7 223
A.6 Missing Proofs from Chapter 9 225

I

Introduction

1 Introduction

1.1 Software Verification

In the last decades, software took a growing importance into all kinds of systems. As we rely more and more on software, the consequences of a bug are more and more dramatic, causing great financial and even human losses. A notorious example is the spectacular Ariane 5 failure[1] caused by an integer overflow, which resulted in more than $370 million loss. More recent examples are the Microsoft Zune Z2K bug[2] and the Microsoft Azure Storage service interruption[3] both due to non-termination, the Toyota unintended acceleration[4] caused by a stack overflow, and the Heartbleed security bug[5].

1.1.1 Testing

The most widespread (and in many cases the only) method used to ensure the quality of software is testing. Many testing methods exist, ranging from black-box and white-box testing to unit and integration testing. They all consist in executing parts or the whole of the program with selected or random inputs in a controlled environment, while monitoring its execution or its output. Achieving an acceptable level of confidence with testing is generally costly and, even then, testing cannot completely eliminate bugs. Therefore, while in some cases it is acceptable to ship potentially erroneous programs and rely on regular updates to correct them, this is not the case for mission critical software which cannot be corrected during missions.

[1] http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf
[2] http://techcrunch.com/2008/12/31/zune-bug-explained-in-detail/
[3] http://azure.microsoft.com/blog/2014/11/19/update-on-azure-storage-service-interruption/
[4] http://www.automotive-spin.it/uploads/12/12W_Bagnara.pdf
[5] http://heartbleed.com

1.1.2 Formal Methods

Formal methods, on the other hand, try to address these problems by providing rigorous mathematical guarantees that a program preserves certain properties. The idea of formally discussing about programs dates back from the early history of computer science: program proofs and invariants are attributed to Robert W. Floyd [Flo67] and Tony Hoare [Hoa69] in the late 1960s, but may be latent in the work of Alan Turing in the late 1940s [Tur49, MJ84]. Current methods can be classified into three categories [CC10]: *deductive methods*, *model checking*, and *static analysis*.

Formal **deductive methods** employ *proof assistants* such as Coq [BC04], or *theorem provers* such as PVS [ORS92] to prove the correctness of programs. These methods rely on the user to provide the inductive arguments needed for the proof, and sometimes to interactively direct the proof itself.

Formal methods based on **model checking** [CE81, QS82] explore exhaustively and automatically finite models of programs so as to determine whether undesirable error states are accessible. Various successful model checking algorithms based on *SAT-solving*, such as IC3 [Bra11], have been developed to complement the traditional model checking based on *Binary Decision Diagrams* [Bry86]. A major difficulty of this approach is to synthesize the model from the program and the property to check. In particular, the size of the model is critical for the checking phase to be practical.

Formal **static analysis** methods analyze directly and without user intervention the program source code at some level of abstraction. Due to decidability and efficiency concerns, the abstraction is incomplete and can result in *false alarms* or false positives (that is, correct programs may be reported as incorrect) but never false negatives. The programs that are reported as correct are indeed correct despite the approximation. The most prominent static analysis methods are based on *Abstract Interpretation* [CC77].

Abstract Interpretation. Abstract Interpretation [CC77] is a general theory for approximating the behavior of programs, developed by Patrick Cousot and Radhia Cousot in the late 1970s, as a unifying framework for static program analysis. It stems from the observation that, to reason about a particular program property, it is not necessary to consider all aspects and details of the program behavior. In fact, reasoning is facilitated by the design of a well-adapted semantics, abstracting away from irrelevant details. Therefore, there

is no universal *general-purpose* program semantics but rather a wide variety of *special-purpose* program semantics, each one of them dedicated to a particular class of program properties and reduced to essentials in order to ignore irrelevant details about program executions.

In the past decade, abstract interpretation-based static analyzers began to have an impact in real-world software development. This is the case, for instance, of the static analyzer Astrée [BCC+10], which is used daily by industrial end-users in order to prove the absence of run-time errors in embedded synchronous C programs. In particular, by carefully designing the abstractions used for the analysis, Astrée has been successfully used to prove, with zero false alarm, the absence of run-time errors in the primary flight control software of the fly-by-wire systems of the Airbus A340 and A380 airplanes.

We provide a formal introduction to Abstract Interpretation in Chapter 2 and we recall the main results used in this thesis, which are later illustrated on a small idealized programming language in Chapter 3.

1.2 Program Properties

Leslie Lamport, in the late 1970s, suggested a classification of program properties into the classes of *safety* and *liveness* properties [Lam77]. Each class encompasses properties of similar character, and every program property can be expressed as the intersection of a safety and a liveness property.

1.2.1 Safety Properties

The class of **safety properties** is informally characterized as the class of properties stating that "something *bad* never happens", that is, a program never reaches an unacceptable state.

Safety properties represent requirements that should be continuously maintained by the program. Indeed, in order to prove safety properties, an invariance principle [Flo67] can be used. A counterexample to a safety property can always be witnessed by observing finite (prefixes of) program executions.

Examples of safety properties include program partial correctness, which guarantees that all terminating computations produce correct results, and mutual exclusion, which guarantees that no two concurrent processes enter their critical section at the same time.

1.2.2 Liveness Properties

The class of **liveness properties**, on the other hand, is informally characterized as the class of properties stating that "something *good* eventually happens", that is, a program *eventually* reaches a desirable state.

Liveness properties typically represent requirements that need not hold continuously, but whose eventual or repeated realization must be guaranteed. They are usually proved by exhibiting a well-founded argument, often called *ranking function*, which provides a measure of the distance from the realization of the requirement [Tur49, Flo67]. Note that, a counterexample to a liveness property cannot be witnessed by observing finite program executions since these can always be extended in order to satisfy the liveness property.

Examples of liveness properties are program **termination**, which guarantees that all program computations are terminating, and starvation freedom, which guarantees that a process will eventually enter its critical section.

1.3 Termination

The **halting problem**, the problem of determining whether a given program will always finish to run or could potentially execute forever, rose to prominence before the invention of programs or computers, in the era of the *Entscheidungsproblem* posed by David Hilbert: the challenge to formalize all mathematics into logic and use mechanical means to determine the validity of mathematical statements. In hopes of either solving the challenge, or showing it impossible, logicians and mathematicians began to search for possible instances of undecidable problems. The undecidability of the halting problem, proved by Alan Turing [Tur36], is the most famous of those findings. In Appendix A.1, we propose a proof of the undecidability of the halting problem reinterpreted by the linguist Geoffrey K. Pullum.

However, relaxing the halting problem by asking for a sound but not necessarily complete solution, allows us to overcome the barrier of its undecidability. Indeed, in the recent past, termination analysis has benefited from many research advances and powerful termination provers have emerged over the years [BCF13, CPR06, GSKT06, HHLP13, LQC15, etc.].

This thesis stems from [CC12], where Patrick Cousot and Radhia Cousot proposed a unifying point of view on the existing approaches for proving program termination, and introduced the idea of the inference of a ranking function by abstract interpretation. We recall and revise their work in Chapter 4.

Chapter 5 and Chapter 6 are devoted to the construction of new abstractions dedicated to program termination. More precisely, Chapter 5 presents abstractions based on piecewise-defined ranking functions, while Chapter 6 presents abstractions based on ordinal-valued ranking functions. Some of the results described in Chapter 5 and Chapter 6 have been the subject of publications in international workshops and conferences [Urb13a, Urb13b, UM14a, UM14c, UM14b] and are presented here with many extensions. In Chapter 7, we detail how these abstractions can be used for proving termination of recursive programs. The implementation of our prototype static analyzer [Urb15] and the most recent experimental evaluation [DU15] are described in Chapter 8.

1.4 Liveness Properties

For reactive programs, which may never terminate, the spectrum of relevant and useful liveness properties is much richer than the single property of termination. For example, it also includes the guarantee that a certain event occurs *infinitely many times*.

In general, these liveness properties are satisfied only under **fairness** hypotheses, that is, a restriction on some infinite behavior according to eventual occurrence of some events. A common property of these fairness notions is that they all imply that, under certain conditions, each of a number of alternative or competing events occur infinitely often in every infinite behavior of the system considered. Nissim Francez distinguishes three main subclasses, depending on the condition guaranteeing the eventual occurrence [Fra86]: *unconditional fairness* guarantees that for each behavior each event occurs infinitely often without any further condition; *weak fairness* implies that an event will not be indefinitely postponed provided that it remains continuously enabled; and *strong fairness* guarantees eventual occurrence under the condition of being enabled infinitely often, but not necessarily continuously.

In Chapter 9, we generalize the abstract interpretation framework proposed for termination by Patrick Cousot and Radhia Cousot [CC12], to other liveness properties. In particular, we focus on two classes of program properties proposed by Zohar Manna and Amir Pnueli [MP90]: the class of *guarantee properties* informally characterized as the class of properties stating that "something good happens *at least once*", and the class of *recurrence properties* informally characterized as the class of properties stating that "something good happens *infinitely often*". In order to effectively prove these properties

we reuse the abstractions proposed in Chapter 5 and Chapter 6. The results described in Chapter 9 have been published in [UM15].

Note that the guarantee provided by liveness properties that something good will *eventually* happen is not enough for mission critical software, where some desirable event is required to occur within a certain time bound. Indeed, these real-time properties are considered safety properties, and there is a general consensus against the interest of proving liveness properties for mission critical software. However, specifying and verifying real-time properties can be cumbersome, while liveness properties provide a nice approximation, without distracting timing assumptions, of real-time properties. The well-founded measures used to prove these liveness properties can then be mapped back to a real-time duration. In particular, this is the premise of ongoing research work conducted at NASA Ames Research Center where some of the theoretical work presented in this thesis is being used for the verification of avionics software.

We envision further future directions in Chapter 10.

II

Safety

2

Abstract Interpretation

This chapter introduces the notions that are at the foundation of our work and will serve in subsequent chapters. We provide an overview of Abstract Interpretation while defining the terminology and concepts used in this thesis.

Ce chapitre présente les notions à la base de notre travail et qui serviront dans les chapitres suivants. Nous offrons un aperçu de l'Interprétation Abstraite tout en définissant la terminologie et les concepts utilisés dans cette thèse.

2.1 Basic Notions and Notations

In the following, we briefly recall well-known mathematical concepts in order to establish the notation used throughout this thesis.

Sets. A *set* $\mathcal{S}$ is defined as an unordered collection of distinct elements. We write $s \in \mathcal{S}$ (resp. $s \notin \mathcal{S}$) when s is (resp. is not) an element of the set $\mathcal{S}$. A set is expressed in extension when it is uniquely identified by its elements: we write $\{a, b, c\}$ for the set of elements a, b and c. The *empty set* is denoted by $\emptyset$. A set is expressed in comprehension when its elements are specified through a shared property: we write $\{x \in \mathcal{S} \mid P(x)\}$ for the set of elements x of the set $\mathcal{S}$ for which $P(x)$ is true. The cardinality of a set $\mathcal{S}$ is denoted by $|\mathcal{S}|$.

A set $\mathcal{S}$ is a *subset* of a set $\mathcal{S}'$, written $\mathcal{S} \subseteq \mathcal{S}'$, if and only if every element of $\mathcal{S}$ is an element of $\mathcal{S}'$. The empty set is a subset of every set. The *power set* $\mathcal{P}(\mathcal{S})$ of a set $\mathcal{S}$ is the set of all its subsets: $\mathcal{P}(\mathcal{S}) \stackrel{\text{def}}{=} \{S' \mid S' \subseteq S\}$.

The *union* of two sets $\mathcal{A}$ and $\mathcal{B}$, written $\mathcal{A} \cup \mathcal{B}$, is the set of all elements of $\mathcal{A}$ and all elements of $\mathcal{B}$: $\mathcal{A} \cup \mathcal{B} \stackrel{\text{def}}{=} \{x \mid x \in \mathcal{A} \vee x \in \mathcal{B}\}$. More generally, the union of a set of sets $\mathcal{S}$, is denoted by $\bigcup \mathcal{S}$: $\bigcup \mathcal{S} \stackrel{\text{def}}{=} \bigcup_{\mathcal{S}' \in \mathcal{S}} \mathcal{S}' = \{x \mid \exists \mathcal{S}' \in \mathcal{S} : x \in \mathcal{S}'\}$. The *intersection* $\mathcal{A} \cap \mathcal{B}$ of two sets $\mathcal{A}$ and $\mathcal{B}$ is the set of all elements that are elements of both $\mathcal{A}$ and $\mathcal{B}$: $\mathcal{A} \cap \mathcal{B} \stackrel{\text{def}}{=} \{x \mid x \in \mathcal{A} \ \wedge \ x \in \mathcal{B}\}$. More generally,

the intersection of a set of sets $\mathcal{S}$ is denoted by $\bigcap \mathcal{S}$: $\bigcap \mathcal{S} \stackrel{\text{def}}{=} \bigcap_{\mathcal{S}' \in \mathcal{S}} \mathcal{S}' = \{x \mid \forall \mathcal{S}' \in \mathcal{S} : x \in \mathcal{S}'\}$. The *relative complement* of a set $\mathcal{B}$ in a set $\mathcal{A}$, denoted by $\mathcal{A} \setminus \mathcal{B}$, is the set of all elements of $\mathcal{A}$ that are not elements of $\mathcal{B}$: $\mathcal{A} \setminus \mathcal{B} \stackrel{\text{def}}{=} \{x \mid x \in \mathcal{A} \ \wedge \ x \notin \mathcal{B}\}$. When $\mathcal{B} \subseteq \mathcal{A}$ and the set $\mathcal{A}$ is clear from the context, we simply write $\neg\mathcal{B}$ for $\mathcal{A} \setminus \mathcal{B}$ and we call it *complement* of $\mathcal{B}$. The *cartesian product* of two sets $\mathcal{A}$ and $\mathcal{B}$, denoted by $\mathcal{A} \times \mathcal{B}$, is the set of all pairs where the first component is an element of $\mathcal{A}$ and the second component is an element of $\mathcal{B}$: $\mathcal{A} \times \mathcal{B} \stackrel{\text{def}}{=} \{\langle x, y\rangle \mid x \in \mathcal{A} \ \wedge \ y \in \mathcal{B}\}$. More generally, $\mathcal{S}_1 \times \cdots \times \mathcal{S}_n \stackrel{\text{def}}{=} \{\langle x_1, \ldots, x_n\rangle \mid x_1 \in \mathcal{S}_1 \ \wedge \ \cdots \ \wedge \ x_n \in \mathcal{S}_n\}$ and $\mathcal{S}^n \stackrel{\text{def}}{=} \{\langle x_1, \ldots, x_n\rangle \mid x_1 \in \mathcal{S} \ \wedge \ \cdots \ \wedge \ x_n \in \mathcal{S}\}$ is the set of n-tuples of elements of $\mathcal{S}$.

A *covering* of a set $\mathcal{S}$ is a set $\mathcal{P}$ of non-empty subsets of $\mathcal{S}$ such that any element $s \in \mathcal{S}$ belongs to a set in $\mathcal{P}$: $\mathcal{S} = \bigcup \mathcal{P}$. A *partition* of a set $\mathcal{S}$ is a covering $\mathcal{P}$ such that any two sets in $\mathcal{P}$ are disjoint, that is, any element $s \in \mathcal{S}$ belongs to a *unique* set in $\mathcal{P}$: $\forall X, Y \in \mathcal{P} : X \neq Y \Rightarrow X \cap Y = \emptyset$.

Relations. A *binary relation* R between two sets $\mathcal{A}$ and $\mathcal{B}$ is a subset of the cartesian product $\mathcal{A} \times \mathcal{B}$. We often write $x \; R \; y$ for $\langle x, y\rangle \in R$.

The following are some important properties which may hold for a binary relation R over a set $\mathcal{S}$:

$$\forall x \in \mathcal{S} : x \; R \; x \qquad \text{(reflexivity)}$$
$$\forall x \in \mathcal{S} : \neg(x \; R \; x) \qquad \text{(irreflexivity)}$$
$$\forall x, y \in \mathcal{S} : x \; R \; y \Rightarrow y \; R \; x \qquad \text{(symmetry)}$$
$$\forall x, y \in \mathcal{S} : x \; R \; y \wedge y \; R \; x \Rightarrow x = y \qquad \text{(anti-symmetry)}$$
$$\forall x, y, z \in \mathcal{S} : x \; R \; y \wedge y \; R \; z \Rightarrow x \; R \; z \qquad \text{(transitivity)}$$
$$\forall x, y \in \mathcal{S} : x \; R \; y \vee y \; R \; x \qquad \text{(totality)}$$

An *equivalence relation* is a binary relation which is reflexive, symmetric, and transitive. A binary relation which is reflexive (resp. irreflexive), anti-symmetric, and transitive is called a *partial order* (resp. *strict partial order*). A *preorder* is reflexive and transitive, but not necessarily anti-symmetric. A (*strict*) *total order* is a (strict) partial order which is total.

Ordered Sets. A *partially ordered set* or *poset* (resp. *preordered set*) $\langle \mathcal{D}, \sqsubseteq\rangle$ is a set $\mathcal{D}$ equipped with a partial order (resp. preorder) $\sqsubseteq$. A finite partially ordered set $\langle \mathcal{D}, \sqsubseteq\rangle$ can be represented by a *Hasse diagram* such as the one in Figure 2.1: each element $x \in \mathcal{D}$ is uniquely represented by a node of the diagram, and there is an edge from a node $x \in \mathcal{D}$ to a node $y \in \mathcal{D}$ if y covers x,

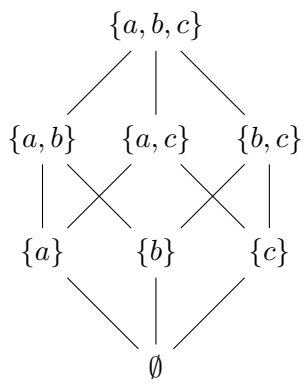


Figure 2.1: Hasse diagrams for the partially ordered set $\langle \mathcal{P}(\{a,b,c\}), \subseteq \rangle$.

that is, $x \leq y$ and there exists no $z \in \mathcal{D}$ such that $x \leq z \leq y$. Hasse diagrams are usually drawn placing the elements higher than the elements they cover.

Let $\langle \mathcal{D}, \sqsubseteq \rangle$ be a partially ordered set. The least element of the poset, when it exists, is denoted by $\bot$: $\forall d \in \mathcal{D} : \bot \sqsubseteq d$. Similarly, the greatest element of the poset, when it exists, is denoted by $\top$: $\forall d \in \mathcal{D} : d \sqsubseteq \top$. Note that any partially ordered set can always be equipped with a least (resp. greatest) element by adding a new element that is smaller (resp. greater) than every other element. Let $\mathcal{X} \subseteq \mathcal{D}$. A *maximal element* of $\mathcal{X}$ is an element $d \in \mathcal{X}$ such that, for each $x \in \mathcal{X}$, $x \sqsubseteq d$. When $\mathcal{X}$ has a unique maximal element, it is called *maximum* and it is denoted by $\max \mathcal{X}$. Dually, a *minimal element* of $\mathcal{X}$ is an element $d \in \mathcal{X}$ such that, for each $x \in \mathcal{X}$, $d \sqsubseteq x$. When $\mathcal{X}$ has a unique minimal element, it is called *minimum* and it is denoted by $\min \mathcal{X}$. An *upper bound* of $\mathcal{X}$ is an element $d \in \mathcal{D}$ (not necessarily belonging to $\mathcal{X}$) such that, for each $x \in \mathcal{X}$, $x \sqsubseteq d$. The *least upper bound* (or *lub*, or *supremum*) of $\mathcal{X}$ is an upper bound $d \in \mathcal{D}$ of $\mathcal{X}$ and such that, for every upper bound $d' \in \mathcal{D}$ of $\mathcal{X}$, $d \sqsubseteq d'$. When it exists, it is unique and denoted by $\bigsqcup \mathcal{X}$ (or $\sup \mathcal{X}$). Dually, a *lower bound* of $\mathcal{X}$ is an element $d \in \mathcal{D}$ such that, for each $x \in \mathcal{X}$, $d \sqsubseteq x$. The *greatest lower bound* (or *glb*, or *infimum*) of $\mathcal{X}$ is a lower bound $d \in \mathcal{D}$ of $\mathcal{X}$ such that, for every lower bound $d' \in \mathcal{D}$ of $\mathcal{X}$, $d' \sqsubseteq d$. When it exists, it is unique and denoted by $\bigsqcap \mathcal{X}$ (or $\inf \mathcal{X}$). Note that, the notion of maximal element, maximum, and least upper bound (resp. minimal element, minimum, and greatest lower bound) are different, as illustrated by the following example.

Example 2.1.1

Let us consider the partially ordered set $\langle \mathcal{P}(\{a,b,c\}), \subseteq \rangle$ represented in Figure 2.1 and the subset $\mathcal{X} \stackrel{\text{def}}{=} \{\{a\},\{b\},\{c\},\{a,b\},\{a,c\},\{b,c\}\}$. The maximal

elements of $\mathcal{X}$ are $\{a, b\}$, $\{a, c\}$, and $\{b, c\}$. Thus the maximum $\max \mathcal{X}$ of $\mathcal{X}$ does not exist, while its least upper bound is $\sup \mathcal{X} = \{a, b, c\}$. Similarly, the minimal elements of $\mathcal{X}$ are $\{a\}$, $\{b\}$, and $\{c\}$. Thus, the minimum $\min \mathcal{X}$ of $\mathcal{X}$ does not exist, while its greatest lower bound is $\inf \mathcal{X} = \emptyset$.

A set equipped with a total order is a *totally ordered set*. A totally ordered subset $\mathcal{C}$ of a partially ordered set $\langle \mathcal{D}, \sqsubseteq \rangle$ is called a *chain*. A partially ordered set $\langle \mathcal{D}, \sqsubseteq \rangle$ satisfies the *ascending chain condition* (ACC) if and only if any infinite increasing chain $x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$ of elements of $\mathcal{D}$ is not strictly increasing, that is, $\exists k \geq 0 : \forall j \geq k : x_k = x_j$. Dually, a partially ordered set $\langle \mathcal{D}, \sqsubseteq \rangle$ satisfies the *descending chain condition* (DCC) if and only if any infinite decreasing chain $x_0 \sqsupseteq x_1 \sqsupseteq \cdots \sqsupseteq x_n \sqsupseteq \cdots$ of elements of $\mathcal{D}$ is not strictly decreasing, that is, $\exists k \geq 0 : \forall j \geq k : x_k = x_j$.

A *complete partial order* or *cpo* $\langle \mathcal{D}, \sqsubseteq \rangle$ is a poset where every chain $\mathcal{C}$ has a least upper bound $\bigsqcup \mathcal{C}$, which is called the limit of the chain. Note that, since the empty set $\emptyset$ is a chain, a complete partial order has a least element $\bot = \bigsqcup \emptyset$. Thus, a partially ordered set that satisfies the ascending chain condition and that is equipped with a least element, is a complete partial order.

Lattices. A *lattice* $\langle \mathcal{D}, \sqsubseteq, \sqcup, \sqcap \rangle$ is a poset where each pair of elements $x, y \in \mathcal{D}$ has a least upper bound, denoted by $x \sqcup y$, and a greatest lower bound, denoted by $x \sqcap y$. Any totally ordered set is a lattice. A *complete lattice* $\langle \mathcal{D}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ is a lattice where any subset $\mathcal{X} \subseteq \mathcal{D}$ has a least upper bound $\bigsqcup \mathcal{X}$ and a greatest lower bound $\bigsqcap \mathcal{X}$. A complete lattice has both a least element $\bot \stackrel{\text{def}}{=} \bigsqcap \mathcal{D} = \bigsqcup \emptyset$ and a greatest element $\top \stackrel{\text{def}}{=} \bigsqcup \mathcal{D} = \bigsqcap \emptyset$.

Example 2.1.2

The power set $\langle \mathcal{P}(\mathcal{S}), \subseteq, \cup, \cap, \emptyset, \mathcal{S} \rangle$ of any set $\mathcal{S}$ is a complete lattice.

Functions. A *partial function* f from a set $\mathcal{A}$ to a set $\mathcal{B}$, written $f : \mathcal{A} \rightharpoonup \mathcal{B}$, is a binary relation between $\mathcal{A}$ and $\mathcal{B}$ that pairs each element $x \in \mathcal{A}$ with no more than one element $y \in \mathcal{B}$. The set of all partial functions from a set $\mathcal{A}$ to a set $\mathcal{B}$ is denoted by $\mathcal{A} \rightharpoonup \mathcal{B}$. We write $f(x) = y$ if there exists an element y such that $\langle x, y \rangle \in f$, and we say that $f(x)$ is *defined*, otherwise we say that $f(x)$ is *undefined*. Given a partial function $f : \mathcal{A} \rightharpoonup \mathcal{B}$, we define its *domain* as $\mathrm{dom}(f) \stackrel{\text{def}}{=} \{x \in \mathcal{A} \mid \exists y \in \mathcal{B} : f(x) = y\}$. The *totally undefined function*, denoted by $\dot{\emptyset}$, has the empty set as domain: $\mathrm{dom}(\dot{\emptyset}) = \emptyset$. The *join* of two partial functions $f_1 : \mathcal{A} \rightharpoonup \mathcal{B}$ and $f_2 : \mathcal{A} \rightharpoonup \mathcal{B}$ with disjoint domains, denoted

by $f_1 \dot{\cup} f_2 \colon \mathcal{A} \rightharpoonup \mathcal{B}$, is defined as follows:

$$f_1 \dot{\cup} f_2 \stackrel{\text{def}}{=} \lambda x \in \mathcal{A}. \begin{cases} f_1(x) & x \in \mathrm{dom}(f_1) \\ f_2(x) & x \in \mathrm{dom}(f_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $\mathrm{dom}(f_1) \cap \mathrm{dom}(f_2) = \emptyset$.

A (*total*) *function* f from a set $\mathcal{A}$ to a set $\mathcal{B}$, written $f \colon \mathcal{A} \to \mathcal{B}$, is a partial function that pairs each $x \in \mathcal{A}$ with exactly one element $y \in \mathcal{B}$. Equivalently, a (total) function $f \colon \mathcal{A} \to \mathcal{B}$ is a partial function such that $\mathrm{dom}(f) = \mathcal{A}$. The set of all functions from a set $\mathcal{A}$ to a set $\mathcal{B}$ is denoted by $\mathcal{A} \to \mathcal{B}$.

We sometimes denote functions using the *lambda notation* $\lambda x \in \mathcal{A}.\ f(x)$, or more concisely $\lambda x.\ f(x)$.

The *composition* of two functions $f \colon \mathcal{A} \to \mathcal{B}$ and $g \colon \mathcal{B} \to \mathcal{C}$ is another function $g \circ f \colon \mathcal{A} \to \mathcal{C}$ such that $\forall x \in \mathcal{A} : (g \circ f)(x) = g(f(x))$.

The following properties may hold for a function $f \colon \mathcal{A} \to \mathcal{B}$:

$$\forall x, y \in \mathcal{A} : f(x) = f(y) \Rightarrow x = y \qquad \text{(injectivity)}$$
$$\forall y \in \mathcal{B} : \exists x \in \mathcal{A} : f(x) = y \qquad \text{(surjectivity)}$$

A *bijective* function, also called *isomorphism*, is both injective and surjective. Two sets $\mathcal{A}$ and $\mathcal{B}$ are *isomorphic* if there exists a bijective function $f \colon \mathcal{A} \to \mathcal{B}$. The *inverse* of a bijective function $f \colon \mathcal{A} \to \mathcal{B}$ is the bijective function $f^{-1} \colon \mathcal{B} \to \mathcal{A}$ defined as $f^{-1} \stackrel{\text{def}}{=} \{\langle b, a\rangle \mid \langle a, b\rangle \in f\}$.

Let $\langle \mathcal{D}_1, \sqsubseteq_1\rangle$ and $\langle \mathcal{D}_2, \sqsubseteq_2\rangle$ be partially ordered sets. A function $f \colon \mathcal{D}_1 \to \mathcal{D}_2$ is said to be *monotonic* when, for each $x, y \in \mathcal{D}_1$, $x \sqsubseteq_1 y \Rightarrow f(x) \sqsubseteq_2 f(y)$. It is *continuous* (or *Scott-continuous*) when it preserves existing least upper bounds of chains, that is, for each chain $\mathcal{C} \subseteq \mathcal{D}_1$, if $\bigsqcup \mathcal{C}$ exists then $f(\bigsqcup \mathcal{C}) = \bigsqcup\{f(x) \mid x \in \mathcal{C}\}$. Dually, it is *co-continuous* when it preserves existing greatest lower bounds of chains, that is, if $\bigsqcap \mathcal{C}$ exists then $f(\bigsqcap \mathcal{C}) = \bigsqcap\{f(x) \mid x \in \mathcal{C}\}$. A *complete* $\sqcup$*-morphism* (resp. *complete* $\sqcap$*-morphism*) preserves existing least upper bounds (resp. greatest lower bounds) of arbitrary non-empty sets.

Pointwise Lifting. Given a complete lattice $\langle \mathcal{D}, \sqsubseteq, \sqcup, \sqcap, \bot, \top\rangle$ (resp. a lattice, a cpo, a poset) and a set $\mathcal{S}$, the set $\mathcal{S} \to \mathcal{D}$ of all functions from $\mathcal{S}$ to $\mathcal{D}$ inherits the complete lattice (resp. lattice, cpo, poset) structure of $\mathcal{D}$: $\langle \mathcal{S} \to$

$\mathcal{D}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top}\rangle$ where the dotted operators are defined by *pointwise lifting*:

$$\begin{aligned} f \mathrel{\dot{\sqsubseteq}} g &\stackrel{\text{def}}{=} \forall s \in \mathcal{S} : f(s) \sqsubseteq g(s) \\ \dot{\bigsqcup} \mathcal{X} &\stackrel{\text{def}}{=} \lambda s.\ \bigsqcup\{f(s) \mid f \in \mathcal{X}\} \\ \dot{\bigsqcap} \mathcal{X} &\stackrel{\text{def}}{=} \lambda s.\ \bigsqcap\{f(s) \mid f \in \mathcal{X}\} \\ \dot{\bot} &\stackrel{\text{def}}{=} \lambda s.\ \bot \\ \dot{\top} &\stackrel{\text{def}}{=} \lambda s.\ \top \end{aligned} \tag{2.1.1}$$

Ordinals. The theory of ordinals was introduced by Georg Cantor as the core of his set theory [Can95, Can97].

A binary relation R over a set $\mathcal{S}$ is *well-founded* when every non-empty subset of $\mathcal{S}$ has a least element with respect to R. A well-founded total order is called a *well-order* (or a *well-ordering*). A *well-ordered set* $\langle \mathcal{W}, \leq \rangle$ is a set $\mathcal{W}$ equipped with a well-ordering $\leq$. Every well-ordered set is associated with a so-called *ordered type*. Two well-ordered sets $\langle \mathcal{A}, \leq_{\mathcal{A}} \rangle$ and $\langle \mathcal{B}, \leq_{\mathcal{B}} \rangle$ are said to have the same order type if they are *order-isomorphic*, that is, if there exists a bijective function $f \colon \mathcal{A} \to \mathcal{B}$ such that, for all elements $x, y \in \mathcal{A}$, $x \leq_{\mathcal{A}} y$ if and only if $f(x) \leq_{\mathcal{B}} f(y)$.

An *ordinal* is defined as the order type of a well-ordered set and it provides a canonical representative for all well-ordered sets that are order-isomorphic to that well-ordered set. We use lower case Greek letters to denote ordinals. In fact, a well-ordered set $\langle \mathcal{W}, \leq \rangle$ with order type α is order-isomorphic to the well-ordered set $\{x \in \mathcal{W} \mid x < \alpha\}$ of all ordinals strictly smaller than the ordinal α itself. In particular, as suggested by John von Neumann [vN23], this property permits to define each ordinal as the well-ordered set of all ordinals that precede it: the smallest ordinal is the empty set $\emptyset$, denoted by 0. The *successor* of an ordinal α is defined as $\alpha \cup \{\alpha\}$ and is denoted by $\alpha + 1$, or equivalently, by $\mathrm{succ}(\alpha)$. Thus, the first successor ordinal is $\{0\}$, denoted by 1. The next is $\{0, 1\}$, denoted by 2. Continuing in this manner, we obtain all natural numbers, that is, all *finite* ordinals. A *limit ordinal* is an ordinal number which is neither 0 nor a successor ordinal. The set $\mathbb{N}$ of all natural numbers, denoted by ω, is the first limit ordinal and the first *transfinite* ordinal.

We use $\langle \mathbb{O}, \leq \rangle$ to denote the well-ordered set of ordinals. In the following, we will see that the theory of ordinals is the most general setting for proving program termination.

Fixpoints. Given a partially ordered set $\langle \mathcal{D}, \sqsubseteq \rangle$ and a function $f \colon \mathcal{D} \to \mathcal{D}$, a *fixpoint* of f is an element $x \in \mathcal{D}$ such that $f(x) = x$. An element $x \in \mathcal{D}$

such that $x \sqsubseteq f(x)$ is called a *pre-fixpoint* while, dually, a *post-fixpoint* is an element $x \in \mathcal{D}$ such that $f(x) \sqsubseteq x$. The *least fixpoint* of f, written $\operatorname{lfp} f$, is a fixpoint of f such that, for every fixpoint $x \in \mathcal{D}$ of f, $\operatorname{lfp} f \sqsubseteq x$. We write $\operatorname{lfp}_d f$ for the least fixpoint of f which is greater than or equal to an element $d \in \mathcal{D}$. Dually, we define the *greatest fixpoint* of f, denoted by $\operatorname{gfp} f$, and the greatest fixpoint of f smaller than or equal to $d \in \mathcal{D}$, denoted by $\operatorname{gfp}_d f$. When the order $\sqsubseteq$ is not clear from the context, we explicitly write $\operatorname{lfp}^{\sqsubseteq} f$ and $\operatorname{gfp}^{\sqsubseteq} f$.

We now recall a fundamental theorem due to Alfred Tarski [Tar55]:

Theorem 2.1.3 (Tarski's Fixpoint Theorem) *The set of fixpoints of a monotonic function $f\colon \mathcal{D} \to \mathcal{D}$ over a complete lattice is also a complete lattice.*

Proof.
See [Tar55]. ■

In particular, the theorem guarantees that f has a least fixpoint $\operatorname{lfp} f = \bigsqcap\{x \in \mathcal{D} \mid f(x) \sqsubseteq x\}$ and a greatest fixpoint $\operatorname{gfp} f = \bigsqcup\{x \in \mathcal{D} \mid x \sqsubseteq f(x)\}$.

However, such fixpoint characterizations are not constructive. An alternative constructive characterization is often attributed to Stephen Cole Kleene:

Theorem 2.1.4 (Kleene's Fixpoint Theorem) *Let $\langle \mathcal{D}, \sqsubseteq \rangle$ be a complete partial order and let $f\colon \mathcal{D} \to \mathcal{D}$ be a Scott-continuous function. Then, f has a least fixpoint which is the least upper bound of the increasing chain*

$$\bot \;\sqsubseteq\; f(\bot) \;\sqsubseteq\; f(f(\bot)) \;\sqsubseteq\; f(f(f(\bot))) \;\sqsubseteq\; \dots$$

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

In case of monotonic but not continuous functions, a theorem by Patrick Cousot and Radhia Cousot [CC79] expresses fixpoints as limits of possibly *transfinite* iteration sequences:

Theorem 2.1.5 *Let $f\colon \mathcal{D} \to \mathcal{D}$ be a monotonic function over a complete partial order and let $d \in \mathcal{D}$ be a pre-fixpoint. Then, the iteration sequence:*

$$f^\delta \stackrel{def}{=} \begin{cases} d & \delta = 0 \text{ (zero case)} \\ f(f^\alpha) & \delta = \alpha + 1 \text{ (successor case)} \\ \bigsqcup\{f^\alpha \mid \alpha < \delta\} & \text{otherwise (limit case)} \end{cases}$$

converges towards the least fixpoint $\operatorname{lfp}_d f$.

Proof.
See [CC79]. ■

In the following, the partial order between the fixpoint iterates is called *computational order*, in order to distinguish it from the *approximation order* defined in the next Section 2.2.2.

2.2 Abstract Interpretation

Abstract Interpretation [CC77, Cou78] is a general theory for approximating the semantics of programs, originally developed by Patrick Cousot and Radhia Cousot in the late 1970s, as a unifying framework for static program analysis. In the following, we recall the main definitions and results that will be used throughout this thesis. We refer to [CC92b] for an exhaustive presentation.

Transition Systems. The *semantics* of a program is a mathematical characterization of all possible behaviors of the program when executed for all possible input data. In order to be independent from the choice of a particular programming language, programs are often formalized as transition systems:

Definition 2.2.1 (Transition System) *A* transition system *is a pair* $\langle \Sigma, \tau \rangle$ *where* Σ *is a (potentially infinite) set of states and the transition relation* $\tau \subseteq \Sigma \times \Sigma$ *describes the possible transitions between states.*

In a labelled transition system $\langle \Sigma, \mathcal{A}, \tau \rangle$, $\mathcal{A}$ *is a set of actions that are used to label the transitions described by the transition relation* $\tau \subseteq \Sigma \times \mathcal{A} \times \Sigma$.

Note that this model allows representing programs with (possibly unbounded) non-determinism. In the following, in order to lighten the notation, a transition $\langle s, s' \rangle \in \tau$ (resp. a labelled transition $\langle s, a, s' \rangle \in \tau$) between a state s and another state s' is sometimes written as $s \to s'$ (resp. $s \xrightarrow{a} s'$).

In some cases, a set $\mathcal{I} \subseteq \Sigma$ is designated as the set of *initial states*. The set of *blocking* or *final states* is $\Omega \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma : \langle s, s' \rangle \notin \tau\}$.

We define the following functions to manipulate sets of program states.

Definition 2.2.2 *Given a transition system* $\langle \Sigma, \tau \rangle$, $\mathrm{post} \colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ *maps a set of program states* $X \in \mathcal{P}(\Sigma)$ *to the set of their successors with respect to the program transition relation* τ*:*

$$\mathrm{post}(X) \stackrel{def}{=} \{s' \in \Sigma \mid \exists s \in X : \langle s, s' \rangle \in \tau\} \tag{2.2.1}$$

Definition 2.2.3 *Given a transition system* $\langle \Sigma, \tau \rangle$, $\mathrm{pre}\colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ *maps a set of program states* $X \in \mathcal{P}(\Sigma)$ *to the set of their predecessors with respect to the program transition relation* τ*:*

$$\mathrm{pre}(X) \stackrel{def}{=} \{s \in \Sigma \mid \exists s' \in X : \langle s, s' \rangle \in \tau\} \tag{2.2.2}$$

Definition 2.2.4 *Given a transition system* $\langle \Sigma, \tau \rangle$, $\widetilde{\mathrm{post}}\colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ *maps a set of states* $X \in \mathcal{P}(\Sigma)$ *to the set of states whose successors with respect to the program transition relation* τ *are all in the set* X*:*

$$\widetilde{\mathrm{post}}(X) = \Sigma \setminus \mathrm{post}(\Sigma \setminus X) \stackrel{def}{=} \{s' \in \Sigma \mid \forall s \in \Sigma : \langle s, s' \rangle \in \tau \Rightarrow s \in X\} \tag{2.2.3}$$

Definition 2.2.5 *Given a transition system* $\langle \Sigma, \tau \rangle$, $\widetilde{\mathrm{pre}}\colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ *maps a set of states* $X \in \mathcal{P}(\Sigma)$ *to the set of states whose predecessors with respect to the program transition relation* τ *are all in the set* X*:*

$$\widetilde{\mathrm{pre}}(X) = \Sigma \setminus \mathrm{pre}(\Sigma \setminus X) \stackrel{def}{=} \{s \in \Sigma \mid \forall s' \in \Sigma : \langle s, s' \rangle \in \tau \Rightarrow s' \in X\} \tag{2.2.4}$$

2.2.1 Maximal Trace Semantics

The semantics generated by a transition system is the set of computations described by the transition system. We formally define this notion below.

Sequences. Given a set $\mathcal{S}$, the set $\mathcal{S}^n \stackrel{\text{def}}{=} \{s_0 \cdots s_{n-1} \mid \forall i < n : s_i \in \mathcal{S}\}$ is the set of all sequences of exactly n elements from $\mathcal{S}$. We write ε to denote the empty sequence, i.e., $\mathcal{S}^0 \triangleq \{\varepsilon\}$.

In the following, let $\mathcal{S}^* \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \mathcal{S}^n$ be the set of all finite sequences, $\mathcal{S}^+ \stackrel{\text{def}}{=} \mathcal{S}^* \setminus \mathcal{S}^0$ be the set of all non-empty finite sequences, $\mathcal{S}^\omega$ be the set of all infinite sequences, $\mathcal{S}^{+\infty} \stackrel{\text{def}}{=} \mathcal{S}^+ \cup \mathcal{S}^\omega$ be the set of all non-empty finite or infinite sequences and $\mathcal{S}^{*\infty} \stackrel{\text{def}}{=} \mathcal{S}^* \cup \mathcal{S}^\omega$ be the set of all finite or infinite sequences of elements from $\mathcal{S}$. In the following, in order to ease the notation, sequences of a single element $s \in \mathcal{S}$ are often written omitting the curly brackets, e.g., we write s^ω and $s^{+\infty}$ instead of $\{s\}^\omega$ and $\{s\}^{+\infty}$.

We write $\sigma\sigma'$ (or $\sigma \cdot \sigma'$) for the concatenation of two sequences $\sigma, \sigma' \in \mathcal{S}^{+\infty}$ (with $\sigma\varepsilon = \varepsilon\sigma = \sigma$, and $\sigma\sigma' = \sigma$ when $\sigma \in \mathcal{S}^\omega$), $\mathcal{T}^+ \stackrel{\text{def}}{=} \mathcal{T} \cap \mathcal{S}^+$ for the selection of the non-empty finite sequences of $\mathcal{T} \subseteq \mathcal{S}^{+\infty}$, $\mathcal{T}^\omega \stackrel{\text{def}}{=} \mathcal{T} \cap \mathcal{S}^\omega$ for the selection of the infinite sequences of $\mathcal{T} \subseteq \mathcal{S}^{+\infty}$ and $\mathcal{T} \,;\, \mathcal{T}' \stackrel{\text{def}}{=} \{\sigma s \sigma' \mid s \in \mathcal{S} \ \wedge\ \sigma s \in \mathcal{T} \ \wedge\ s\sigma' \in \mathcal{T}'\} \cup \mathcal{T}^\omega$ for the merging of sets of sequences $\mathcal{T}, \mathcal{T}' \subseteq \mathcal{S}^{+\infty}$.

Traces. Given a transition system $\langle \Sigma, \tau \rangle$, a *trace* is a non-empty sequence of states in Σ determined by the transition relation τ, that is, $\langle s, s' \rangle \in \tau$ for each pair of consecutive states $s, s' \in \Sigma$ in the sequence. Note that, the set of final states Ω and the transition relation τ can be understood as a set of traces of length one and a set of traces of length two, respectively. The set of all traces generated by a transition system is called *partial trace semantics*:

Definition 2.2.6 (Partial Trace Semantics) *The* partial trace semantics $\dot{\tau}^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ *generated by a transition system* $\langle \Sigma, \tau \rangle$ *is defined as follows:*

$$\dot{\tau}^{+\infty} \stackrel{def}{=} \dot{\tau}^{+} \cup \tau^{\omega}$$

where $\dot{\tau}^{+} \in \mathcal{P}(\Sigma^{+})$ *is the set of finite traces:*

$$\dot{\tau}^{+} \stackrel{def}{=} \bigcup_{n>0} \{ s_0 \cdots s_{n-1} \in \Sigma^n \mid \forall i < n-1 : \langle s_i, s_{i+1} \rangle \in \tau \}$$

and $\tau^{\omega} \in \mathcal{P}(\Sigma^{\omega})$ *is the set of infinte traces:*

$$\tau^{\omega} \stackrel{def}{=} \{ s_0 s_1 \cdots \in \Sigma^{\omega} \mid \forall i \in \mathbb{N} : \langle s_i, s_{i+1} \rangle \in \tau \}$$

Example 2.2.7

Let $\Sigma = \{a, b\}$ and $\tau = \{\langle a, a \rangle, \langle a, b \rangle\}$. The partial trace semantics generated by $\langle \Sigma, \tau \rangle$ is the set of traces $a^{+\infty} \cup a^{*}b$.

Maximal Trace Semantics. In practice, given a transition system $\langle \Sigma, \tau \rangle$, and possibly a set of initial states $\mathcal{I} \subseteq \Sigma$, the traces worth of consideration (start by an initial state in $\mathcal{I}$ and) either are infinite or terminate with a final state in Ω. These traces define the *maximal trace semantics* $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ and represent infinite computations or completed finite computations:

Definition 2.2.8 (Maximal Trace Semantics) *The* maximal trace semantics $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ *generated by a transition system* $\langle \Sigma, \tau \rangle$ *is defined as:*

$$\tau^{+\infty} \stackrel{def}{=} \tau^{+} \cup \tau^{\omega}$$

where $\tau^{+} \in \mathcal{P}(\Sigma^{+})$ *is the set of finite traces terminating with a final state in* Ω*:*

$$\tau^{+} \stackrel{def}{=} \bigcup_{n>0} \{ s_0 \cdots s_{n-1} \in \Sigma^n \mid \forall i < n-1 : \langle s_i, s_{i+1} \rangle \in \tau, s_{n-1} \in \Omega \}$$

Example 2.2.9

The maximal trace semantics generated by the transition system $\langle \Sigma, \tau \rangle$ of Example 2.2.7 is the set of traces $a^\omega \cup a^*b$. Note that, unlike the partial trace semantics of Example 2.2.7, the maximal trace semantics does not represent partial computations, i.e., finite sequences of $a \in \Sigma$.

In practice, in case a set of initial states $\mathcal{I} \subseteq \Sigma$ is given, only the traces starting from an initial state $s \in \mathcal{I}$ are considered: $\{s\sigma \in \tau^{+\infty} \mid s \in \mathcal{I}\}$.

Remark 2.2.10 It is worth mentioning that not all program semantics are directly generated by a transition system. For example, this is the case of the semantics of programs where the future evolution of a computation depends on the whole computation history.

Example 2.2.11

Let $\Sigma = \{a, b\}$. The set of *fair* traces a^*b, where the event b eventually occurs, is an example of program semantics that cannot be generated by a transition system. As seen in Example 2.2.7 and Example 2.2.9, the transition relation $\tau = \{\langle a, a\rangle, \langle a, b\rangle\}$ introduces spurious traces in $a^{+\infty}$.

In fact, transition systems can only describe computations whose future evolution depends entirely on their current state [Cou85]. We will often come back to this remark throughout this thesis. ■

The following result provides a fixpoint definition of the maximal trace semantics within the complete lattice $\langle \mathcal{P}(\Sigma^{+\infty}), \sqsubseteq, \sqcup, \sqcap, \Sigma^\omega, \Sigma^+ \rangle$, where the computational order is $T_1 \sqsubseteq T_2 \stackrel{\text{def}}{=} T_1^+ \subseteq T_2^+ \wedge T_1^\omega \supseteq T_2^\omega$ [Cou97]:

Theorem 2.2.12 (Maximal Trace Semantics) *The* maximal trace semantics $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ *can be expressed as a least fixpoint in the complete lattice* $\langle \mathcal{P}(\Sigma^{+\infty}), \sqsubseteq, \sqcup, \sqcap, \Sigma^\omega, \Sigma^+ \rangle$ *as follows:*

$$\begin{aligned} \tau^{+\infty} &= \mathrm{lfp}^{\sqsubseteq}\ \phi^{+\infty} \\ \phi^{+\infty}(T) &\stackrel{def}{=} \Omega \cup (\tau\ ;\ T) \end{aligned} \tag{2.2.5}$$

Proof.

See [Cou02]. ■

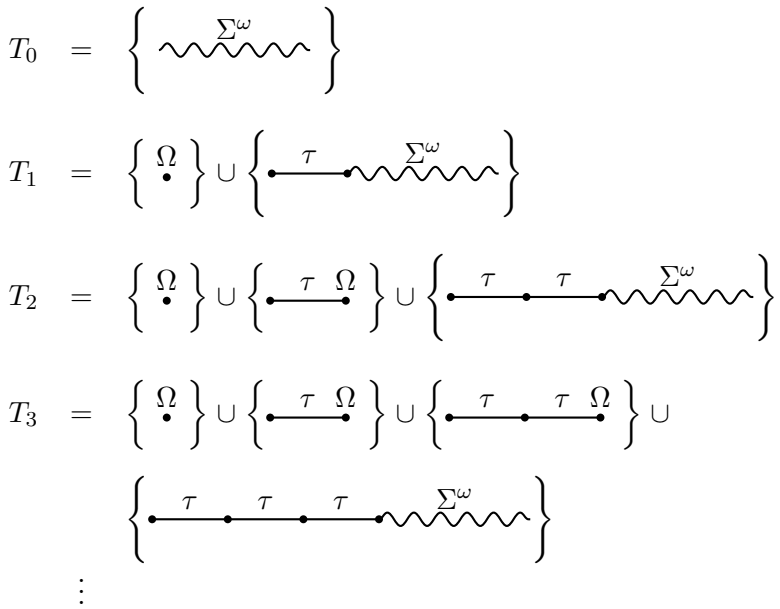


Figure 2.2: Fixpoint iterates of the maximal trace semantics.

In Figure 2.2, we propose an illustration of the fixpoint iterates. Intuitively, the traces belonging to the maximal trace semantics are built *backwards* by prepending transitions to them: the finite traces are built extending other finite traces from the set of final states Ω, and the infinite traces are obtained selecting infinite sequences with increasingly longer prefixes forming traces. In particular, the ith iterate builds all finite traces of length i, and selects all infinite sequences whose prefixes of length i form traces. At the limit we obtain all infinite traces and all finite traces that terminate in Ω.

2.2.2 Galois Connections

The maximal trace semantics carries all information about a program. It is the most precise semantics and it fully describes the behavior of a program.

Then, usually another fixpoint semantics is designed that is minimal, sound and relatively complete for the program properties of interest.

Program Properties. Following [CC77], a property is represented by the set of elements which have this property. By *program property* we mean a property of its executions, that is a property of its semantics. Since a program semantics is a set of traces, a program property is a set of set of traces.

The strongest property of a program with semantics $\mathcal{S} \in \mathcal{P}(\Sigma^{+\infty})$ is the *collecting semantics* $\{\mathcal{S}\} \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$. This is the strongest property because any program with this property must have the same semantics.

We say that a program with semantics $\mathcal{S} \in \mathcal{P}(\Sigma^{+\infty})$ satisfies a property $P \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ if and only if the property P is implied by the collecting semantics $\{\mathcal{S}\} \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ of the program, that is, if and only if $\{\mathcal{S}\} \subseteq P$.

In practice, a weaker form of program correctness is often used. In fact, the traditional safety and liveness properties are trace properties and thus can be modeled as sets of traces. In this case, a program satisfies a property $P \in \mathcal{P}(\Sigma^{+\infty})$ if and only if its semantics $\mathcal{S} \in \mathcal{P}(\Sigma^{+\infty})$ implies the property P, that is, if and only if $\mathcal{S} \subseteq P$.

Example 2.2.13

Let us consider a program with semantics $\mathcal{S} \in \mathcal{P}(\Sigma^{+\infty})$. We can model the property of program termination as the set of all non-empty finite sequences Σ^{+}. The program terminates if and only if $\mathcal{S} \subseteq \Sigma^{+}$.

Note that, not all program properties are trace properties. As an example, the non-interference policy is not a trace property, because whether a trace is allowed by the policy depends on whether another trace is also allowed [CS10].

We prefer program semantics expressed in fixpoint form which directly lead, using David Park's fixpoint induction [Par69], to sound and complete proof methods for the correctness of a program with respect to a property:

Theorem 2.2.14 (Park's Fixpoint Induction Principle) *Let $f : \mathcal{D} \to \mathcal{D}$ be a monotonic function over a complete lattice $\langle \mathcal{D}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ and let $d \in \mathcal{D}$ be a pre-fixpoint. Then, given an element $P \in \mathcal{D}$, we have:*

$$\mathrm{lfp}_d\, f \sqsubseteq P \;\Leftrightarrow\; \exists I \in \mathcal{D} : d \sqsubseteq I \;\wedge\; f(I) \sqsubseteq I \;\wedge\; I \sqsubseteq P. \tag{2.2.6}$$

Dually, we have:

$$P \sqsubseteq \mathrm{gfp}_d\, f \;\Leftrightarrow\; \exists I \in \mathcal{D} : I \sqsubseteq d \;\wedge\; I \sqsubseteq f(I) \;\wedge\; P \sqsubseteq I. \tag{2.2.7}$$

The element $I \in \mathcal{D}$ is called *inductive invariant*.

Hierarchy of Semantics. As mentioned, to reason about a particular program property, it is not necessary to consider all aspects and details of the program behavior. In fact, reasoning is facilitated by the design of a well-adapted semantics, abstracting away from irrelevant matters. Therefore, there

is no universal *general-purpose* program semantics but rather a wide variety of *special-purpose* program semantics, each one of them dedicated to a particular class of program properties and reduced to essentials in order to ignore irrelevant details about program executions.

Abstract interpretation is a method for relating these semantics. In fact, they can be uniformly described as fixpoints of monotonic functions over ordered structures, and organized into a hierarchy of interrelated semantics specifying at various levels of detail the behavior of programs [Cou97, Cou02].

Galois Connections. The correspondence between the semantics in the hierarchy is established by Galois connections formalizing the loss of information:

Definition 2.2.15 (Galois Connection) *Let $\langle \mathcal{D}, \sqsubseteq \rangle$ and $\langle \mathcal{D}^\natural, \sqsubseteq^\natural \rangle$ be two partially ordered sets. A* Galois connection $\langle \mathcal{D}, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{D}^\natural, \sqsubseteq^\natural \rangle$ *is a pair of monotonic functions $\alpha \colon \mathcal{D} \to \mathcal{D}^\natural$ and $\gamma \colon \mathcal{D}^\natural \to \mathcal{D}$ such that:*

$$\forall d \in \mathcal{D}, d^\natural \in \mathcal{D}^\natural : \alpha(d) \sqsubseteq^\natural d^\natural \;\Leftrightarrow\; d \sqsubseteq \gamma(d^\natural) \tag{2.2.8}$$

We write $\langle \mathcal{D}, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftarrow\!\twoheadrightarrow}} \langle \mathcal{D}^\natural, \sqsubseteq^\natural \rangle$ when α is surjective (or, equivalently, γ is injective), $\langle \mathcal{D}, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\twoheadleftarrow\!\rightarrow}} \langle \mathcal{D}^\natural, \sqsubseteq^\natural \rangle$ when α is injective (or, equivalently, γ is surjective), and $\langle \mathcal{D}, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\twoheadleftarrow\!\twoheadrightarrow}} \langle \mathcal{D}^\natural, \sqsubseteq^\natural \rangle$ when both α and γ are bijective.

The posets $\langle \mathcal{D}, \sqsubseteq \rangle$ and $\langle \mathcal{D}^\natural, \sqsubseteq^\natural \rangle$ are called the *concrete domain* and the *abstract domain*, respectively. The function $\alpha \colon \mathcal{D} \to \mathcal{D}^\natural$ is the *abstraction function*, which provides the abstract approximation $\alpha(d)$ of a concrete element $d \in \mathcal{D}$, and $\gamma \colon \mathcal{D}^\natural \to \mathcal{D}$ is the *concretization function*, which provides the concrete element $\gamma(d^\natural)$ corresponding to the abstract description $d^\natural \in \mathcal{D}^\natural$. Note that, the notion of Galois connection can also be defined on preordered sets.

The orders $\sqsubseteq$ and $\sqsubseteq^\natural$ are called *approximation orders*. They dictate the relative precision of the elements in the concrete and abstract domain: if $\alpha(d) \sqsubseteq^\natural d^\natural$, then $d^\natural$ is also a correct abstract approximation of the concrete element d, although less precise than $\alpha(d)$; if $d \sqsubseteq \gamma(d^\natural)$, then $d^\natural$ is also a correct abstract approximation of the concrete element d, although the element d provides more accurate information about program executions than $\gamma(d^\natural)$.

Remark 2.2.16 Observe that, the computational order used to define fixpoints and the approximation order often coincide but, in the general case, they are distinct and totally unrelated. We will need to maintain this distinction throughout the rest of this thesis. ■

The function $\gamma \circ \alpha \colon \mathcal{D} \to \mathcal{D}$ is *extensive*, that is, $\forall d \in \mathcal{D} : d \sqsubseteq \gamma \circ \alpha(d)$, meaning that the loss of information in the abstraction process is sound. The function $\alpha \circ \gamma \colon \mathcal{D}^\natural \to \mathcal{D}^\natural$ is *reductive*, that is, $\forall d^\natural \in \mathcal{D}^\natural : \alpha \circ \gamma(d^\natural) \sqsubseteq^\natural d^\natural$, meaning that the concretization introduces no loss of information.

Given a Galois connection $\langle \mathcal{D}, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{D}^\natural, \sqsubseteq^\natural \rangle$, the abstraction function and the concretization function uniquely determine each other:

$$\alpha(d) \stackrel{\text{def}}{=} \bigsqcap \{d^\natural \mid d \sqsubseteq \gamma(d^\natural)\}$$
$$\gamma(d^\natural) \stackrel{\text{def}}{=} \bigsqcup{}^\natural \{d \mid \alpha(d) \sqsubseteq^\natural d^\natural\}$$

For this reason, we often provide only the definition of the abstraction function or, indifferently, only the definition of the concretization function.

Another important property of Galois connections is that, given two Galois connections $\langle \mathcal{D}, \sqsubseteq \rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle \mathcal{D}^\#, \sqsubseteq^\# \rangle$ and $\langle \mathcal{D}^\#, \sqsubseteq^\# \rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle \mathcal{D}^\natural, \sqsubseteq^\natural \rangle$, their composition $\langle \mathcal{D}, \sqsubseteq \rangle \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} \langle \mathcal{D}^\natural, \sqsubseteq^\natural \rangle$ is also a Galois connection. Hence, abstract interpretation has a *constructive* aspect, since program semantics can be systematically derived by successive abstractions of the maximal trace semantics, rather than just being derived by intuition and justified a posteriori.

Example 2.2.17

The function $\mathrm{post} \in \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ (cf. Definition 2.2.2) and the function $\widetilde{\mathrm{pre}} \in \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ (cf. Definition 2.2.5) form a Galois connection:

$$\langle \mathcal{P}(\Sigma), \subseteq \rangle \underset{\mathrm{post}}{\overset{\widetilde{\mathrm{pre}}}{\leftrightarrows}} \langle \mathcal{P}(\Sigma), \subseteq \rangle.$$

Example 2.2.18 (Reachable State Semantics)

The *reachable state semantics* $\tau_{\mathrm{R}} \in \mathcal{P}(\Sigma)$ collects the set of states that are reachable from a designated set $\mathcal{I} \subseteq \Sigma$ of initial states. It can be derived by abstraction of the maximal trace semantics $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ by means of the Galois connection $\langle \mathcal{P}(\Sigma^{+\infty}), \subseteq \rangle \underset{\alpha^{\mathrm{R}}}{\overset{\gamma^{\mathrm{R}}}{\leftrightarrows}} \langle \mathcal{P}(\Sigma), \subseteq \rangle$ where the abstraction function $\alpha^{\mathrm{R}} \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma)$ is defined as follows:

$$\alpha^{\mathrm{R}}(T) \stackrel{\text{def}}{=} \mathcal{I} \cup \{s \in \Sigma \mid \exists s' \in \mathcal{I}, \sigma \in \Sigma^*, \sigma' \in \Sigma^{*\infty} : s'\sigma s\sigma' \in T\} \tag{2.2.9}$$

This abstraction, from now on, is called *reachability abstraction*.

Note that, the approximation order $\subseteq$ of the concrete domain $\langle \mathcal{P}(\Sigma^{+\infty}), \subseteq \rangle$ differs from the computational order $\sqsubseteq$ used to define the maximal trace semantics in the complete lattice $\langle \mathcal{P}(\Sigma^{+\infty}), \sqsubseteq, \sqcup, \sqcap, \Sigma^\omega, \Sigma^+ \rangle$ (cf. Equation 2.2.5).

The reachable state semantics is thus defined as:

$$\tau_{\mathrm{R}} \stackrel{\text{def}}{=} \alpha^{\mathrm{R}}(\tau^{+\infty})$$

It can be specified in fixpoint form as follows:

$$\begin{aligned} \tau_{\mathrm{R}} &= \mathrm{lfp}^{\subseteq}\ \phi_{\mathrm{R}} \\ \phi_{\mathrm{R}}(S) &\stackrel{\text{def}}{=} \mathcal{I} \cup \mathrm{post}(S) \end{aligned} \tag{2.2.10}$$

In this case, the computational order $\subseteq$ used to define the semantics coincides with the approximation order of the abstract domain $\langle \mathcal{P}(\Sigma), \subseteq \rangle$.

Note that, while the traces belonging to the maximal trace semantics are built backwards, the states belonging to the reachable state semantics are build *forward* from the set of initial states $\mathcal{I}$.

Remark 2.2.19 (Absence of Galois Connection) The use of Galois connections squares with an ideal situation where there is a best way to approximate any concrete property by an abstract property. However, imposing the existence of a Galois connection is sometimes too strong a requirement. In [CC92b], Patrick Cousot and Radhia Cousot illustrate how to relax the Galois connection framework in order to work with only a concretization function or, dually, only an abstraction function. In practice, concretization-based abstract interpretations are much more used and will be frequently encountered throughout this thesis (cf. Section 3.4 and Chapter 5). ■

Fixpoint Transfer. The following theorem provides guidelines for deriving an abstract fixpoint semantics $\mathcal{S}^\natural$ by abstraction of a concrete fixpoint semantics $\mathcal{S}$, or dually, for deriving $\mathcal{S}$ by refinement of $\mathcal{S}^\natural$.

Theorem 2.2.20 (Kleenian Fixpoint Transfer) *Let $\langle \mathcal{D}, \sqsubseteq \rangle$ and $\langle \mathcal{D}^\natural, \sqsubseteq^\natural \rangle$ be complete partial orders, let $\phi\colon \mathcal{D} \to \mathcal{D}$ and $\phi^\natural\colon \mathcal{D}^\natural \to \mathcal{D}^\natural$ be monotonic functions, and let $\alpha\colon \mathcal{D} \to \mathcal{D}^\natural$ be a Scott-continuous abstraction function that satisfies $\alpha(\bot) = \bot^\natural$ and the commutation condition $\alpha \circ \phi = \phi^\natural \circ \alpha$. Then, we have the fixpoint abstraction $\alpha(\mathrm{lfp}^{\sqsubseteq}\ \phi) = \mathrm{lfp}^{\sqsubseteq^\natural}\ \phi^\natural$.*

Dually, let $\gamma\colon \mathcal{D}^\natural \to \mathcal{D}$ be a Scott-continuous concretization function that satisfies $\bot = \gamma(\bot^\natural)$ and the commutation condition $\phi \circ \gamma = \gamma \circ \phi^\natural$. Then, we have the fixpoint derivation $\mathrm{lfp}^{\sqsubseteq}\ \phi = \gamma(\mathrm{lfp}^{\sqsubseteq^\natural}\ \phi^\natural)$.

In particular, for the respective iterates of $\phi\colon \mathcal{D} \to \mathcal{D}$ and $\phi^\natural\colon \mathcal{D}^\natural \to \mathcal{D}^\natural$ from $\bot$ and $\bot^\natural$ (cf. Theorem 2.1.5) we have: $\forall \delta \in \mathbb{O} : \alpha(\phi^\delta) = \phi^{\natural\delta}$.

Proof.
See [Cou02]. ■

When the abstraction function is not Scott-continuous, but it preserves greatest lower bounds, we can rely on the following theorem.

Theorem 2.2.21 (Tarskian Fixpoint Transfer) *Let* $\langle \mathcal{D}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ *and* $\langle \mathcal{D}^\natural, \sqsubseteq^\natural, \sqcup^\natural, \sqcap^\natural, \bot^\natural, \top^\natural \rangle$ *be complete lattices, let* $\phi \colon \mathcal{D} \to \mathcal{D}$ *and* $\phi^\natural \colon \mathcal{D}^\natural \to \mathcal{D}^\natural$ *be monotonic functions, and let* $\alpha \colon \mathcal{D} \to \mathcal{D}^\natural$ *be an abstraction function that is a complete* $\sqcap$*-morphism and that satisfies* $\phi^\natural \circ \alpha \sqsubseteq^\natural \alpha \circ \phi$ *and the post-fixpoint correspondence* $\forall d^\natural \in \mathcal{D}^\natural : \phi^\natural(d^\natural) \sqsubseteq^\natural d^\natural \Rightarrow \exists d \in \mathcal{D} : \phi(d) \sqsubseteq d \wedge \alpha(d) = d^\natural$ *(i.e., each abstract post-fixpoint of* $\phi^\natural$ *is the abstraction by* α *of some concrete post-fixpoint of* ϕ*). Then, we have* $\alpha(\mathrm{lfp}^{\sqsubseteq}\ \phi) = \mathrm{lfp}^{\sqsubseteq}\ \phi^\natural$.

Proof.
See [Cou02]. ■

In the particular case when the abstraction function $\alpha \colon \mathcal{D} \to \mathcal{D}^\natural$ is a complete $\sqcup$-morphism, there exists a unique concretization function $\gamma \colon \mathcal{D}^\natural \to \mathcal{D}$ such that $\langle \mathcal{D}, \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{D}^\natural, \sqsubseteq^\natural \rangle$ is a Galois connection.

Fixpoint Approximation. In case no optimal fixpoint abstraction of a concrete semantics $\mathcal{S}$ can be defined, we settle for a *sound* abstraction, that is, an abstract semantics $\mathcal{S}^\natural$ such that $\alpha(\mathcal{S}) \sqsubseteq^\natural \mathcal{S}^\natural$, or equivalently $\mathcal{S} \sqsubseteq \gamma(\mathcal{S}^\natural)$, for the approximation orders $\sqsubseteq$ and $\sqsubseteq^\natural$.

2.2.3 Widening and Narrowing

We must now address the practical problem of effectively computing these program semantics. In [CC76], Patrick Cousot and Radhia Cousot introduced the idea of using widening and narrowing operators in order to accelerate the convergence of increasing and decreasing iteration sequences to a fixpoint over-approximation. In [Cou78], the dual operators are also considered.

Widening. The *widening* operator is used to enforce or accelerate the convergence of increasing iteration sequences over abstract domains with infinite or very long strictly ascending chains, or even over finite but very large abstract domains. It is defined as follows:

Definition 2.2.22 (Widening) *Let* $\langle \mathcal{D}, \sqsubseteq \rangle$ *be a partially ordered set. A* widening *operator* $\triangledown \colon (\mathcal{D} \times \mathcal{D}) \to \mathcal{D}$ *is such that:*

(1) for all element $x, y \in \mathcal{D}$, we have $x \sqsubseteq x \triangledown y$ and $y \sqsubseteq x \triangledown y$;

(2) for all increasing chains $x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$, the increasing chain

$$\begin{array}{rcl} y_0 & \stackrel{def}{=} & x_0 \\ y_{n+1} & \stackrel{def}{=} & y_n \triangledown x_{n+1} \end{array}$$

is ultimately stationary, that is, $\exists l \geq 0 : \forall j \geq l : y_j = y_l$.

Intuitively, given and abstract domain $\langle \mathcal{D}, \sqsubseteq \rangle$ and a function $\phi \colon \mathcal{D} \to \mathcal{D}$, the widening uses two consecutive iterates y_n and $\phi(y_n)$ in order to extrapolate the next iterate $y_{n+1} \stackrel{\text{def}}{=} y_n \triangledown \phi(y_n)$. This *extrapolation* should be an over-approximation for soundness, that is $y_n \sqsubseteq y_{n+1}$ and $\phi(y_n) \sqsubseteq y_{n+1}$ (cf. Definition 2.2.22(1)), and enforce convergence for termination (cf. Definition 2.2.22(2)). In this way, the widening allows computing in finite time an over-approximation of a Kleenian fixpoint:

Theorem 2.2.23 (Fixpoint Approximation with Widening) *Let $\langle \mathcal{D}, \sqsubseteq \rangle$ be a complete partial order, let $\phi \colon \mathcal{D} \to \mathcal{D}$ be a Scott-continuous function, let $d \in \mathcal{D}$ be a pre-fixpoint, and let $\triangledown \colon (\mathcal{D} \times \mathcal{D}) \to \mathcal{D}$ be a widening. Then, the following increasing chain:*

$$\begin{array}{rcl} y_0 & \stackrel{def}{=} & d \\ y_{n+1} & \stackrel{def}{=} & y_n \triangledown \phi(y_n) \end{array}$$

is ultimately stationary and its limit, denoted by $\phi^{\triangledown}$, is such that $\mathrm{lfp}_d\, \phi \sqsubseteq \phi^{\triangledown}$.

Proof.
See [CC92c]. ■

In [CC92c], Patrick Cousot and Radhia Cousot demonstrated that computing in an abstract domain with infinite ascending chains using a widening is strictly more powerful than any finite abstraction. Intuitively, the widening adds a dynamic dimension to the abstraction, which is more flexible than relying only on the static choice of an abstract domain.

Narrowing. The *narrowing* operator is used to enforce or accelerate the convergence of decreasing iteration sequences. It is defined as follows:

Definition 2.2.24 (Narrowing) *Let $\langle \mathcal{D}, \sqsubseteq \rangle$ be a partially ordered set. A* narrowing *operator $\triangle \colon (\mathcal{D} \times \mathcal{D}) \to \mathcal{D}$ is such that:*

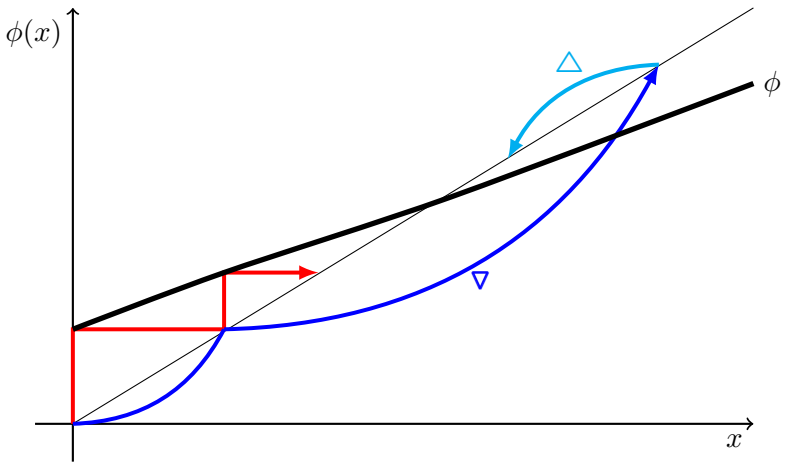


Figure 2.3: Example of increasing iteration with widening ($\xrightarrow{\nabla}$) followed by a decreasing iteration with narrowing ($\xrightarrow{\triangle}$) in order to over-approximate the limit of an increasing iteration sequence ($\longrightarrow$).

(1) for all element $x, y \in \mathcal{D}$, if $x \sqsupseteq y$ we have $x \sqsupseteq (x \triangle y) \sqsupseteq y$;

(2) for all decreasing chains $x_0 \sqsupseteq x_1 \sqsupseteq \cdots \sqsupseteq x_n \sqsupseteq \cdots$, the decreasing chain

$$\begin{array}{rcl} y_0 & \stackrel{def}{=} & x_0 \\ y_{n+1} & \stackrel{def}{=} & y_n \triangle x_{n+1} \end{array}$$

is ultimately stationary, that is, $\exists l \geq 0 : \forall j \geq l : y_j = y_l$.

It is often the case that the limit ϕ^∇ of the iteration sequence with widening is a strict post-fixpoint of ϕ: $\phi(\phi^\nabla) \sqsubset \phi^\nabla$. Hence, the over-approximation ϕ^∇ can be refined by a decreasing iteration without widening:

$$\begin{array}{rcl} y_0 & \stackrel{\text{def}}{=} & \phi^\nabla \\ y_{n+1} & \stackrel{\text{def}}{=} & \phi(y_n) \end{array}$$

However, this decreasing sequence can be infinite. The *narrowing* operator is used to limit the refinement while enforcing termination (cf. Figure 2.3). Intuitively, the narrowing uses two consecutive iterates y_n and $\phi(y_n)$ in order to compute the next iterate $y_{n+1} \stackrel{\text{def}}{=} y_n \triangle \phi(y_n)$. This should be an *interpolation* for soundness, that is $y_n \sqsupseteq y_{n+1} \sqsupseteq \phi(y_n)$ (cf. Definition 2.2.24(1)), and enforce convergence for termination (cf. Definition 2.2.24(2)). In this way, the narrowing allows refining in finite time an over-approximation of a fixpoint:

Theorem 2.2.25 (Fixpoint Refinement with Narrowing) *Let $\langle \mathcal{D}, \sqsubseteq \rangle$ be a complete partial order, let $\phi \colon \mathcal{D} \to \mathcal{D}$ be a Scott-continuous function, let $d \in \mathcal{D}$ be a pre-fixpoint and let $a \in \mathcal{D}$ be a post-fixpoint such that $d \sqsubseteq a$, and let $\triangle \colon (\mathcal{D} \times \mathcal{D}) \to \mathcal{D}$ be a narrowing. Then, the decreasing chain*

$$\begin{array}{rcl} y_0 & \stackrel{def}{=} & a \\ y_{n+1} & \stackrel{def}{=} & y_n \triangle \phi(y_n) \end{array}$$

is ultimately stationary and its limit, denoted by $\phi^{\triangle}$, is such that $\mathrm{lfp}_d\, \phi \sqsubseteq \phi^{\triangle}$.

Proof.
See [CC92c]. ■

Dual Widening. The *dual widening* operator is used to enforce or accelerate the convergence of decreasing iterations sequences to a (greatest) fixpoint under-approximation. Its definition is the dual of Definition 2.2.22:

Definition 2.2.26 (Dual Widening) *Let $\langle \mathcal{D}, \sqsubseteq \rangle$ be a partially ordered set. A* dual widening *operator $\bar{\nabla} \colon (\mathcal{D} \times \mathcal{D}) \to \mathcal{D}$ is such that:*

(1) for all element $x, y \in \mathcal{D}$, we have $x \sqsupseteq x \bar{\nabla} y$ and $y \sqsupseteq x \bar{\nabla} y$;

(2) for all decreasing chains $x_0 \sqsupseteq x_1 \sqsupseteq \cdots \sqsupseteq x_n \sqsupseteq \cdots$, the decreasing chain

$$\begin{array}{rcl} y_0 & \stackrel{def}{=} & x_0 \\ y_{n+1} & \stackrel{def}{=} & y_n \bar{\nabla} x_{n+1} \end{array}$$

is ultimately stationary, that is, $\exists l \geq 0 : \forall j \geq l : y_j = y_l$.

The widening and the dual widening are *extrapolators*: they are used to find abstract elements outside the range of known abstract elements [Cou15].

Dual Narrowing. The *dual narrowing* operator is used to enforce or accelerate the convergence of increasing iterations sequences to a (greatest) fixpoint under-approximation. Its definition is the dual of Definition 2.2.24.

The narrowing and the dual narrowing are *interpolators*: they are used to find abstract elements inside the range of known abstract elements [Cou15].

In conclusion, a *sound* and fully *automatic* static analyzer can be designed by systematically approximating the semantics of programs, with an inevitable loss of information, from a concrete to a less precise abstract setting, until the resulting semantics is computable.

3

A Small Imperative Language

The formal treatment given in the previous chapter is language independent. In this chapter, we look back at the notions introduced in the context of a simple sequential programming language that will be used to illustrate our work throughout the rest of this thesis.

Le traitement formel donné dans le chapitre précédent est indépendant du langage. Dans ce chapitre, nous nous penchons sur les notions introduites dans le contexte d'un simple langage de programmation séquentiel qui sera utilisé pour illustrer notre travail dans le reste de cette thèse.

3.1 A Small Imperative Language

We consider a simple sequential non-deterministic programming language with no procedures, no pointers and no recursion. The variables are statically allocated and the only data type is the set $\mathbb{Z}$ of mathematical integers.

In Chapter 7 we will introduce procedures and recursion, while pointers and machine integers and floats will remain out of the scope of this work.

Language Syntax. In Figure 3.1, we define inductively the syntax of our programming language.

A program *prog* consists of an instruction followed by a unique label $l \in \mathcal{L}$. Another unique label appears within each instruction. An instruction *stmt* is either a `skip` instruction, a variable assignment, a conditional `if` statement, a `while` loop or a sequential composition of instructions.

Arithmetic expressions *aexp* involve variables $X \in \mathcal{X}$, numeric intervals $[a, b]$ and the arithmetic operators $+$, $-$, $*$, $/$ for addition, subtraction, multiplication, and division. Numeric intervals have constant and possibly infinite bounds, and denote a random choice of a number in the interval. This provides

$$
\begin{array}{lcll}
\mathit{aexp} & ::= & X & X \in \mathcal{X} \\
 & | & [i_1, i_2] & i_1 \in \mathbb{Z} \cup \{-\infty\},\ i_2 \in \mathbb{Z} \cup \{+\infty\},\ i_1 \leq i_2 \\
 & | & -\ \mathit{aexp} & \\
 & | & \mathit{aexp} \diamond \mathit{aexp} & \diamond \in \{+, -, *, /\} \\
 & & & \\
\mathit{bexp} & ::= & \texttt{?} & \\
 & | & \texttt{not}\ \mathit{bexp} & \\
 & | & \mathit{bexp}\ \texttt{and}\ \mathit{bexp} & \\
 & | & \mathit{bexp}\ \texttt{or}\ \mathit{bexp} & \\
 & | & \mathit{aexp} \bowtie \mathit{aexpr} & \bowtie \in \{<, \leq, =, \neq\} \\
 & & & \\
\mathit{stmt} & ::= & {}^{l}\texttt{skip} & \\
 & | & {}^{l}X := \mathit{aexp} & l \in \mathcal{L},\ X \in \mathcal{X} \\
 & | & \texttt{if}\ {}^{l}\mathit{bexp}\ \texttt{then}\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt}\ \texttt{fi} & l \in \mathcal{L} \\
 & | & \texttt{while}\ {}^{l}\mathit{bexp}\ \texttt{do}\ \mathit{stmt}\ \texttt{od} & l \in \mathcal{L} \\
 & | & \mathit{stmt}\ \mathit{stmt} & \\
 & & & \\
\mathit{prog} & ::= & \mathit{stmt}\ {}^{l} & l \in \mathcal{L}
\end{array}
$$

Figure 3.1: Syntax of our programming language.

a notion of non-determinism useful to model user input or to approximate arithmetic expressions that cannot be represented exactly in the language. Numeric constants are a particular case of numeric interval. In the following, we often write the constant c for the interval $[c, c]$.

Boolean expressions *bexp* are built by comparing arithmetic expressions, and are combined using the boolean `not`, `and`, and `or` operators. The boolean expression `?` represents a non-deterministic choice and is useful to provide a sequential encoding of concurrent programs by modeling a (possibly, but not necessarily, fair) scheduler. Whenever clear from the context, we frequently abuse notation and use the symbol `?` to also denote the numeric interval $[-\infty, +\infty]$.

3.2 Maximal Trace Semantics

In the following, we instantiate the general definitions of transition system and maximal trace semantics of Section 2.2 with our small imperative language.

Expression Semantics. An *environment* $\rho \colon \mathcal{X} \to \mathbb{Z}$ maps each program variable $X \in \mathcal{X}$ to its value $\rho(X) \in \mathbb{Z}$. Let $\mathcal{E}$ denote the set of all environments.

$$
\begin{array}{lcl}
[\![\, X \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, \rho(X) \,\} \\
[\![\, [a,b] \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, x \mid a \leq x \leq b \,\} \\
[\![\, -\, aexp \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, -x \mid x \in [\![\, aexp \,]\!]\rho \,\} \\
[\![\, aexp_1 \,+\, aexp_2 \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, x + y \mid x \in [\![\, aexp_1 \,]\!],\ y \in [\![\, aexp_2 \,]\!]\rho \,\} \\
[\![\, aexp_1 \,-\, aexp_2 \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, x - y \mid x \in [\![\, aexp_1 \,]\!],\ y \in [\![\, aexp_2 \,]\!]\rho \,\} \\
[\![\, aexp_1 \,*\, aexp_2 \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, x * y \mid x \in [\![\, aexp_1 \,]\!],\ y \in [\![\, aexp_2 \,]\!]\rho \,\} \\
[\![\, aexp_1 \,/\, aexp_2 \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, trnk(x \,/\, y) \mid x \in [\![\, aexp_1 \,]\!],\ y \in [\![\, aexp_2 \,]\!]\rho,\ y \neq 0 \,\}
\end{array}
$$

where $trnk\colon \mathbb{R} \to \mathbb{Z}$ is defined as follows:

$$
trnk(x) \stackrel{\text{def}}{=} \begin{cases} \max\{y \in \mathbb{Z} \mid y \leq x\} & x \geq 0 \\ \min\{y \in \mathbb{Z} \mid y \geq x\} & x < 0 \end{cases}
$$

Figure 3.2: Semantics of arithmetic expressions *aexp*.

The semantics of an arithmetic expression *aexp* is a function $[\![aexp]\!]\colon \mathcal{E} \to \mathcal{P}(\mathbb{Z})$ mapping an environment $\rho \in \mathcal{E}$ to the set of all possible values for the expression *aexp* in the environment. It is presented in Figure 3.2. Note that, the set of values for an expression may contain several elements because of the non-determinism embedded in the expressions. It might also be empty due to undefined results. In fact, this is the case of divisions by zero. The *trnk* function rounds the result of the division towards an integer.

Similarly, the semantics $[\![bexp]\!]\colon \mathcal{E} \to \mathcal{P}(\{\text{true}, \text{false}\})$ of boolean expressions *bexp* maps an environment $\rho \in \mathcal{E}$ to the set of all possible truth values for the expression *bexp* in the environment. It is presented in Figure 3.3. In the following, we write `true` and `false` to represent a boolean expression that is always true and always false, respectively.

Transition Systems. A program *state* $s \in \mathcal{L} \times \mathcal{E}$ is a pair consisting of a label $l \in \mathcal{L}$ and an environment $\rho \in \mathcal{E}$, where the environment defines the values of the program variables at the program control point designated by the label. Let Σ denote the set of all program states.

The *initial* control point $i[\![\, stmt \,]\!] \in \mathcal{L}$ (resp. $i[\![\, prog \,]\!] \in \mathcal{L}$) of an instruction *stmt* (resp. a program *prog*) defines where the execution of the instruction (resp. program) starts, and the *final* control point $f[\![\, stmt \,]\!] \in \mathcal{L}$ (resp. $f[\![\, prog \,]\!] \in \mathcal{L}$) defines where the execution of the instruction *stmt* (resp. program *prog*) ends. The formal definitions are in Figure 3.4 and Figure 3.5.

$$
\begin{array}{lcl}
[\![\, \texttt{?} \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, \text{true}, \text{false} \,\} \\
[\![\, \texttt{not}\ \mathit{bexp} \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, \neg x \mid x \in [\![\, \mathit{bexp} \,]\!]\rho \,\} \\
[\![\, \mathit{bexp}_1\ \texttt{and}\ \mathit{bexp}_2 \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, x \wedge y \mid x \in [\![\, \mathit{bexp}_1 \,]\!],\ y \in [\![\, \mathit{bexp}_2 \,]\!]\rho \,\} \\
[\![\, \mathit{bexp}_1\ \texttt{or}\ \mathit{bexp}_2 \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, x \vee y \mid x \in [\![\, \mathit{bexp}_1 \,]\!],\ y \in [\![\, \mathit{bexp}_2 \,]\!]\rho \,\} \\
[\![\, \mathit{aexp}_1 < \mathit{aexp}_2 \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, x < y \mid x \in [\![\, \mathit{aexp}_1 \,]\!],\ y \in [\![\, \mathit{aexp}_2 \,]\!]\rho \,\} \\
[\![\, \mathit{aexp}_1 \leq \mathit{aexp}_2 \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, x \leq y \mid x \in [\![\, \mathit{aexp}_1 \,]\!],\ y \in [\![\, \mathit{aexp}_2 \,]\!]\rho \,\} \\
[\![\, \mathit{aexp}_1 = \mathit{aexp}_2 \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, x = y \mid x \in [\![\, \mathit{aexp}_1 \,]\!],\ y \in [\![\, \mathit{aexp}_2 \,]\!]\rho \,\} \\
[\![\, \mathit{aexp}_1 \neq \mathit{aexp}_2 \,]\!]\rho & \stackrel{\text{def}}{=} & \{\, x \neq y \mid x \in [\![\, \mathit{aexp}_1 \,]\!],\ y \in [\![\, \mathit{aexp}_2 \,]\!]\rho \,\}
\end{array}
$$

Figure 3.3: Semantics of boolean expressions *bexp*.

$$
\begin{array}{lcll}
\mathit{stmt} & ::= & {}^{l}\texttt{skip} & i[\![\, {}^{l}\texttt{skip} \,]\!] \stackrel{\text{def}}{=} l \\
 & \mid & {}^{l}X := \mathit{aexp} & i[\![\, {}^{l}X := \mathit{aexp} \,]\!] \stackrel{\text{def}}{=} l \\
 & \mid & \texttt{if}\ {}^{l}\mathit{bexp}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \texttt{fi} & \\
 & & \qquad i[\![\, \texttt{if}\ {}^{l}\mathit{bexp}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \texttt{fi} \,]\!] \stackrel{\text{def}}{=} l & \\
 & \mid & \texttt{while}\ {}^{l}\mathit{bexp}\ \texttt{do}\ \mathit{stmt}_1\ \texttt{od} & \\
 & & \qquad i[\![\, \texttt{while}\ {}^{l}\mathit{bexp}\ \texttt{do}\ \mathit{stmt}_1\ \texttt{od} \,]\!] \stackrel{\text{def}}{=} l & \\
 & \mid & \mathit{stmt}_1\ \mathit{stmt}_2 & i[\![\, \mathit{stmt}_1\ \mathit{stmt}_2 \,]\!] \stackrel{\text{def}}{=} i[\![\, \mathit{stmt}_1 \,]\!] \\
\mathit{prog} & ::= & \mathit{stmt}\ {}^{l} & i[\![\, \mathit{stmt}\ {}^{l} \,]\!] \stackrel{\text{def}}{=} i[\![\, \mathit{stmt} \,]\!]
\end{array}
$$

Figure 3.4: Initial control point of instructions *stmt* and programs *prog*.

Example 3.2.1

Let us consider the following program:

$$
\begin{array}{l}
{}^{1}x := \texttt{?} \\
\texttt{while}\ {}^{2}(1 < x)\ \texttt{do} \\
\quad {}^{3}x := x - 1 \\
\texttt{od}^{4}
\end{array}
$$

We have the following final program control points:

$$
\begin{array}{rcl}
f[\![\, {}^{1}x := \texttt{?}\ \texttt{while}\ {}^{2}(1 < x)\ \texttt{do}\ {}^{3}x := x - 1\ \texttt{od}^{4} \,]\!] & = & \mathbf{4} \\
f[\![\, {}^{1}x := \texttt{?}\ \texttt{while}\ {}^{2}(1 < x)\ \texttt{do}\ {}^{3}x := x - 1\ \texttt{od} \,]\!] & = & \mathbf{4} \\
f[\![\, {}^{1}x := \texttt{?} \,]\!] & = & \mathbf{2} \\
f[\![\, \texttt{while}\ {}^{2}(1 < x)\ \texttt{do}\ {}^{3}x := x - 1\ \texttt{od} \,]\!] & = & \mathbf{4} \\
f[\![\, {}^{3}x := x - 1 \,]\!] & = & \mathbf{2}
\end{array}
$$

Note that, the final control point $f[\![\, \mathit{stmt} \,]\!] \in \mathcal{L}$ of an instruction *stmt* does not belong to the set of control points appearing in the instruction.

$$
\begin{array}{lcll}
\mathit{stmt} & ::= & {}^{l}\texttt{skip} & f[\![\, {}^{l}\texttt{skip} \,]\!] \stackrel{\text{def}}{=} f[\![\, \mathit{stmt} \,]\!] \\
 & | & {}^{l}X := \mathit{aexp} & f[\![\, {}^{l}X := \mathit{aexp} \,]\!] \stackrel{\text{def}}{=} f[\![\, \mathit{stmt} \,]\!] \\
 & | & \texttt{if}\ {}^{l}\mathit{bexp}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \texttt{fi} & \\
 & & f[\![\, \texttt{if}\ {}^{l}\mathit{bexp}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \texttt{fi} \,]\!] \stackrel{\text{def}}{=} f[\![\, \mathit{stmt} \,]\!] & \\
 & & f[\![\, \mathit{stmt}_1 \,]\!] \stackrel{\text{def}}{=} f[\![\, \mathit{stmt} \,]\!] & \\
 & & f[\![\, \mathit{stmt}_2 \,]\!] \stackrel{\text{def}}{=} f[\![\, \mathit{stmt} \,]\!] & \\
 & | & \texttt{while}\ {}^{l}\mathit{bexp}\ \texttt{do}\ \mathit{stmt}_1\ \texttt{od} & \\
 & & f[\![\, \texttt{while}\ {}^{l}\mathit{bexp}\ \texttt{do}\ \mathit{stmt}_1\ \texttt{od} \,]\!] \stackrel{\text{def}}{=} f[\![\, \mathit{stmt} \,]\!] & \\
 & & f[\![\, \mathit{stmt}_1 \,]\!] \stackrel{\text{def}}{=} l & \\
 & | & \mathit{stmt}_1\ \mathit{stmt}_2 & f[\![\, \mathit{stmt}_1\ \mathit{stmt}_2 \,]\!] \stackrel{\text{def}}{=} f[\![\, \mathit{stmt} \,]\!] \\
 & & f[\![\, \mathit{stmt}_1 \,]\!] \stackrel{\text{def}}{=} i[\![\, \mathit{stmt}_2 \,]\!] & \\
 & & f[\![\, \mathit{stmt}_2 \,]\!] \stackrel{\text{def}}{=} f[\![\, \mathit{stmt} \,]\!] & \\
\mathit{prog} & ::= & \mathit{stmt}\ {}^{l} & f[\![\, \mathit{stmt}\ {}^{l} \,]\!] \stackrel{\text{def}}{=} l
\end{array}
$$

Figure 3.5: Final control point of instructions *stmt* and programs *prog*.

A program execution starts at its initial program control point with any possible value for the program variables. Thus, the set of initial states of a program *prog* is $\mathcal{I} \stackrel{\text{def}}{=} \{\langle i[\![\, \mathit{prog} \,]\!], \rho\rangle \mid \rho \in \mathcal{E}\}$. It is sometimes useful to assume that a set $E \subseteq \mathcal{E}$ of initial environments is given, in which case the program initial states correspond to the initial program control point paired with any initial environment: $\mathcal{I} \stackrel{\text{def}}{=} \{\langle i[\![\, \mathit{prog} \,]\!], \rho\rangle \mid \rho \in E\}$. The set of final states of a program *prog* is $\mathcal{Q} \stackrel{\text{def}}{=} \{\langle f[\![\, \mathit{prog} \,]\!], \rho\rangle \mid \rho \in \mathcal{E}\}$.

Remark 3.2.2 In Section 2.2 we defined the final states to have no successors with respect to the transition relation, meaning that the program halts: $\Omega \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma : \langle s, s'\rangle \notin \tau\}$. This is the case when the program successfully terminates by reaching its final label, or when a *run-time error* occurs. For the sake of simplicity, the definition of program final states given in this section ignores possible run-time errors silently halting the program. ■

We now define the transition relation $\tau \in \Sigma \times \Sigma$. In particular, in Figure 3.6, we define the transition semantics $\tau[\![\, \mathit{stmt} \,]\!] \in \Sigma \times \Sigma$ of each program instruction *stmt*. Given an environment $\rho \in \mathcal{E}$, a program variable $X \in \mathcal{X}$ and a value $v \in \mathbb{Z}$, we denote by $\rho[X \leftarrow v]$ the environment obtained by writing the value v into the variable X in the environment ρ:

$$
\rho[X \leftarrow v](x) = \begin{cases} v & x = X \\ \rho(x) & x \neq X \end{cases}
$$

$$
\begin{aligned}
&\tau[\![\, {}^{l}\texttt{skip} \,]\!] \stackrel{\text{def}}{=} \{\langle l, \rho\rangle \to \langle f[\![\, {}^{l}\texttt{skip} \,]\!], \rho\rangle \mid \rho \in \mathcal{E}\} \\
&\tau[\![\, {}^{l}X := aexp \,]\!] \stackrel{\text{def}}{=} \\
&\qquad \{\langle l, \rho\rangle \to \langle f[\![\, {}^{l}X := aexp \,]\!], \rho[X \leftarrow v]\rangle \mid \rho \in \mathcal{E},\ v \in [\![aexp]\!]\rho\} \\
&\tau[\![\, \texttt{if}\ {}^{l}bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi} \,]\!] \stackrel{\text{def}}{=} \\
&\qquad \{\langle l, \rho\rangle \to \langle i[\![\, stmt_1 \,]\!], \rho\rangle \mid \rho \in \mathcal{E},\ \text{true} \in [\![bexp]\!]\rho\} \cup \tau[\![\, stmt_1 \,]\!] \cup \\
&\qquad \{\langle l, \rho\rangle \to \langle i[\![\, stmt_2 \,]\!], \rho\rangle \mid \rho \in \mathcal{E},\ \text{false} \in [\![bexp]\!]\rho\} \cup \tau[\![\, stmt_2 \,]\!] \\
&\tau[\![\, \texttt{while}\ {}^{l}bexp\ \texttt{do}\ stmt\ \texttt{od} \,]\!] \stackrel{\text{def}}{=} \\
&\qquad \{\langle l, \rho\rangle \to \langle i[\![\, stmt \,]\!], \rho\rangle \mid \rho \in \mathcal{E},\ \text{true} \in [\![bexp]\!]\rho\} \cup \tau[\![\, stmt \,]\!] \cup \\
&\qquad \{\langle l, \rho\rangle \to \langle f[\![\, \texttt{while}\ {}^{l}bexp\ \texttt{do}\ stmt\ \texttt{od} \,]\!], \rho\rangle \mid \rho \in \mathcal{E},\ \text{false} \in [\![bexp]\!]\rho\} \\
&\tau[\![\, stmt_1\ stmt_2 \,]\!] \stackrel{\text{def}}{=} \tau[\![\, stmt_1 \,]\!] \cup \tau[\![\, stmt_2 \,]\!]
\end{aligned}
$$

Figure 3.6: Transition semantics of instructions *stmt*.

The semantics of a `skip` instruction simply moves control from the initial label of the instruction to its final label. The execution of a variable assignment ${}^{l}X := aexp$ moves control from the initial label of the instruction to its final label, and modifies the current environment in order to assign any of the possible values of *aexp* to the variable X. The semantics of a conditional statement `if` ${}^{l}bexp$ `then` $stmt_1$ `else` $stmt_2$ `fi` moves control from the initial label of the instruction to the initial label of $stmt_1$, if true is a possible value for *bexp*, and to the initial label of $stmt_2$, if false is a possible value for *bexp*; then, $stmt_1$ and $stmt_2$ are executed. Similarly, the execution of a while statement `while` ${}^{l}bexp$ `do` *stmt* `od` moves control from the initial label of the instruction to its final label, if false is a possible value for *bexp*, and to the initial label of $stmt_1$, if true is a possible value for *bexp*; then *stmt* is executed. Note that, control moves from the end of *stmt* to the initial label l of the `while` loop, since l is the final label of *stmt* (cf. Figure 3.5). Finally, the semantics of the sequential combination of instructions $stmt_1$ $stmt_2$ executes $stmt_1$ and $stmt_2$. Note that, control moves from the end of $stmt_1$ to the beginning of $stmt_2$, since the final label of $stmt_1$ is the initial label of $stmt_2$ (cf. Figure 3.5).

The transition relation $\tau \in \Sigma \times \Sigma$ of a program *prog* is defined by the semantics $\tau[\![\, prog \,]\!] \in \Sigma \times \Sigma$ of the program as $\tau[\![\, prog \,]\!] = \tau[\![\, stmt\ {}^{l} \,]\!] \stackrel{\text{def}}{=} \tau[\![\, stmt \,]\!]$.

Example 3.2.3

Let us consider again the program of Example 3.2.1:

$$
\begin{array}{l}
{}^{1}x := \texttt{?} \\
\texttt{while}\ {}^{2}(1 < x)\ \texttt{do} \\
\quad {}^{3}x := x - 1 \\
\texttt{od}^{4}
\end{array}
$$

The set of program environments $\mathcal{E}$ contains functions $\rho\colon \{x\} \to \mathbb{Z}$ mapping the program variable x to any possible value $\rho(x) \in \mathbb{Z}$. The set of program states $\Sigma \stackrel{\text{def}}{=} \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}\} \times \mathcal{E}$ consists of all pairs of labels and environments; the initial states are $\mathcal{I} \stackrel{\text{def}}{=} \{\langle \mathbf{1}, \rho\rangle \mid \rho \in \mathcal{E}\}$. The program transition relation $\tau \in \Sigma \times \Sigma$ is defined as follows:

$$\begin{aligned}\tau \stackrel{\text{def}}{=} & \{\langle \mathbf{1}, \rho\rangle \to \langle \mathbf{2}, \rho[x \leftarrow v]\rangle \mid \rho \in \mathcal{E} \wedge v \in \mathbb{Z}\}\\ & \cup \{\langle \mathbf{2}, \rho\rangle \to \langle \mathbf{3}, \rho\rangle \mid \rho \in \mathcal{E} \wedge \text{true} \in [\![1 < x]\!]\rho\}\\ & \cup \{\langle \mathbf{3}, \rho\rangle \to \langle \mathbf{2}, \rho[x \leftarrow \rho(x) - 1]\rangle \mid \rho \in \mathcal{E}\}\\ & \cup \{\langle \mathbf{2}, \rho\rangle \to \langle \mathbf{4}, \rho\rangle \mid \rho \in \mathcal{E} \wedge \text{false} \in [\![1 < x]\!]\rho\}\end{aligned}$$

The set of final states is $\mathcal{Q} \stackrel{\text{def}}{=} \{\langle \mathbf{4}, \rho\rangle \mid \rho \in \mathcal{E}\}$. Note that, the definition of final states assumes all possible values for the program variable x although when executing the program we have $x \leq 1$ on program exit.

Maximal Trace Semantics. In the following, we provide a structural definition of the fixpoint maximal trace semantics $\tau^{+\infty} \in \Sigma^{+\infty}$ (Equation 2.2.5) by induction on the syntax of programs.

We recall that a program trace is a non-empty sequence of program states in Σ determined by the program transition relation τ.

Example 3.2.4

Let us consider again the program of Example 3.2.3:

```
1 x := ?
while 2 (1 < x) do
  3 x := x - 1
od 4
```

We write $\{\langle x, v\rangle\}$ to denote the environment $\rho\colon \{x\} \to \mathbb{Z}$ mapping the program variable x to the value $v \in \mathbb{Z}$. The following sequence of program states:

$$\langle \mathbf{1}, \{\langle x, 42\rangle\}\rangle\langle \mathbf{2}, \{\langle x, 2\rangle\}\rangle\langle \mathbf{3}, \{\langle x, 2\rangle\}\rangle\langle \mathbf{2}, \{\langle x, 1\rangle\}\rangle\langle \mathbf{4}, \{\langle x, 1\rangle\}\rangle$$

is an example of program trace determined by the transition relation τ.

We only consider program traces starting from the set $\mathcal{I}$ of initial states of a program *prog*. Accordingly, in Figure 3.7 we define the trace semantics $\tau^{+\infty}[\![\, \mathit{stmt} \,]\!] \in \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+\infty})$ of each program instruction *stmt*. Analogously to Equation 2.2.5, the program traces are built *backwards*: each

$$\tau^{+\infty}[\![\, {}^{l}\texttt{skip} \,]\!]T \stackrel{\text{def}}{=} \left\{ \langle l, \rho\rangle\langle f[\![\, {}^{l}\texttt{skip} \,]\!], \rho\rangle\sigma \;\middle|\; \begin{array}{c} \rho \in \mathcal{E},\ \sigma \in \Sigma^{*\infty} \\ \langle f[\![\, {}^{l}\texttt{skip} \,]\!], \rho\rangle\sigma \in T \end{array} \right\}$$

$$\tau^{+\infty}[\![\, {}^{l}X := aexp \,]\!]T \stackrel{\text{def}}{=}$$
$$\left\{ \langle l, \rho\rangle\langle f[\![\, {}^{l}X := aexp \,]\!], \rho[X \leftarrow v]\rangle\sigma \;\middle|\; \begin{array}{c} \rho \in \mathcal{E},\ v \in [\![aexp]\!]\rho,\ \sigma \in \Sigma^{*\infty} \\ \langle f[\![\, {}^{l}X := aexp \,]\!], \rho[X \leftarrow v]\rangle\sigma \in T \end{array} \right\}$$

$$\tau^{+\infty}[\![\, \texttt{if}\ {}^{l}bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi} \,]\!]T \stackrel{\text{def}}{=}$$
$$\left\{ \langle l, \rho\rangle\langle i[\![\, stmt_1 \,]\!], \rho\rangle\sigma \;\middle|\; \begin{array}{c} \rho \in \mathcal{E},\ \text{true} \in [\![bexp]\!]\rho,\ \sigma \in \Sigma^{*\infty} \\ \langle i[\![\, stmt_1 \,]\!], \rho\rangle\sigma \in \tau^{+\infty}[\![\, stmt_1 \,]\!]T \end{array} \right\} \cup$$
$$\left\{ \langle l, \rho\rangle\langle i[\![\, stmt_2 \,]\!], \rho\rangle\sigma \;\middle|\; \begin{array}{c} \rho \in \mathcal{E},\ \text{false} \in [\![bexp]\!]\rho,\ \sigma \in \Sigma^{*\infty} \\ \langle i[\![\, stmt_2 \,]\!], \rho\rangle\sigma \in \tau^{+\infty}[\![\, stmt_2 \,]\!]T \end{array} \right\}$$

$$\tau^{+\infty}[\![\, \texttt{while}\ {}^{l}bexp\ \texttt{do}\ stmt\ \texttt{od} \,]\!]T \stackrel{\text{def}}{=} \text{lfp}^{\sqsubseteq}\ \phi^{+\infty}$$
$$\phi^{+\infty}(X) \stackrel{\text{def}}{=} \left\{ \langle l, \rho\rangle\langle i[\![\, stmt \,]\!], \rho\rangle\sigma \;\middle|\; \begin{array}{c} \rho \in \mathcal{E},\ \text{true} \in [\![bexp]\!]\rho,\ \sigma \in \Sigma^{*\infty} \\ \langle i[\![\, stmt \,]\!], \rho\rangle\sigma \in \tau^{+\infty}[\![\, stmt \,]\!]X \end{array} \right\} \cup$$
$$\left\{ \langle l, \rho\rangle\langle f[\![\, \texttt{while}\ \cdots\ \texttt{od} \,]\!], \rho\rangle\sigma \;\middle|\; \begin{array}{c} \rho \in \mathcal{E},\ \text{false} \in [\![bexp]\!]\rho,\ \sigma \in \Sigma^{*\infty} \\ \langle f[\![\, \texttt{while}\ \cdots\ \texttt{od} \,]\!], \rho\rangle\sigma \in T \end{array} \right\}$$

$$\tau^{+\infty}[\![\, stmt_1\ stmt_2 \,]\!]T \stackrel{\text{def}}{=} \tau^{+\infty}[\![\, stmt_1 \,]\!](\tau^{+\infty}[\![\, stmt_2 \,]\!]T)$$

Figure 3.7: Maximal trace semantics of instructions *stmt*.

function $\tau^{+\infty}[\![\, stmt \,]\!] \in \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+\infty})$ takes as input a set of traces starting with the final label of the instruction *stmt* and outputs a set of traces starting with the initial label of *stmt*.

The trace semantics of a `skip` instruction takes as input a set $T \in \mathcal{P}(\Sigma^{+\infty})$ of traces starting with environments paired with the final label of the instruction and, according to the transition relation semantics of the instruction, it prepends to them the same environments paired with its initial label.

The trace semantics of a variable assignment ${}^{l}X := aexp$ takes as input a set $T \in \mathcal{P}(\Sigma^{+\infty})$ of traces starting with the final label of the instruction and it prepends to them all program states with its initial label that are allowed by the transition relation semantics of the instruction (cf. Figure 3.6).

Similarly, given a conditional instruction `if` ${}^{l}bexp$ `then` $stmt_1$ `else` $stmt_2$ `fi`, its trace semantics prepends all program states with its initial label that are allowed by its transition relation semantics to the traces starting with the initial labels of $stmt_1$ and $stmt_2$; these are obtained by means of the trace semantics of $stmt_2$ and $stmt_1$ taking as input a set $T \in \mathcal{P}(\Sigma^{+\infty})$ of traces starting with the final label of the conditional instruction.

The trace semantics of a loop instruction `while` ${}^{l}bexp$ `do` *stmt* `od` is defined as the least fixpoint of the function $\phi^{+\infty} \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+\infty})$ within the complete lattice $\langle \mathcal{P}(\Sigma^{+\infty}), \sqsubseteq, \sqcup, \sqcap, \Sigma^{\omega}, \Sigma^{+} \rangle$, analogously to Equation 2.2.5. The

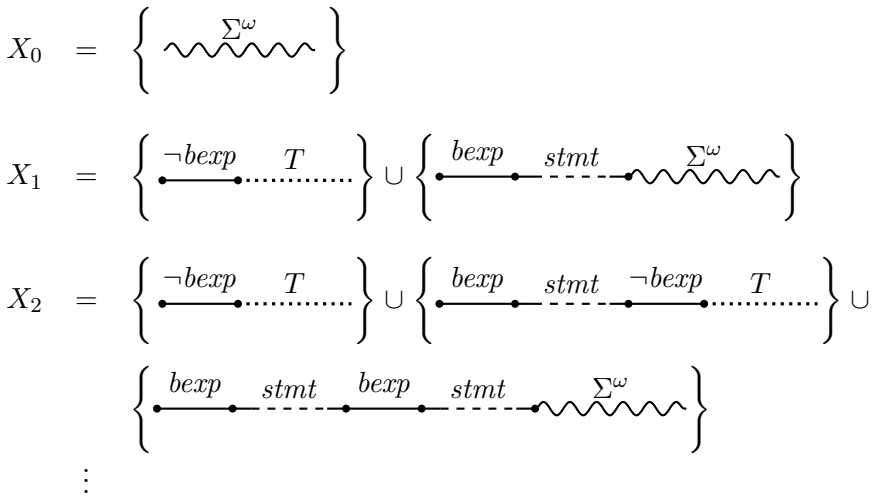


Figure 3.8: Fixpoint iterates of the trace semantics of a loop instruction.

iteration sequence, starting from all infinite *sequences* $\Sigma^\omega \in \mathcal{P}(\Sigma^{+\infty})$, builds the set of program *traces* that consist of an infinite number of iterations within the loop, and it prepends a finite number of iterations within the loop to an input set $T \in \mathcal{P}(\Sigma^{+\infty})$ of traces starting with the final label of the loop instruction. In Figure 3.7, we have shortened $f[\![\, \texttt{while}\ ^l bexp\ \texttt{do}\ stmt\ \texttt{od} \,]\!]$ by means of $f[\![\, \texttt{while}\ \cdots\ \texttt{od} \,]\!]$. In particular, the function $\phi^{+\infty} \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+\infty})$ takes as input a set $X \in \mathcal{P}(\Sigma^{+\infty})$ of traces: initially, X is the set of all infinite sequences Σ^ω and, at each iteration, the function prepends all program states whose label is the initial label of the loop instruction, to the traces belonging to the input set T, and to the traces that are obtained by means of the trace semantics of the loop body *stmt* from the set X. In this way, after the ith iteration, the set X contains the program traces starting at the initial label of the loop instruction whose prefix consist of from zero up to $i-1$ iterations within the loop and whose suffix is a trace in T, and the *sequences* whose prefix are program *traces* which consist of $i-1$ iterations within the loop. With a slight abuse of notation, we depict the fixpoints iterates in Figure 3.8.

Finally, the trace semantics of the sequential combination of instructions $stmt_1\ stmt_2$ takes as input a set $T \in \mathcal{P}(\Sigma^{+\infty})$ of traces starting with the final label of $stmt_2$, determines from T the set of traces $\tau^{+\infty}[\![\, stmt_2 \,]\!]T$ belonging to the trace semantics of $stmt_2$, and outputs the set of traces determined by the trace semantics of $stmt_1$ from $\tau^{+\infty}[\![\, stmt_2 \,]\!]T$.

The maximal trace semantics $\tau^{+\infty}[\![\, prog \,]\!] \in \mathcal{P}(\Sigma^{+\infty})$ of a program *prog* is the maximal trace semantics $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ defined in Equation 2.2.5 restricted to the traces starting from the program initial states $\mathcal{I}$, and it is defined taking as input the set of program final states $\mathcal{Q}$:

Definition 3.2.5 (Maximal Trace Semantics) *The* maximal trace semantics $\tau^{+\infty}[\![\, prog \,]\!] \in \mathcal{P}(\Sigma^{+\infty})$ *of a program prog is:*

$$\tau^{+\infty}[\![\, prog \,]\!] = \tau^{+\infty}[\![\, stmt^{\,l} \,]\!] \stackrel{def}{=} \tau^{+\infty}[\![\, stmt \,]\!]\mathcal{Q} \tag{3.2.1}$$

where the trace semantics $\tau^{+\infty}[\![\, stmt \,]\!] \in \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+\infty})$ *of each program instruction stmt is defined in Figure 3.7.*

Note that, as pointed out in Remark 3.2.2, possible run-time errors are ignored. More specifically, all traces containing run-time errors are discarded and do not belong to the maximal trace semantics of a program *prog*.

In the following, we often simply write $\tau^{+\infty}$ instead of $\tau^{+\infty}[\![\, prog \,]\!]$.

3.3 Invariance Semantics

The reachable state semantics $\tau_{\mathrm{R}} \in \mathcal{P}(\Sigma)$ introduced by Example 2.2.18 abstracts the maximal trace semantics $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ by collecting the possible values of the program variables at each program point and disregarding other information. In the following, we provide an isomorphic *invariance semantics* defined by induction on the syntax of our small language.

Note that the complete lattice $\langle \mathcal{P}(\Sigma), \subseteq, \cup, \cap, \emptyset, \Sigma \rangle$ on which the reachable state semantics is defined is isomorphic by partitioning with respect to the program control points and by pointwise lifting (cf. Equation 2.1.1) to $\langle \mathcal{L} \to \mathcal{P}(\mathcal{E}), \dot{\subseteq}, \dot{\cup}, \dot{\cap}, \lambda l.\, \emptyset, \lambda l.\, \mathcal{E} \rangle$. This can be formalized as a Galois connection $\langle \mathcal{P}(\Sigma), \subseteq \rangle \underset{\alpha^{\mathrm{I}}}{\overset{\gamma^{\mathrm{I}}}{\leftrightarrows}} \langle \mathcal{L} \to \mathcal{P}(\mathcal{E}), \dot{\subseteq} \rangle$ where the abstraction $\alpha^{\mathrm{I}} \colon \mathcal{P}(\Sigma) \to (\mathcal{L} \to \mathcal{P}(\mathcal{E}))$ and the concretization $\gamma^{\mathrm{I}} \colon (\mathcal{L} \to \mathcal{P}(\mathcal{E})) \to \mathcal{P}(\Sigma)$ are defined as follows:

$$\begin{aligned} \alpha^{\mathrm{I}}(R) &\stackrel{\mathrm{def}}{=} \lambda l \in \mathcal{L}.\ \{\rho \in \mathcal{E} \mid \langle l, \rho \rangle \in R\} \\ \gamma^{\mathrm{I}}(I) &\stackrel{\mathrm{def}}{=} \{\langle l, \rho \rangle \in \Sigma \mid \rho \in I(l)\} \end{aligned} \tag{3.3.1}$$

This abstraction, from now on, is called *invariance abstraction*. In this form, the reachable state semantics $\tau_{\mathrm{R}} \in \mathcal{P}(\Sigma)$ becomes an invariance semantics $\tau_{\mathrm{I}} \in \mathcal{L} \to \mathcal{P}(\mathcal{E})$ associating to each program control point $l \in \mathcal{L}$ an *invariant* $E \in \mathcal{P}(\mathcal{E})$ which collects the set of possible program environments for each

$$
\begin{aligned}
&\tau_{\text{post}}[\![\, {}^{l}\texttt{skip} \,]\!]E \stackrel{\text{def}}{=} E \\
&\tau_{\text{post}}[\![\, {}^{l}X := aexp \,]\!]E \stackrel{\text{def}}{=} \{\rho[X \leftarrow v] \in \mathcal{E} \mid \rho \in E,\, v \in [\![aexp]\!]\rho\} \\
&\tau_{\text{post}}[\![\, \texttt{if}\ {}^{l}bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi} \,]\!]E \stackrel{\text{def}}{=} \\
&\qquad \tau_{\text{post}}[\![stmt_1]\!]\{\rho \in E \mid \text{true} \in [\![bexp]\!]\rho\} \cup \\
&\qquad \tau_{\text{post}}[\![stmt_2]\!]\{\rho \in E \mid \text{false} \in [\![bexp]\!]\rho\} \\
&\tau_{\text{post}}[\![\, \texttt{while}\ {}^{l}bexp\ \texttt{do}\ stmt\ \texttt{od} \,]\!]E \stackrel{\text{def}}{=} \{\rho \in \text{lfp}\ \phi_{\text{post}} \mid \text{false} \in [\![bexp]\!]\rho\} \\
&\qquad \phi_{\text{post}}(X) \stackrel{\text{def}}{=} E \cup \tau_{\text{post}}[\![stmt]\!]\{\rho \in X \mid \text{true} \in [\![bexp]\!]\rho\} \\
&\tau_{\text{post}}[\![\, stmt_1\ \ stmt_2 \,]\!]E \stackrel{\text{def}}{=} \tau_{\text{post}}[\![\, stmt_2 \,]\!](\tau_{\text{post}}[\![\, stmt_1 \,]\!]E)
\end{aligned}
$$

Figure 3.9: Invariance semantics of instructions *stmt*.

time the program execution reaches that program control point. We can now define the invariance semantics pointwise within the power set of environments.

In Figure 3.9, for each program instruction *stmt*, we define its postcondition semantics $\tau_{\text{post}}[\![\, stmt \,]\!] \colon \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$. Analogously to the states belonging to the reachable state semantics in Example 2.2.18, the environment belonging to the postcondition semantics are built *forward*: each function $\tau_{\text{post}}[\![\, stmt \,]\!] \colon \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ takes as input a set of environments and outputs the set of possible environments at the final control point of the instruction.

The postcondition semantics $\tau_{\text{post}}[\![\, prog \,]\!] \in \mathcal{P}(\mathcal{E})$ of a program *prog* outputs the set of possible program environments at the final program control point $f[\![\, prog \,]\!] \in \mathcal{L}$. It is defined from the set of all program environments $\mathcal{E}$ as:

Definition 3.3.1 (Postcondition Semantics) *Given a program prog, its* postcondition semantics $\tau_{\text{post}}[\![\, prog \,]\!] \in \mathcal{P}(\mathcal{E})$ *is:*

$$\tau_{\text{post}}[\![\, prog \,]\!] = \tau_{\text{post}}[\![\, stmt\ {}^{l} \,]\!] \stackrel{def}{=} \tau_{\text{post}}[\![\, stmt \,]\!]\mathcal{E} \tag{3.3.2}$$

where the postcondition semantics $\tau_{\text{post}}[\![\, stmt \,]\!] \in \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ *of each program instruction stmt is defined in Figure 3.9.*

In this way, we can collect the set of possible program environments at each program control point. At the initial control point, the possible environments are all program environments:

$$\tau_{\text{I}}(i[\![\, prog \,]\!]) \stackrel{\text{def}}{=} \mathcal{E}.$$

Then, for each program instruction *stmt*, the set $\tau_{\text{I}}(f[\![\, stmt \,]\!]) \in \mathcal{P}(\mathcal{E})$ of possible environments at its final control point is defined by the postcondition

semantics (cf. Figure 3.9), taking as input the set $E \in \mathcal{P}(\mathcal{E})$ of possible environments at the final control point of the instruction preceding *stmt* (or all program environments in case *stmt* is the first instruction of the program):

$$\tau_{\mathrm{I}}(f[\![\, stmt \,]\!]) \stackrel{\text{def}}{=} \tau_{\text{post}}[\![\, stmt \,]\!]E.$$

In case of a `while` loop instruction, the set $\tau_{\mathrm{I}}(i[\![\, \texttt{while}\ ^{l}bexp\ \texttt{do}\ stmt\ \texttt{od} \,]\!]) \in \mathcal{P}(\mathcal{E})$ of possible environments at its initial control point is defined as the least fixpoint greater than the input set E of the function $\phi_{\text{post}} \colon \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ defined in Figure 3.9:

$$\tau_{\mathrm{I}}(i[\![\, \texttt{while}\ ^{l}bexp\ \texttt{do}\ stmt\ \texttt{od} \,]\!]) \stackrel{\text{def}}{=} \mathrm{lfp}_E\ \phi_{\text{post}}$$

At each iteration, the function $\phi_{\text{post}} \colon \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ accumulates the possible environments after another loop iteration from a given set of environments $X \in \mathcal{P}(\mathcal{E})$. The set $\tau_{\mathrm{I}}(i[\![\, stmt \,]\!]) \in \mathcal{P}(\mathcal{E})$ of possible environments at the initial control point of the loop body is the set of possible environments at the initial control point of the loop that satisfy the boolean expression *bexp*:

$$\tau_{\mathrm{I}}(i[\![\, stmt \,]\!]) \stackrel{\text{def}}{=} \{\rho \in \tau_{\mathrm{I}}(i[\![\, \texttt{while}\ ^{l}bexp\ \texttt{do}\ stmt\ \texttt{od} \,]\!]) \mid \mathrm{true} \in [\![bexp]\!]\rho\}$$

Example 3.3.2

Let us consider again the program of Example 3.2.4:

$$\begin{array}{l} {}^{1}x := \texttt{?} \\ \texttt{while}\ {}^{2}(1 < x)\ \texttt{do} \\ \quad {}^{3}x := x - 1 \\ \texttt{od}^{4} \end{array}$$

The following are the set of possible program environments collected by the invariance semantics at each program control point:

$$\begin{array}{lcl} \tau_{\mathrm{I}}(\mathbf{1}) & = & \mathcal{E} \\ \tau_{\mathrm{I}}(\mathbf{2}) & = & \mathrm{lfp}_{\tau_{\text{post}}[\![\, {}^{1}x:=\texttt{?} \,]\!]\tau_{\mathrm{I}}(\mathbf{1})}\ \phi_{\text{post}} \ = \ \mathcal{E} \\ \tau_{\mathrm{I}}(\mathbf{3}) & = & \{\rho \in \tau_{\mathrm{I}}(\mathbf{2}) \mid \mathrm{true} \in [\![1 < x]\!]\rho\} \ = \ \{\rho \in \mathcal{E} \mid 2 \leq \rho(x)\} \\ \tau_{\mathrm{I}}(\mathbf{4}) & = & \{\rho \in \tau_{\mathrm{I}}(\mathbf{2}) \mid \mathrm{false} \in [\![1 < x]\!]\rho\} \ = \ \{\rho \in \mathcal{E} \mid \rho(x) \leq 1\} \end{array}$$

where $\phi_{\text{post}}(X) = X \cup \tau_{\text{post}}[\![\, {}^{3}x := x - 1 \,]\!]\{\rho \in X \mid \mathrm{true} \in [\![1 < x]\!]\rho\}$.

In the next section, we present further abstractions of the invariance semantics by means of numerical abstract domains.

3.4 Numerical Abstract Domains

We consider concretization-based abstractions of the following form:

$$\langle \mathcal{P}(\mathcal{E}), \subseteq \rangle \xleftarrow{\gamma_{\mathrm{D}}} \langle \mathcal{D}, \sqsubseteq_{\mathrm{D}} \rangle$$

which provide, for each program control point $l \in \mathcal{L}$, a sound decidable abstraction $\tau_{\mathrm{I}}^{\natural}(l) \in \mathcal{D}$ of the invariance semantics $\tau_{\mathrm{I}}(l) \in \mathcal{P}(\mathcal{E})$ (cf. Section 3.3). We have $\tau_{\mathrm{I}}(l) \subseteq \gamma_{\mathrm{D}}(\tau_{\mathrm{I}}^{\natural}(l))$, meaning that the abstract invariance semantics $\tau_{\mathrm{I}}^{\natural}(l)$ is an over-approximation of the set of environments $\tau_{\mathrm{I}}(l)$.

In some cases, there also exists a Galois connection (cf. Remark 2.2.19):

$$\langle \mathcal{P}(\mathcal{E}), \subseteq \rangle \underset{\alpha_{\mathrm{D}}}{\overset{\gamma_{\mathrm{D}}}{\leftrightarrows}} \langle \mathcal{D}, \sqsubseteq_{\mathrm{D}} \rangle$$

By pointwise lifting (cf. Equation 2.1.1) we obtain the following:

$$\langle \mathcal{L} \to \mathcal{P}(\mathcal{E}), \dot{\subseteq} \rangle \xleftarrow{\dot{\gamma_{\mathrm{D}}}} \langle \mathcal{L} \to \mathcal{D}, \dot{\sqsubseteq}_{\mathrm{D}} \rangle$$

or, when an abstraction function $\alpha_{\mathrm{D}} \colon \mathcal{P}(\mathcal{E}) \to \mathcal{D}$ also exists:

$$\langle \mathcal{L} \to \mathcal{P}(\mathcal{E}), \dot{\subseteq} \rangle \underset{\dot{\alpha_{\mathrm{D}}}}{\overset{\dot{\gamma_{\mathrm{D}}}}{\leftrightarrows}} \langle \mathcal{L} \to \mathcal{D}, \dot{\sqsubseteq}_{\mathrm{D}} \rangle$$

which provides a (concretization-based) abstraction of the invariance semantics $\tau_{\mathrm{I}} \in \mathcal{L} \to \mathcal{P}(\mathcal{E})$ by partitioning with respect to the program control points. No approximation is made on $\mathcal{L}$. On the other hand, each program control point $l \in \mathcal{L}$ is associated with an element $d \in \mathcal{D}$.

Numerical Abstract Domains. The abstract domain $\langle \mathcal{D}, \sqsubseteq_{\mathrm{D}} \rangle$ is a *numerical abstract domain* and it obeys the following signature:

Definition 3.4.1 (Numerical Abstract Domain) *A* numerical abstract domain *is characterized by a choice of:*

- *a set $\mathcal{D}$ whose elements are computer-representable;*
- *a partial order $\sqsubseteq_D$ together with an effective algorithm to implement it;*
- *a concretization-based abstraction $\langle \mathcal{P}(\mathcal{E}), \subseteq \rangle \xleftarrow{\gamma_D} \langle \mathcal{D}, \sqsubseteq_D \rangle$ or, when possible, a Galois connection $\langle \mathcal{P}(\mathcal{E}), \subseteq \rangle \underset{\alpha_D}{\overset{\gamma_D}{\leftrightarrows}} \langle \mathcal{D}, \sqsubseteq_D \rangle$;*
- *a least element $\bot_D \in \mathcal{D}$ such that $\gamma_D(\bot_D) = \emptyset$;*

- *a greatest element* $\top_D \in \mathcal{D}$ *such that* $\gamma_D(\top_D) = \mathcal{E}$*;*
- *a sound binary operator* $\sqcup_D$ *such that* $\gamma_D(d_1) \cup \gamma_D(d_2) \subseteq \gamma_D(d_1 \sqcup_D d_2)$*;*
- *a sound binary operator* $\sqcap_D$ *such that* $\gamma_D(d_1) \cap \gamma_D(d_2) \subseteq \gamma_D(d_1 \sqcap_D d_2)$*;*
- *a sound unary operator* $\text{ASSIGN}_D[\![X := aexp]\!] : \mathcal{D} \to \mathcal{D}$*, together with an effective algorithm to handle a variable assignment* ${}^l X := aexp$*, such that* $\tau_I[\![{}^l X := aexp]\!]\gamma_D(d) \subseteq \gamma_D(\text{ASSIGN}_D[\![X := aexp]\!]d)$*;*
- *a sound backward operator* $\text{B-ASSIGN}_D[\![X := aexp]\!] : \mathcal{D} \to \mathcal{D} \to \mathcal{D}$*, together with an effective algorithm to refine an element* $d \in \mathcal{D}$ *given the result* $r \in \mathcal{D}$ *of the assignment* ${}^l X := aexp$*, such that* $\{\rho \in \gamma_D(d) \mid \tau_I[\![{}^l X := aexp]\!]\{\rho\} \subseteq \gamma_D(r)\} \subseteq \gamma_D((\text{B-ASSIGN}_D[\![X := aexp]\!]d)(r)) \subseteq \gamma_D(d)$*;*
- *a sound unary operator* $\text{FILTER}_D[\![bexp]\!] : \mathcal{D} \to \mathcal{D}$*, together with an effective algorithm to handle a boolean expression bexp, such that* $\{\rho \in \gamma_D(d) \mid \text{true} \in [\![bexp]\!]\rho\} \subseteq \gamma_D(\text{FILTER}_D[\![bexp]\!]d)$*;*
- *a sound widening operator* $\triangledown_D$*, when* $\langle \mathcal{D}, \sqsubseteq_D \rangle$ *does not satisfy the ACC;*
- *a sound narrowing operator* $\triangledown_D$*, when* $\langle \mathcal{D}, \sqsubseteq_D \rangle$ *does not satisfy the DCC;*

The backward assignment operator $\text{B-ASSIGN}_\text{D}[\![X := aexp]\!] : \mathcal{D} \to \mathcal{D} \to \mathcal{D}$, is not directly used to abstract the invariance semantics. However, it usually defined in order to allow the combination of forward and backward analyses [CC92a]. It will also be useful in Chapter 5.

Abstract Invariance Semantics. The operators of the numerical abstract domains can now be used to define the abstract invariance semantics.

In Figure 3.10 we define, for each program instruction *stmt*, its abstract postcondition semantics $\tau_\text{D}[\![stmt]\!] : \mathcal{D} \to \mathcal{D}$. Each function $\tau_\text{D}[\![stmt]\!] : \mathcal{D} \to \mathcal{D}$ takes as input a numerical abstraction and outputs the possible numerical abstraction at the final control point of the instruction. For a `while` loop, $\phi_\text{D}^\triangledown$ is the limit of the following increasing chain (cf. Theorem 2.2.23):

$$\begin{array}{rcl} y_0 & \stackrel{\text{def}}{=} & d \\ y_{n+1} & \stackrel{\text{def}}{=} & y_n \triangledown_\text{D} \phi_\text{D}(y_n) \end{array}$$

The abstract postcondition semantics $\tau_\text{D}[\![prog]\!] \in \mathcal{D}$ of a program *prog* outputs the possible numerical abstraction at the final program control point $f[\![prog]\!] \in \mathcal{L}$. It is defined taking as input the element $\top_\text{D}$ as:

$$
\begin{array}{l}
\tau_{\mathrm{D}}[\![\, {}^{l}\mathtt{skip} \,]\!]d \stackrel{\mathrm{def}}{=} d \\
\tau_{\mathrm{D}}[\![\, {}^{l}X := aexp \,]\!]d \stackrel{\mathrm{def}}{=} \mathrm{ASSIGN}_{\mathrm{D}}[\![\, X := aexp \,]\!]d \\
\tau_{\mathrm{D}}[\![\, \mathtt{if}\ {}^{l}bexp\ \mathtt{then}\ stmt_1\ \mathtt{else}\ stmt_2\ \mathtt{fi} \,]\!]d \stackrel{\mathrm{def}}{=} F_1(d) \sqcup_{\mathrm{D}} F_2(d) \\
\qquad F_1(d) \stackrel{\mathrm{def}}{=} \tau_{\mathrm{D}}[\![\, stmt_1 \,]\!](\mathrm{FILTER}_{\mathrm{D}}[\![\, bexp \,]\!]d) \\
\qquad F_2(d) \stackrel{\mathrm{def}}{=} \tau_{\mathrm{D}}[\![\, stmt_2 \,]\!](\mathrm{FILTER}_{\mathrm{D}}[\![\, \mathtt{not}\ bexp \,]\!]d) \\
\tau_{\mathrm{D}}[\![\, \mathtt{while}\ {}^{l}bexp\ \mathtt{do}\ stmt\ \mathtt{od} \,]\!]d \stackrel{\mathrm{def}}{=} \mathrm{FILTER}_{\mathrm{D}}[\![\, \mathtt{not}\ bexp \,]\!]\phi_{\mathrm{D}}^{\triangledown} \\
\qquad \phi_{\mathrm{D}}(x) \stackrel{\mathrm{def}}{=} d \sqcup_{\mathrm{D}} \tau_{\mathrm{D}}[\![\, stmt \,]\!](\mathrm{FILTER}_{\mathrm{D}}[\![\, bexp \,]\!]x) \\
\tau_{\mathrm{D}}[\![\, stmt_1\ stmt_2 \,]\!]d \stackrel{\mathrm{def}}{=} \tau_{\mathrm{D}}[\![\, stmt_2 \,]\!](\tau_{\mathrm{D}}[\![\, stmt_1 \,]\!]d)
\end{array}
$$

Figure 3.10: Abstract invariance semantics of instructions *stmt*.

Definition 3.4.2 (Abstract Postcondition Semantics) *The* abstract postcondition semantics $\tau_D[\![\, prog \,]\!] \in \mathcal{P}(\mathcal{E})$ *of a program prog is:*

$$\tau_D[\![\, prog \,]\!] = \tau_D[\![\, stmt\ {}^{l} \,]\!] \stackrel{def}{=} \tau_D[\![\, stmt \,]\!]\top_D \tag{3.4.1}$$

where the abstract postcondition semantics $\tau_D[\![\, stmt \,]\!] \in \mathcal{D} \to \mathcal{D}$ *of each program instruction stmt is defined in Figure 3.10.*

The following result proves the soundness of $\tau_{\mathrm{D}}[\![\, prog \,]\!] \in \mathcal{D}$ with respect to the postcondition semantics $\tau_{\mathrm{post}}[\![\, prog \,]\!] \in \mathcal{D}$ (cf. Definition 3.3.1):

Theorem 3.4.3 $\tau_{\mathrm{post}}[\![\, prog \,]\!] \subseteq \gamma_D(\tau_D[\![\, prog \,]\!])$

Proof (Sketch).
The proof follows from the soundness of the operators of the numerical abstract domain (cf. Definition 3.4.1) used for the definition of $\tau_{\mathrm{D}}[\![\, prog \,]\!] \in \mathcal{D}$. ■

In this way, we can collect the possible numerical abstractions at each program control point. At the initial control point, the possible numerical abstraction is the element $\top_{\mathrm{D}}$:

$$\tau_{\mathrm{I}}^{\natural}(i[\![\, prog \,]\!]) \stackrel{\mathrm{def}}{=} \top_{\mathrm{D}}.$$

Then, for each instruction *stmt*, the numerical abstraction $\tau_{\mathrm{I}}^{\natural}(f[\![\, stmt \,]\!]) \in \mathcal{D}$ at its final control point is defined by the abstract postcondition semantics (cf. Figure 3.10), taking as input the numerical abstraction $d \in \mathcal{D}$ at the final

control point of the instruction preceding *stmt* (or the element $\top_{\mathrm{D}}$ in case *stmt* is the first instruction of the program):

$$\tau_{\mathrm{I}}^{\natural}(f[\![\, stmt \,]\!]) \stackrel{\text{def}}{=} \tau_{\mathrm{D}}[\![\, stmt \,]\!]d.$$

For a `while` loop, the numerical abstraction $\tau_{\mathrm{I}}^{\natural}(i[\![\, \texttt{while}\ ^{l} bexp\ \texttt{do}\ stmt\ \texttt{od} \,]\!]) \in \mathcal{D}$ at its initial control point is defined as the limit $\phi_{\mathrm{D}}^{\triangledown}$ of the increasing chain $y_0 \stackrel{\text{def}}{=} d$, $y_{n+1} \stackrel{\text{def}}{=} y_n \triangledown_{\mathrm{D}} \phi_{\mathrm{D}}(y_n)$, where d is the input numerical abstraction and the function $\phi_{\mathrm{D}} \colon \mathcal{D} \to \mathcal{D}$ is defined in Figure 3.10:

$$\tau_{\mathrm{I}}^{\natural}(i[\![\, \texttt{while}\ ^{l} bexp\ \texttt{do}\ stmt\ \texttt{od} \,]\!]) \stackrel{\text{def}}{=} \phi_{\mathrm{D}}^{\triangledown}$$

The numerical abstraction $\tau_{\mathrm{I}}^{\natural}(i[\![\, stmt \,]\!]) \in \mathcal{D}$ at the initial control point of the loop body is the numerical abstraction at the initial control point of the loop filtered by the boolean expression *bexp*:

$$\tau_{\mathrm{I}}^{\natural}(i[\![\, stmt \,]\!]) \stackrel{\text{def}}{=} \mathrm{FILTER}_{\mathrm{D}}[\![\, bexp \,]\!]\tau_{\mathrm{I}}^{\natural}(i[\![\, \texttt{while}\ ^{l} bexp\ \texttt{do}\ stmt\ \texttt{od} \,]\!])$$

The following result proves, for each program control point $l \in \mathcal{L}$, the soundness of the abstract invariance semantics $\tau_{\mathrm{I}}^{\natural} \in \mathcal{D}$ with respect to the invariance semantics $\tau_{\mathrm{I}}(l) \in \mathcal{P}(\mathcal{E})$ proposed in Section 3.3:

Theorem 3.4.4 $\forall l \in \mathcal{L} : \tau_I(l) \subseteq \gamma_D(\tau_I^{\natural}(l))$

Proof (Sketch).
The proof follows from the definition of $\tau_{\mathrm{I}}(l) \in \mathcal{P}(\mathcal{E})$ (cf. Section 3.3) and $\tau_{\mathrm{I}}^{\natural} \in \mathcal{D}$, and from Theorem 3.4.3. ∎

Various numerical abstract domains have been proposed in the literature. We refer to [Min04] for an overview. In the following, we briefly recall the well-known numerical abstract domains of intervals [CC76], convex polyhedra [CH78], and octagons [Min06]. They are the foundation upon which we develop new abstract domains in Chapter 5 and Chapter 6.

3.4.1 Intervals Abstract Domain

The *intervals abstract domain* is a *non-relational* numerical abstract domain. Non-relational abstract domains abstract each program variable independently, thus forgetting any relationship between variables. They are an abstraction

of the power set of environments given by the Galois connection $\langle \mathcal{P}(\mathcal{E}), \subseteq \rangle \underset{\alpha^{\mathrm{C}}}{\overset{\gamma^{\mathrm{C}}}{\leftrightarrows}} \langle \mathcal{X} \to \mathcal{P}(\mathbb{Z}), \dot{\subseteq} \rangle$ where the abstraction $\alpha^{\mathrm{C}} \colon \mathcal{P}(\mathcal{E}) \to (\mathcal{X} \to \mathcal{P}(\mathbb{Z}))$ and the concretization $\gamma^{\mathrm{C}} \colon (\mathcal{X} \to \mathcal{P}(\mathbb{Z})) \to \mathcal{P}(\mathcal{E})$ are defined as follows:

$$\begin{aligned} \alpha^{\mathrm{C}}(E) &\stackrel{\text{def}}{=} \lambda X \in \mathcal{X}.\ \{\rho(X) \in \mathbb{Z} \mid \rho \in E\} \\ \gamma^{\mathrm{C}}(f) &\stackrel{\text{def}}{=} \{\rho \in \mathcal{E} \mid \forall X \in \mathcal{X} : \rho(X) \in f(X)\} \end{aligned} \tag{3.4.2}$$

This abstraction, from now on, is called *cartesian abstraction*.

Specifically, the interval abstract domain maintains an upper bound and a lower bound on the possible values of each program variable. It was introduced by Patrick Cousot and Radhia Cousot [CC76] and it is still widely used as it is efficient and yet able to provide valuable information crucial to prove the absence of various run-time errors.

In the following, the intervals abstract domain is denoted by $\langle \mathcal{B}, \sqsubseteq_{\mathrm{B}} \rangle$. The abstraction is formalized by a Galois connection $\langle \mathcal{X} \to \mathcal{P}(\mathbb{Z}), \dot{\subseteq} \rangle \underset{\alpha_{\mathrm{B}}}{\overset{\gamma_{\mathrm{B}}}{\leftrightarrows}} \langle \mathcal{B}, \sqsubseteq_{\mathrm{B}} \rangle$. Most operators of the domain rely on well-known interval arithmetics [Moo66]. We do not discuss the details here and we refer instead to [Min04].

3.4.2 Polyhedra Abstract Domain

The *polyhedra abstract domain*, introduced by Patrick Cousot and Nicolas Halbwachs [CH78], is a *relational* numerical abstract domain. Relational abstract domains are more precise than non-relational ones since they are able to preserve some of the relationships between the program variables. Specifically, the polyhedra abstract domain allows inferring affine inequalities $c_1 X_1 + \ldots + c_k X_k + c_{k+1} \geq 0$ between the program variables.

In the following, the polyhedra abstract domain is denoted by $\langle \mathcal{P}, \sqsubseteq_{\mathrm{P}} \rangle$. The abstraction is formalized as a concretization-based abstraction $\langle \mathcal{P}(\mathcal{E}), \subseteq \rangle \xleftarrow{\gamma_{\mathrm{P}}} \langle \mathcal{P}, \sqsubseteq_{\mathrm{P}} \rangle$. We again omit the details and we refer to [CH78, Min04].

3.4.3 Octagons Abstract Domain

The *octagons abstract domain* is a *weakly-relational* abstract domain. It was introduced by Antoine Miné [Min06] to answer the need for a trade-off between non-relational abstract domains, that are very cheap but quite imprecise, and relational abstract domains, that are very expressive but quite costly.

The octagons abstract domain, in the following denoted by $\langle \mathcal{O}, \sqsubseteq_{\mathrm{O}} \rangle$, allows inferring inequalities of the form $\pm X_i \pm X_j \geq c$ between the program variables. It is based on the efficient Difference Bound Matrix data structure [LLPY97].

The abstraction, since our program variables have integer values (cf. Section 3.1), is formalized by a Galois connection $\langle \mathcal{P}(\mathcal{E}), \subseteq \rangle \underset{\alpha_{\mathrm{O}}}{\overset{\gamma_{\mathrm{O}}}{\leftrightarrows}} \langle \mathcal{O}, \sqsubseteq_{\mathrm{O}} \rangle$. We refer to [Min04, Min06] for a more detailed discussion.

Note that, the intervals [CC76], convex polyhedra [CH78], and octagons [Min06] numerical abstract domains maintain information about the set of possible values of the program variables along with the possible numerical relationships between them using *convex sets* consisting of conjunctions of linear constraints. The convexity of these abstract domains makes the analysis scalable. On the other hand, it might lead to too harsh approximations and imprecisions in the analysis, ultimately yielding false alarms and a failure of the analyzer to prove the desired program property.

The key for an adequate cost versus precision trade-off is handling disjunctions arising during the analysis (e.g., from program tests and loops). In practice, numerical abstract domains are usually refined by adding weak forms of disjunctions to increase the expressivity while minimizing the cost of the analysis [CCM10, GR98, GC10a, SISG06, etc.].

III

Termination

4

An Abstract Interpretation Framework for Termination

In this chapter, we recall and revise the abstract interpretation framework for potential and definite termination proposed by Patrick Cousot and Radhia Cousot [CC12]. In particular, we recall their fixpoint semantics for definite termination, which is at the basis of our work. For potential termination, we choose a different fixpoint semantics. Then, we provide the definition of the semantics for definite termination for our small language of Chapter 2.

Dans ce chapitre, nous rappelons et révisons le cadre de l'interprétation abstraite de terminaison et de terminaison potentielle proposé par Patrick Cousot et Radhia Cousot [CC12]. En particulier, nous rappelons leur sémantique de point fixe pour la terminaison, qui est à la base de notre travail. Pour la terminaison potentielle, nous choisissons une sémantique de point fixe différente. Ensuite, nous fournissons la définition de la sémantique de terminaison pour notre petit langage du Chapitre 2.

4.1 Ranking Functions

The traditional method for proving program termination dates back to Alan Turing [Tur49] and Robert W. Floyd [Flo67]. It consists in inferring *ranking functions*, namely functions from program states to elements of a well-ordered set whose value decreases during program execution.

Definition 4.1.1 (Ranking Function) *Given a transition system $\langle \Sigma, \tau \rangle$, a* ranking function *is a partial function $f \colon \Sigma \rightharpoonup \mathcal{W}$ from the set of states Σ into a well-ordered set $\langle \mathcal{W}, \leq \rangle$ whose value decreases through transitions between states, that is $\forall s, s' \in \mathrm{dom}(f) : \langle s, s' \rangle \in \tau \Rightarrow f(s') < f(s)$.*

The best known well-ordered sets are the natural numbers $\langle \mathbb{N}, \leq \rangle$ and the ordinals $\langle \mathbb{O}, \leq \rangle$, and the most obvious ranking function maps each program state to the number of program execution steps until termination, or some well-chosen upper bound on this number.

Example 4.1.2

Let us consider again the loop within the program of Example 3.3.2:

```
while ²(1 < x) do
  ³x := x − 1
od⁴
```

The loop terminates whatever the initial value of the variable x. In order to prove it, we define a ranking function $f\colon \Sigma \to \mathbb{N}$ whose domain coincides with the set of program states and whose value is an upper bound on the number of state transitions until termination. We have defined the program transition relation $\tau \in \Sigma \times \Sigma$ in Example 3.2.3. In particular, as in Section 3.3, we partition with respect to the program control points, and we define the ranking function $f\colon \mathcal{L} \to (\mathcal{E} \to \mathbb{N})$:

$$
\begin{aligned}
f(\mathbf{2}) &\stackrel{\text{def}}{=} \lambda\rho.\begin{cases} 1 & \rho(x) \leq 1 \\ 2\rho(x) - 1 & 1 < \rho(x) \end{cases} \\
f(\mathbf{3}) &\stackrel{\text{def}}{=} \lambda\rho.\begin{cases} 2 & \rho(x) \leq 2 \\ 2\rho(x) - 2 & 2 < \rho(x) \end{cases} \\
f(\mathbf{4}) &\stackrel{\text{def}}{=} \lambda\rho.\, 0
\end{aligned}
$$

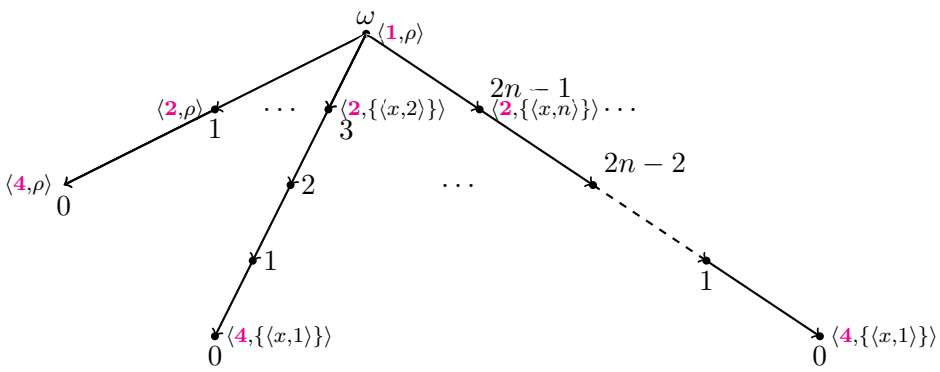
Note that, at the final program control point **4**, the program is terminated and thus no transitions are needed.

At the program control point **2** and **3**, the value of the ranking function depends on the value of the program variable x in the current environment ρ. As an example, given the state $\langle \mathbf{2}, \{\langle x, 2\rangle\}\rangle$, the transitions needed to reach termination are $\langle \mathbf{2}, \{\langle x, 2\rangle\}\rangle \to \langle \mathbf{3}, \{\langle x, 2\rangle\}\rangle$, $\langle \mathbf{3}, \{\langle x, 2\rangle\}\rangle \to \langle \mathbf{2}, \{\langle x, 1\rangle\}\rangle$, and $\langle \mathbf{2}, \{\langle x, 1\rangle\}\rangle \to \langle \mathbf{4}, \{\langle x, 1\rangle\}\rangle$ (cf. Example 3.2.4). Indeed, $f(\mathbf{2})\{\langle x, 2\rangle\} = 3$, $f(\mathbf{3})\{\langle x, 2\rangle\} = 2$, and $f(\mathbf{2})\{\langle x, 1\rangle\} = 1$.

Now, let us consider the whole program of Example 3.3.2:

```
¹x := ?
while ²(1 < x) do
  ³x := x − 1
od⁴
```

At the initial program control point **1**, there is a non-deterministic assignment which gives to the program variable x a random integer value. An upper bound on the number of transitions until termination is thus ω:



Thus, in order to prove that the program terminates, we define a ranking function $f\colon \Sigma \to \mathbb{O}$ mapping the program states to *ordinals*:

$$f(\mathbf{1}) \stackrel{\text{def}}{=} \lambda\rho.\,\omega$$

where $f(\mathbf{2})$, $f(\mathbf{3})$, and $f(\mathbf{4})$ are defined as above. Note that, the value ω does not provide an upper bound on the number of transitions until termination that depends on the value of the program variable x but simply testifies that this number is finite, meaning that the program always terminates. Indeed, even when starting from ω, a strictly decreasing function cannot decrease forever since ordinals are a well-ordered set.

In the following, we also consider a weaker notion of ranking function, called *potential ranking function*. The value of a potential ranking function decreases at least along one transition during program execution.

Definition 4.1.3 (Potential Ranking Function) *Given a transition system $\langle \Sigma, \tau \rangle$, a* potential ranking function *is a partial function $f\colon \Sigma \rightharpoonup \mathcal{W}$ from Σ to a well-ordered set $\langle \mathcal{W}, \leq \rangle$ whose value decreases through at least one transition from each state, that is $\forall s \in \mathrm{dom}(f) : (\exists s' \in \mathrm{dom}(f) : \langle s, s' \rangle \in \tau) \Rightarrow \exists s' \in \mathrm{dom}(f) : \langle s, s' \rangle \in \tau \wedge f(s') < f(s)$.*

4.2 Termination Semantics

In [CC12], Patrick Cousot and Radhia Cousot prove the existence of a *most precise program ranking function* that can be derived by abstract interpreta-

tion of the program maximal trace semantics. In the following, we recall from [CC12] the results that are most relevant to our work.

Potential and Definite Termination. In presence of non-determinism, we distinguish between *potential termination* or *may-terminate* properties and *definite termination* or *must-terminate* properties. The property of potential termination requires a program to have at least one finite execution trace.

Definition 4.2.1 (Potential Termination) *A program with trace semantics* $\mathcal{S} \in \mathcal{P}(\Sigma^{+\infty})$ may terminate *if and only if* $\mathcal{S} \cap \Sigma^{+} \neq \emptyset$.

The property of definite termination requires all execution traces of a program to be finite (cf. also Example 2.2.13).

Definition 4.2.2 (Definite Termination) *A program with trace semantics* $\mathcal{S} \in \mathcal{P}(\Sigma^{+\infty})$ must terminate *if and only if* $\mathcal{S} \subseteq \Sigma^{+}$.

In the following, when clear from the context, we will refer to the property of definite termination simply as property of termination.

4.2.1 Termination Trace Semantics

We present a first abstraction of the program maximal trace semantics (cf. Definition 2.2.8 and Equation 2.2.5, and Definition 3.2.5) into a *potential termination trace semantics* and a *definite termination trace semantics*.

The definite (resp. potential) termination trace semantics eliminates the program execution traces that are not starting with a state from which the program execution must (resp. may) terminate.

Example 4.2.3

Let us consider the following non-deterministic program:

```
while 1( ? ) do
  2skip
od3
```

The program has no variables: $\mathcal{X} \stackrel{\text{def}}{=} \emptyset$. Thus, the set of program environments $\mathcal{E}$ only contains the totally undefined function $\rho\colon \emptyset \to \mathbb{Z}$. The set of program states is $\Sigma \stackrel{\text{def}}{=} \{\mathbf{1}, \mathbf{2}, \mathbf{3}\} \times \mathcal{E}$, and the set of initial states is $\mathcal{I} \stackrel{\text{def}}{=} \{\mathbf{1}\} \times \mathcal{E}$. The program transition relation $\tau \in \Sigma \times \Sigma$ is $\tau \stackrel{\text{def}}{=} \{\langle \mathbf{1}, \rho\rangle \to \langle \mathbf{2}, \rho\rangle \mid \rho \in \mathcal{E}\} \cup \{\langle \mathbf{2}, \rho\rangle \to \langle \mathbf{1}, \rho\rangle \mid \rho \in \mathcal{E}\} \cup \{\langle \mathbf{1}, \rho\rangle \to \langle \mathbf{3}, \rho\rangle \mid \rho \in \mathcal{E}\}$.

The maximal trace semantics $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ of the program (cf. Definition 3.2.5) contains the execution traces starting from the initial states $\mathcal{I}$ that enter the `while` loop a non-deterministic number of times: $\tau^{+\infty} \stackrel{\text{def}}{=} \{\langle \mathbf{1}, \rho\rangle(\langle \mathbf{2}, \rho\rangle)^* \langle \mathbf{3}, \rho\rangle \mid \rho \in \mathcal{E}\} \cup \{\langle \mathbf{1}, \rho\rangle(\langle \mathbf{2}, \rho\rangle)^\omega \mid \rho \in \mathcal{E}\}$. In the following, we abstract such maximal trace semantics into a definite termination trace semantics and a potential termination trace semantics.

Potential Termination Trace Semantics. The *potential termination trace semantics* $\tau_{\mathrm{m}} \in \mathcal{P}(\Sigma^+)$ eliminates all program infinite traces. It can be derived by abstract interpretation of the maximal trace semantics $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ by means of the Galois connection $\langle \mathcal{P}(\Sigma^{+\infty}), \subseteq\rangle \underset{\alpha^{\mathrm{m}}}{\overset{\gamma^{\mathrm{m}}}{\leftrightarrows}} \langle \mathcal{P}(\Sigma^{+}), \subseteq\rangle$, where the abstraction function $\alpha^{\mathrm{m}} \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^+)$ and the concretization function $\gamma^{\mathrm{m}} \colon \mathcal{P}(\Sigma^+) \to \mathcal{P}(\Sigma^{+\infty})$ are defined as follows:

$$\begin{aligned} \alpha^{\mathrm{m}}(T) &\stackrel{\text{def}}{=} T \cap \Sigma^+ \\ \gamma^{\mathrm{m}}(T) &\stackrel{\text{def}}{=} T \cup \Sigma^\omega \end{aligned} \tag{4.2.1}$$

This abstraction, from now on, is called *potential termination abstraction*. It forgets about non-terminating program executions.

Example 4.2.4

Let $T = \{ab, aba, ba, bb, ba^\omega\}$. Then, the potential termination abstraction of T is $\alpha^{\mathrm{m}}(T) = \{ab, aba, ba, bb\}$.

Definition 4.2.5 (Potential Termination Trace Semantics) *The* potential termination trace semantics $\tau_m \in \mathcal{P}(\Sigma^+)$ *is defined as follows:*

$$\tau_m \stackrel{def}{=} \alpha^m(\tau^{+\infty})$$

where $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ *is the maximal trace semantics (cf. Definition 2.2.8).*

The following result provides, by Kleenian fixpoint transfer (cf. Theorem 2.2.20) from the fixpoint maximal trace semantics (cf. Equation 2.2.5), a fixpoint definition of the potential termination trace semantics within the complete lattice $\langle \mathcal{P}(\Sigma^+), \subseteq, \cup, \cap, \emptyset, \Sigma^+\rangle$:

Theorem 4.2.6 (Potential Termination Trace Semantics) *The* potential termination trace semantics $\tau_m \in \mathcal{P}(\Sigma^+)$ *can be expressed as a least fixpoint in the complete lattice* $\langle \mathcal{P}(\Sigma^+), \subseteq, \cup, \cap, \emptyset, \Sigma^+\rangle$ *as follows:*

$$\begin{aligned} \tau_m &= \mathrm{lfp}^{\subseteq}\, \phi_m \\ \phi_m(T) &\stackrel{def}{=} \Omega \cup (\tau \,;\, T) \end{aligned} \tag{4.2.2}$$

In this case, the abstraction function $\alpha^{\mathrm{m}} \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+})$ is a complete $\cup$-morphism and thus there also exists a Galois connection $\langle \mathcal{P}(\Sigma^{+\infty}), \sqsubseteq \rangle \underset{\alpha^{\mathrm{m}}}{\overset{\bar{\gamma}^{\mathrm{m}}}{\leftrightarrows}} \langle \mathcal{P}(\Sigma^{+}), \subseteq \rangle$, where the concretization function $\bar{\gamma}^{\mathrm{m}} \in \mathcal{P}(\Sigma^{+}) \to \mathcal{P}(\Sigma^{+\infty})$ is defined as $\bar{\gamma}^{\mathrm{m}}(T) \stackrel{\mathrm{def}}{=} T$. Note that, because of the distinction between the approximation order $\subseteq$ and the computational order $\sqsubseteq$, such Galois connection is different from the potential termination abstraction presented above (cf. Equation 4.2.1).

Example 4.2.7

Let us consider again the program of Example 4.2.3:

```
while [1]( ? ) do
  [2]skip
od[3]
```

Its potential termination trace semantics is $\tau_{\mathrm{m}} \stackrel{\mathrm{def}}{=} \{\langle \mathbf{1}, \rho\rangle(\langle \mathbf{2}, \rho\rangle)^{*}\langle \mathbf{3}, \rho\rangle \mid \rho \in \mathcal{E}\}$ since an execution trace starting from the state $\langle \mathbf{1}, \rho\rangle$ *may* terminate (by choosing a transition to the state $\langle \mathbf{3}, \rho\rangle$).

Definite Termination Trace Semantics. The *definite termination trace semantics* $\tau_{\mathrm{M}} \in \mathcal{P}(\Sigma^{+})$ eliminates all program execution traces potentially branching, through local non-determinism, to non-termination.

We define the *neighborhood* of a sequence $\sigma \in \Sigma^{+\infty}$ in a set of sequences $T \subseteq \Sigma^{+\infty}$ as the set of sequences $\sigma' \in T$ with a common prefix with σ:

$$\mathrm{nbhd}(\sigma, T) \stackrel{\mathrm{def}}{=} \{\sigma' \in T \mid \mathrm{pf}(\sigma) \cap \mathrm{pf}(\sigma') \neq \emptyset\} \tag{4.2.3}$$

where $\mathrm{pf} \in \Sigma^{+\infty} \to \mathcal{P}(\Sigma^{+\infty})$ yields the set of prefixes of a sequence $\sigma \in \Sigma^{+\infty}$:

$$\mathrm{pf}(\sigma) \stackrel{\mathrm{def}}{=} \{\sigma' \in \Sigma^{+\infty} \mid \exists \sigma'' \in \Sigma^{*\infty} : \ \sigma = \sigma'\sigma''\}. \tag{4.2.4}$$

A program execution trace belongs to the definite termination trace semantics if and only if it is finite and its neighborhood in the program semantics consists only of finite traces. The corresponding *definite termination abstraction* $\alpha^{\mathrm{M}} \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+})$ is defined as follows:

$$\alpha^{\mathrm{M}}(T) \stackrel{\mathrm{def}}{=} \{\sigma \in T^{+} \mid \mathrm{nhbd}(\sigma, T^{\omega}) = \emptyset\}. \tag{4.2.5}$$

where $\mathcal{T}^{+} \stackrel{\mathrm{def}}{=} \mathcal{T} \cap \Sigma^{+}$ and $\mathcal{T}^{\omega} \stackrel{\mathrm{def}}{=} \mathcal{T} \cap \Sigma^{\omega}$.

Example 4.2.8

Let $T = \{ab, aba, ba, bb, ba^\omega\}$. Then, $\alpha^{\mathrm{M}}(T) = \{ab, aba\}$ since $\mathrm{pf}(ab) \cap \mathrm{pf}(ba^\omega) = \emptyset$ and $\mathrm{pf}(aba) \cap \mathrm{pf}(ba^\omega) = \emptyset$, while $\mathrm{pf}(ba) \cap \mathrm{pf}(ba^\omega) = \{ba\} \neq \emptyset$ and $\mathrm{pf}(bb) \cap \mathrm{pf}(ba^\omega) = \{b\} \neq \emptyset$.

Definition 4.2.9 (Definite Termination Trace Semantics) *The* definite termination trace semantics $\tau_M \in \mathcal{P}(\Sigma^+)$ *is defined as follows:*

$$\tau_M \stackrel{def}{=} \alpha^M(\tau^{+\infty})$$

where $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ *is the maximal trace semantics (cf. Definition 2.2.8).*

The following result provides, by Tarskian fixpoint transfer (cf. Theorem 2.2.21) from the fixpoint maximal trace semantics (cf. Equation 2.2.5), a fixpoint definition of the definite termination trace semantics within the complete lattice $\langle \mathcal{P}(\Sigma^+), \subseteq, \cup, \cap, \emptyset, \Sigma^+ \rangle$:

Theorem 4.2.10 (Definite Termination Trace Semantics) *The* definite termination trace semantics $\tau_M \in \mathcal{P}(\Sigma^+)$ *can be expressed as a least fixpoint in the complete lattice* $\langle \mathcal{P}(\Sigma^+), \subseteq, \cup, \cap, \emptyset, \Sigma^+ \rangle$*:*

$$\begin{aligned} \tau_M &= \mathrm{lfp}^{\subseteq}\ \phi_M \\ \phi_M(T) &\stackrel{def}{=} \Omega \cup ((\tau\ ;\ T) \cap \neg(\tau\ ;\ \neg T)) \end{aligned} \tag{4.2.6}$$

where $\neg(\tau\ ;\ \neg T)$ *stands for* $\Sigma^+ \setminus (\tau\ ;\ (\Sigma^+ \setminus T))$*: the term* $\neg(\tau\ ;\ \neg T)$ *eliminates potential transition towards non-terminating executions.*

Example 4.2.11

Let us consider again the program of Example 4.2.3:

```
while 1( ? ) do
  2skip
od3
```

Its definite termination trace semantics is $\tau_{\mathrm{M}} \stackrel{\mathrm{def}}{=} \emptyset$ since for any execution trace starting from the state $\langle \mathbf{1}, \rho \rangle$ there is a possibility of non-termination (by always choosing a transition to the state $\langle \mathbf{2}, \rho \rangle$).

4.2.2 Termination Semantics

We now further abstract the definite (resp. potential) termination trace semantics into a *definite* (resp. *potential*) *termination semantics*, which is the most precise ranking function (resp. potential ranking function) that can be derived by abstract interpretation of the program maximal trace semantics.

Proposition 4.2.12 *A program, whose maximal trace semantics is generated by a transition system $\langle \Sigma, \tau \rangle$, terminates if and only if the program transition relation $\tau \in \Sigma \times \Sigma$ is well-founded.*

When considering a given set $\mathcal{I} \subseteq \Sigma$ of initial states, the program execution traces starting from an initial state are terminating if and only if the program transition relation is well-founded when restricted to reachable states: $\tau \subseteq \alpha^{\mathrm{R}}(\tau^{+\infty}) \times \alpha^{\mathrm{R}}(\tau^{+\infty})$, where $\alpha^{\mathrm{R}} \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma)$ is the reachability abstraction defined in Example 2.2.18.

In the case when the program semantics is not generated by a transition system (cf. Remark 2.2.10), we might consider its *transition abstraction* given by the Galois connection $\langle \mathcal{P}(\Sigma^{+\infty}), \subseteq \rangle \underset{\vec{\alpha}}{\overset{\vec{\gamma}}{\leftrightarrows}} \langle \mathcal{P}(\Sigma \times \Sigma), \subseteq \rangle$ where the abstraction function $\vec{\alpha} \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma \times \Sigma)$ extracts from a set of sequences $T \subseteq \Sigma^{+\infty}$ the smallest transition relation $r \subseteq \Sigma \times \Sigma$ that generates T:

$$\vec{\alpha}\,(T) \stackrel{\text{def}}{=} \{\langle s, s' \rangle \mid \exists \sigma \in \Sigma^*, \sigma' \in \Sigma^{*\infty} : \ \sigma s s' \sigma' \in T\}. \tag{4.2.7}$$

Note, however, that in this case the condition of Proposition 4.2.12 is sufficient but not necessary (i.e., the program execution traces starting from an initial states are terminating *if* the program transition relation is well-founded when restricted to reachable states), as shown by the following counterexample.

Counterexample 4.2.13 *Let $\Sigma = \{a, b\}$. The program whose semantics T is the set of fair traces a^*b is terminating but the transition abstraction $\vec{\alpha}\,(T) = \{\langle a, a \rangle, \langle a, b \rangle\}$ generates the infinite trace a^ω (cf. also Remark 2.2.10). Thus, the transition relation restricted to the reachable states is not well-founded.*

An over-approximation $\mathcal{R} \supseteq \alpha^{\mathrm{R}}(\tau^{+\infty})$ of the reachable states can be computed by abstract interpretation, as we have seen in Section 3.4. The program transition relation restricted to the reachable states $\tau \subseteq \alpha^{\mathrm{R}}(\tau^{+\infty}) \times \alpha^{\mathrm{R}}(\tau^{+\infty})$ is well-founded if its over-approximation $r \subseteq \mathcal{R} \times \mathcal{R}$ is well-founded. Moreover, $r \subseteq \mathcal{R} \times \mathcal{R}$ is well-founded if and only if there exists a ranking function

$f\colon \Sigma \rightharpoonup \mathcal{W}$ into a well-ordered set $\langle \mathcal{W}, \leq \rangle$ whose domain is $\mathcal{R}$: $\mathrm{dom}(f) = \mathcal{R}$. Thus, for programs whose maximal trace semantics is generated by a transition system, (potential) termination can be proved by exhibiting a (potential) ranking function mapping invariant states to elements of a well-ordered set.

In the following, we consider the set of partial functions from the program states Σ into the well-ordered set of ordinals $\langle \mathbb{O}, \leq \rangle$.

Potential Termination Semantics. The *potential termination semantics* is the most precise potential ranking function $\tau_{\mathrm{mt}} \in \Sigma \rightharpoonup \mathbb{O}$ for a program. It is defined starting from the final states in Ω, where the function has value zero, and retracing the program *backwards* while mapping each program state in Σ potentially leading to a final state (i.e., a program state such that there exists a terminating program execution trace to which it belongs) to an ordinal in $\mathbb{O}$ representing the minimum number of program execution steps remaining to termination. The domain $\mathrm{dom}(\tau_{\mathrm{mt}})$ of τ_{mt} is the set of states from which the program execution may terminate: at least one trace branching from a state $s \in \mathrm{dom}(\tau_{\mathrm{mt}})$ terminates in at least $\tau_{\mathrm{mt}}(s)$ execution steps, while all traces branching from a state $s \notin \mathrm{dom}(\tau_{\mathrm{mt}})$ do not terminate.

Intuitively, a potential ranking function f_1 is more precise than another potential ranking function f_2 when it is defined over a smaller set of program states, that is, it can disprove termination for more program states, and when its value is always smaller, that is, the minimum number of program execution steps required for termination is lower. The *approximation order* is then:

$$f_1 \sqsubseteq f_2 \iff \mathrm{dom}(f_1) \subseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_1) : f_1(x) \leq f_2(x). \tag{4.2.8}$$

Definition 4.2.14 (Potential Termination Semantics) *The* potential termination semantics $\tau_{mt} \in \Sigma \rightharpoonup \mathbb{O}$ *is derived by abstract interpretation of the maximal trace semantics* $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$*:*

$$\tau_{mt} \stackrel{def}{=} \alpha^{mrk}(\alpha^{m}(\tau^{+\infty})) = \alpha^{mrk}(\tau_{m}) \tag{4.2.9}$$

where the potential ranking abstraction $\alpha^{mrk}\colon \mathcal{P}(\Sigma^{+}) \to (\Sigma \rightharpoonup \mathbb{O})$ *is:*

$$\alpha^{mrk}(T) \stackrel{def}{=} \alpha^{mv}(\vec{\alpha}\,(T)) \tag{4.2.10}$$

where the function $\alpha^{mv}\colon \mathcal{P}(\Sigma \times \Sigma) \to (\Sigma \rightharpoonup \mathbb{O})$ *provides the rank of the elements in the domain of a relation* $r \subseteq \Sigma \times \Sigma$*:*

$$\alpha^{mv}(\emptyset) \stackrel{def}{=} \dot{\emptyset}$$

$$\alpha^{mv}(r)s \stackrel{def}{=} \begin{cases} 0 & \forall s' \in \Sigma : \ \langle s, s' \rangle \notin r \\ \inf \left\{ \alpha^{mv}(r)s' + 1 \;\middle|\; \begin{array}{c} s' \in \mathrm{dom}(\alpha^{mv}(r)) \\ \wedge \ \langle s, s' \rangle \in r \end{array} \right\} & \text{otherwise} \end{cases}$$

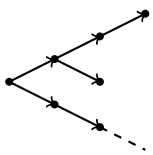
The following result provides, by Kleenian fixpoint transfer (cf. Theorem 2.2.20) from the fixpoint potential termination trace semantics (cf. Equation 4.2.2), a fixpoint definition of the potential termination semantics within the partially ordered set $\langle \Sigma \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$, where the computational order coincides with the approximation order defined in Equation 4.2.8:

Theorem 4.2.15 (Potential Termination Semantics) *The* potential termination semantics $\tau_{mt} \in \Sigma \rightharpoonup \mathbb{O}$ *of a program can be expressed as a least fixpoint in the partially ordered set* $\langle \Sigma \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$*:*

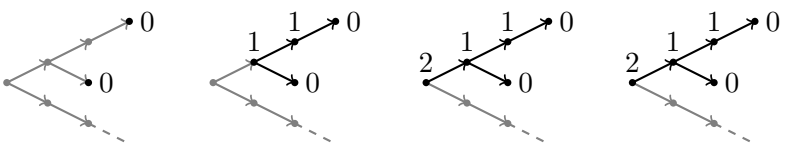
$$\begin{aligned} \tau_{mt} &= \mathrm{lfp}^{\sqsubseteq}_{\dot{\emptyset}}\ \phi_{mt} \\ \phi_{mt}(f) &\stackrel{def}{=} \lambda s. \begin{cases} 0 & s \in \Omega \\ \inf\{f(s')+1 \mid \langle s, s' \rangle \in \tau\} & s \in \mathrm{pre}(\mathrm{dom}(f)) \\ \text{undefined} & \text{otherwise} \end{cases} \end{aligned} \tag{4.2.11}$$

Example 4.2.16

Let us consider the following trace semantics:



The fixpoint iterates of the potential termination semantics $\tau_{mt} \in \Sigma \rightharpoonup \mathbb{O}$ are:



where unlabelled states are outside the domain of the function.

Note that our definition of the potential termination semantics $\tau_{mt} \in \Sigma \rightharpoonup \mathbb{O}$ differs from [CC12]: the value of the ranking function for a state is the *greatest lower bound* (instead of the *least upper bound*) of the value plus one of the ranking function for its successors. In fact, the following example shows that the definition of [CC12] does not guarantee the existence of a least fixed point.

Example 4.2.17

Let us consider again the program of Example 4.2.7:

```
while 1( ? ) do
  2skip
od3
```

The iterates of the potential termination semantics $\tau_{mt} \in \Sigma \rightharpoonup \mathbb{O}$ with respect to the program control points are the following:

$$\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl}
f_0(\mathbf{1}) & \stackrel{\text{def}}{=} & \dot{\emptyset} & f_0(\mathbf{2}) & \stackrel{\text{def}}{=} & \dot{\emptyset} & f_0(\mathbf{3}) & \stackrel{\text{def}}{=} & \dot{\emptyset} \\
f_1(\mathbf{1}) & \stackrel{\text{def}}{=} & \dot{\emptyset} & f_1(\mathbf{2}) & \stackrel{\text{def}}{=} & \dot{\emptyset} & f_1(\mathbf{3}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 0 \\
f_2(\mathbf{1}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 1 & f_2(\mathbf{2}) & \stackrel{\text{def}}{=} & \dot{\emptyset} & f_2(\mathbf{3}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 0 \\
f_3(\mathbf{1}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 1 & f_3(\mathbf{2}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 2 & f_3(\mathbf{3}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 0 \\
f_4(\mathbf{1}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 1 & f_4(\mathbf{2}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 2 & f_4(\mathbf{3}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 0
\end{array}$$

Instead, the iterates of the potential termination semantics of [CC12] are:

$$\begin{array}{lcl@{\qquad}lcl@{\qquad}lcl}
f_0(\mathbf{1}) & \stackrel{\text{def}}{=} & \dot{\emptyset} & f_0(\mathbf{2}) & \stackrel{\text{def}}{=} & \dot{\emptyset} & f_0(\mathbf{3}) & \stackrel{\text{def}}{=} & \dot{\emptyset} \\
f_1(\mathbf{1}) & \stackrel{\text{def}}{=} & \dot{\emptyset} & f_1(\mathbf{2}) & \stackrel{\text{def}}{=} & \dot{\emptyset} & f_1(\mathbf{3}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 0 \\
f_2(\mathbf{1}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 1 & f_2(\mathbf{2}) & \stackrel{\text{def}}{=} & \dot{\emptyset} & f_2(\mathbf{3}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 0 \\
f_3(\mathbf{1}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 1 & f_3(\mathbf{2}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 2 & f_3(\mathbf{3}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 0 \\
f_4(\mathbf{1}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 3 & f_4(\mathbf{2}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 2 & f_4(\mathbf{3}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 0 \\
f_5(\mathbf{1}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 3 & f_5(\mathbf{2}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 4 & f_5(\mathbf{3}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 0 \\
f_6(\mathbf{1}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 5 & f_6(\mathbf{2}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 4 & f_6(\mathbf{3}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 0 \\
 & & & & \vdots & & & & \\
f_\omega(\mathbf{1}) & \stackrel{\text{def}}{=} & \lambda\rho.\ \omega & f_\omega(\mathbf{2}) & \stackrel{\text{def}}{=} & \lambda\rho.\ \omega & f_\omega(\mathbf{3}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 0 \\
f_{\omega+1}(\mathbf{1}) & \stackrel{\text{def}}{=} & \lambda\rho.\ \omega+1 & f_{\omega+1}(\mathbf{2}) & \stackrel{\text{def}}{=} & \lambda\rho.\ \omega+1 & f_{\omega+1}(\mathbf{3}) & \stackrel{\text{def}}{=} & \lambda\rho.\ 0 \\
 & & & & \vdots & & & &
\end{array}$$

In particular, note that the value of the potential termination semantics at the program control points **1** and **2** is always increasing.

The potential termination semantics is sound and complete for proving potential termination of a program for a given set of initial states $\mathcal{I} \subseteq \Sigma$:

Theorem 4.2.18 *A program may terminate for execution traces starting from a given set of initial states $\mathcal{I}$ if and only if $\mathcal{I} \subseteq \mathrm{dom}(\tau_{mt})$.*

Proof.

See Appendix A.2. ■

Definite Termination Semantics. The *definite termination semantics* is the most precise ranking function $\tau_{\mathrm{Mt}} \in \Sigma \rightharpoonup \mathbb{O}$ for a program. It is defined starting from the final states in Ω, where the function has value zero, and retracing the program *backwards* while mapping each program state in Σ definitely leading to a final state (i.e., a program state such that all program execution traces to which it belongs are terminating) to an ordinal in $\mathbb{O}$ representing an upper bound on the number of program execution steps remaining to termination. The domain $\mathrm{dom}(\tau_{\mathrm{Mt}})$ of τ_{Mt} is the set of states from which the program execution must terminate: all traces branching from a state $s \in \mathrm{dom}(\tau_{\mathrm{Mt}})$ terminate in at most $\tau_{\mathrm{Mt}}(s)$ execution steps, while at least one trace branching from a state $s \notin \mathrm{dom}(\tau_{\mathrm{Mt}})$ does not terminate.

Intuitively, a ranking function f_1 is more precise than another ranking function f_2 when it is defined over a larger set of program states, that is, it can prove termination for more program states, and when its value is always smaller, that is, the maximum number of program execution steps required for termination is smaller. The *approximation order* is then:

$$f_1 \preccurlyeq f_2 \iff \mathrm{dom}(f_1) \supseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_2) : f_1(x) \leq f_2(x). \quad (4.2.12)$$

Definition 4.2.19 (Definite Termination Semantics) *The* definite termination semantics $\tau_{Mt} \in \Sigma \rightharpoonup \mathbb{O}$ *is derived by abstract interpretation of the maximal trace semantics* $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$*:*

$$\tau_{Mt} \stackrel{def}{=} \alpha^{Mrk}(\alpha^{M}(\tau^{+\infty})) = \alpha^{Mrk}(\tau_{M}) \quad (4.2.13)$$

where the ranking abstraction $\alpha^{Mrk}\colon \mathcal{P}(\Sigma^{+}) \to (\Sigma \rightharpoonup \mathbb{O})$ *is:*

$$\alpha^{Mrk}(T) \stackrel{def}{=} \alpha^{Mv}(\vec{\alpha}\,(T)) \quad (4.2.14)$$

where the function $\alpha^{Mv}\colon \mathcal{P}(\Sigma \times \Sigma) \to (\Sigma \rightharpoonup \mathbb{O})$ *provides the rank of the elements in the domain of a relation* $r \subseteq \Sigma \times \Sigma$*:*

$$\alpha^{Mv}(r)s \stackrel{def}{=} \begin{cases} 0 & \forall s' \in \Sigma : \langle s, s' \rangle \notin r \\ \sup \left\{ \alpha^{Mv}(r)s' + 1 \;\middle|\; \begin{array}{c} s' \in \mathrm{dom}(\alpha^{Mv}(r)) \\ \wedge\ \langle s, s' \rangle \in r \end{array} \right\} & \text{otherwise} \end{cases}$$

The following result provides a fixpoint definition of the definite termination semantics within the partially ordered set $\langle \Sigma \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$, where the computational order is defined as:

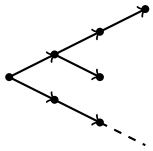
$$f_1 \sqsubseteq f_2 \iff \mathrm{dom}(f_1) \subseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_1) : f_1(x) \leq f_2(x). \quad (4.2.15)$$

Theorem 4.2.20 (Definite Termination Semantics) *The definite termination semantics $\tau_{Mt} \in \Sigma \rightharpoonup \mathbb{O}$ can be expressed as a least fixpoint in the partially ordered set $\langle \Sigma \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$:*

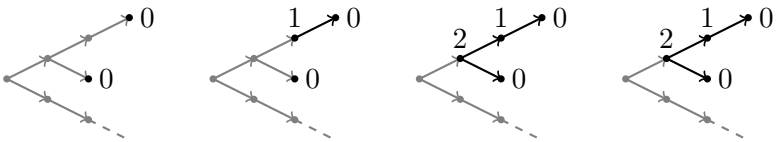
$$\begin{aligned} \tau_{Mt} &= \mathrm{lfp}_{\dot{\emptyset}}^{\sqsubseteq}\ \phi_{Mt} \\ \phi_{Mt}(f) &\stackrel{def}{=} \lambda s. \begin{cases} 0 & s \in \Omega \\ \sup\{f(s')+1 \mid \langle s, s'\rangle \in \tau\} & s \in \widetilde{\mathrm{pre}}(\mathrm{dom}(f)) \\ \text{undefined} & \text{otherwise} \end{cases} \end{aligned} \tag{4.2.16}$$

Example 4.2.21

Let us consider again the trace semantics of Example 4.2.16:



The fixpoint iterates of the definite termination semantics $\tau_{\mathrm{Mt}} \in \Sigma \rightharpoonup \mathbb{O}$ are:



where unlabelled states are outside the domain of the function.

Note that, the approximation order $\preccurlyeq$ and the computational order $\sqsubseteq$ coincide when the ranking functions have the same domain:

Lemma 4.2.22 $\mathrm{dom}(f_1) = \mathrm{dom}(f_2) \Rightarrow (f_1 \preccurlyeq f_2 \iff f_1 \sqsubseteq f_2)$

Proof.

The proof is immediate from Equation 4.2.12 and Equation 4.2.15. ■

The definite termination semantics is sound and complete for proving definite termination of a program for a given set of initial states $\mathcal{I} \subseteq \Sigma$:

Theorem 4.2.23 *A program must terminate for execution traces starting from a given set of initial states $\mathcal{I}$ if and only if $\mathcal{I} \subseteq \mathrm{dom}(\tau_{Mt})$.*

Proof.

See [CC12]. ■

4.3 Denotational Definite Termination Semantics

In this work, we are mostly interested in proving program *definite* termination.

In the following, we provide a structural definition of the fixpoint definite termination semantics $\tau_{\mathrm{Mt}} \in \Sigma \rightharpoonup \mathbb{O}$ (cf. Equation 4.2.16) by induction on the syntax of programs written in the small language presented in Chapter 3.

In Section 3.3 we partitioned the reachable state semantics τ_{R} (cf. Equation 2.2.10) into an invariance semantics $\tau_{\mathrm{I}} \in \mathcal{L} \to \mathcal{P}(\mathcal{E})$. Similarly, we partition τ_{Mt} with respect to the program control points: $\tau_{\mathrm{Mt}} \in \mathcal{L} \to (\mathcal{E} \rightharpoonup \mathbb{O})$. In this way, to each program control point $l \in \mathcal{L}$ corresponds a partial function $f \colon \mathcal{E} \rightharpoonup \mathbb{O}$, and to each program instruction *stmt* corresponds a termination semantics $\tau_{\mathrm{Mt}}[\![\, \mathit{stmt} \,]\!] \colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$.

The termination semantics $\tau_{\mathrm{Mt}}[\![\, \mathit{stmt} \,]\!] \colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ of each program instruction *stmt* outputs a ranking function whose domain represents the terminating environments at the initial label of *stmt*, which is determined taking as input a ranking function whose domain represents the terminating environments at the final label of *stmt*, and whose value represents an upper bound on the number of program execution steps remaining to termination.

The termination semantics of a `skip` instruction takes as input a ranking function $f \colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain represents the terminating environments at the final label of the instruction, and increases its value by one to take into account that from the environments at the initial label of the instruction another program execution step is necessary before termination:

$$\tau_{\mathrm{Mt}}[\![\, {}^{l}\mathtt{skip} \,]\!] f \stackrel{\text{def}}{=} \lambda \rho \in \mathrm{dom}(f).\ f(\rho) + 1 \tag{4.3.1}$$

Similarly, the termination semantics of a variable assignment ${}^{l}X := \mathit{aexp}$ takes as input a ranking function $f \colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain represent the terminating environments at the final label of the instruction. The resulting ranking function is defined over the environments that when subject to the variable assignment always belong to the domain of the input ranking function. The value of the input ranking function for these environments is increased by one, to take into account another execution step before termination, and the value of the resulting ranking function is the least upper bound of these values:

$$\tau_{\mathrm{Mt}}[\![\, {}^{l}X := \mathit{aexp} \,]\!] f \stackrel{\text{def}}{=} \lambda \rho. \begin{cases} \sup\{f(\rho[X \leftarrow v]) + 1 \mid v \in [\![\mathit{aexp}]\!]\rho\} & \\ \qquad\qquad\qquad\qquad\qquad\qquad \exists v \in [\![\mathit{aexp}]\!]\rho \wedge & \\ \qquad\qquad\qquad \forall v \in [\![\mathit{aexp}]\!]\rho :\ \rho[X \leftarrow v] \in \mathrm{dom}(f) & \\ \text{undefined} & \text{otherwise} \end{cases} \tag{4.3.2}$$

Note that, all environments yielding a run-time error due to a division by zero do not belong to the domain of the termination semantics of the assignment.

Example 4.3.1

Let $\mathcal{X} \stackrel{\text{def}}{=} \{x\}$. We consider the following ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$:

$$f(\rho) \stackrel{\text{def}}{=} \begin{cases} 2 & \rho(x) = 1 \\ 3 & \rho(x) = 2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

and the backward assignment $x := x + [1, 2]$. The termination semantics of the assignment, given the ranking function, is:

$$\tau_{\text{Mt}}[\![\, x := x + [1, 2] \,]\!] f(\rho) \stackrel{\text{def}}{=} \begin{cases} 4 & \rho(x) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

In particular, note that the function is only defined when $\rho(x) = 0$. In fact, when $\rho(x) = -1$, we have $[\![x + [1, 2]]\!]\rho = \{0, 1\}$ and $\rho[x \leftarrow 0] \notin \text{dom}(f)$. Similarly, when $\rho(x) = 1$, we have $[\![x+[1, 2]]\!]\rho = \{2, 3\}$ and $\rho[x \leftarrow 3] \notin \text{dom}(f)$.

Given a conditional instruction `if` ^{l}*bexp* `then` *stmt*$_1$ `else` *stmt*$_2$ `fi`, its termination semantics takes as input a ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$ and derives the termination semantics $\tau_{\text{Mt}}[\![\, \mathit{stmt}_1 \,]\!] f$ of *stmt*$_1$, in the following denoted by S_1, and the termination semantics $\tau_{\text{Mt}}[\![\, \mathit{stmt}_2 \,]\!] f$ of *stmt*$_2$, in the following denoted by S_2. Then, the termination semantics of the conditional instruction is defined by means of the ranking function $F[f]\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments that belong to the domain of S_1 and to the domain of S_2, and that due to non-determinism may both satisfy and do not satisfy the boolean expression *bexp*:

$$F[f] \stackrel{\text{def}}{=} \lambda\rho \in \text{dom}(S_1) \cap \text{dom}(S_2).\ \begin{cases} \sup\{S_1(\rho) + 1, S_2(\rho) + 1\} & \\ & [\![\mathit{bexp}]\!]\rho = \{\text{true}, \text{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

and the ranking function $F_1[f]\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments $\rho \in \mathcal{E}$ that belong to the domain of S_1 and that must satisfy *bexp*:

$$F_1[f] \stackrel{\text{def}}{=} \lambda\rho \in \text{dom}(S_1).\ \begin{cases} S_1(\rho) + 1 & [\![\mathit{bexp}]\!]\rho = \{\text{true}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

and the ranking function $F_2[f]\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments that belong to the domain of S_2 and that cannot satisfy *bexp*:

$$F_2[f] \stackrel{\text{def}}{=} \lambda\rho \in \mathrm{dom}(S_2).\ \begin{cases} S_2(\rho) + 1 & [\![bexp]\!]\rho = \{\text{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The resulting ranking function is defined joining $F[f]$, $F_1[f]$, and $F[f]$:

$$\tau_{\mathrm{Mt}}[\![\, \texttt{if}\ {}^{l}bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi}\,]\!]f \stackrel{\text{def}}{=} F[f] \dot{\cup} F_1[f] \dot{\cup} F_2[f] \tag{4.3.3}$$

Example 4.3.2

Let $\mathcal{X} \stackrel{\text{def}}{=} \{x\}$. We consider the termination semantics of the conditional statement `if` *bexp* `then` $stmt_1$ `else` $stmt_2$ `fi`. We assume, given a ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$, that the termination semantics of $stmt_1$ is defined as:

$$\tau_{\mathrm{Mt}}[\![\, stmt_1 \,]\!]f(\rho) \stackrel{\text{def}}{=} \begin{cases} 1 & \rho(x) \leq 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

and that the termination semantics of $stmt_2$ is defined as

$$\tau_{\mathrm{Mt}}[\![\, stmt_2 \,]\!]f(\rho) \stackrel{\text{def}}{=} \begin{cases} 3 & 0 \leq \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Then, when the boolean expression *bexp* is for example $x \leq 3$, the termination semantics of the conditional statement is:

$$\tau_{\mathrm{Mt}}[\![\, \texttt{if}\ {}^{l}bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi}\,]\!]f(\rho) \stackrel{\text{def}}{=} \begin{cases} 2 & \rho(x) \leq 0 \\ 4 & 3 < \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Instead, when *bexp* is for example the non-deterministic choice `?`, we have:

$$\tau_{\mathrm{Mt}}[\![\, \texttt{if}\ {}^{l}bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi}\,]\!]f(\rho) \stackrel{\text{def}}{=} \begin{cases} 4 & \rho(x) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

The termination semantics of a loop instruction `while` ${}^{l}bexp$ `do` *stmt* `od` takes as input a ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$ the domain of which represents the terminating environments at the final label of the instruction, and outputs the

ranking function which is defined as a least fixpoint of the function $\phi_{\mathrm{Mt}} \colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ within $\langle \mathcal{E} \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$, analogously to Equation 4.2.16:

$$\tau_{\mathrm{Mt}}[\![\, \texttt{while}\ ^{l} \mathit{bexp}\ \texttt{do}\ \mathit{stmt}\ \texttt{od} \,]\!] f \stackrel{\text{def}}{=} \mathrm{lfp}_{\dot{\emptyset}}^{\sqsubseteq}\ \phi_{\mathrm{Mt}} \tag{4.3.4}$$

where the *computational order* is defined as in Equation 4.2.15:

$$f_1 \sqsubseteq f_2 \iff \mathrm{dom}(f_1) \subseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_1) : f_1(x) \leq f_2(x).$$

The function $\phi_{\mathrm{Mt}} \colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ takes as input a ranking function $x \colon \mathcal{E} \rightharpoonup \mathbb{O}$ and adds to its domain the environments for which one more loop iteration is needed before termination. In the following, the termination semantics $\tau_{\mathrm{Mt}}[\![\, \mathit{stmt} \,]\!] x$ of the loop body is denoted by S. The function ϕ_{Mt} is defined by means of the ranking function $F[x] \colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments that belong to the domain of S and to the domain of the input function f, and that may both satisfy and not satisfy the boolean expression *bexp*:

$$F[x] \stackrel{\text{def}}{=} \lambda \rho \in \mathrm{dom}(S) \cap \mathrm{dom}(f). \begin{cases} \sup\{S(\rho) + 1, f(\rho) + 1\} & \\ & [\![\mathit{bexp}]\!]\rho = \{\mathrm{true}, \mathrm{false}\} \\ \mathrm{undefined} & \mathrm{otherwise} \end{cases}$$

and the ranking function $F_1[x] \colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments $\rho \in \mathcal{E}$ that belong to the domain of S and that must satisfy *bexp*:

$$F_1[x] \stackrel{\text{def}}{=} \lambda \rho \in \mathrm{dom}(S). \begin{cases} S(\rho) + 1 & [\![\mathit{bexp}]\!]\rho = \{\mathrm{true}\} \\ \mathrm{undefined} & \mathrm{otherwise} \end{cases}$$

and the ranking function $F_2[f] \colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments that belong to the domain of the input function f and that cannot satisfy *bexp*:

$$F_2[f] \stackrel{\text{def}}{=} \lambda \rho \in \mathrm{dom}(f). \begin{cases} f(\rho) + 1 & [\![\mathit{bexp}]\!]\rho = \{\mathrm{false}\} \\ \mathrm{undefined} & \mathrm{otherwise} \end{cases}$$

The resulting ranking function is defined joining $F[x]$, $F_1[x]$, and $F_2[f]$:

$$\phi_{\mathrm{Mt}}(x) \stackrel{\text{def}}{=} F[x] \mathbin{\dot{\cup}} F_1[x] \mathbin{\dot{\cup}} F_2[f] \tag{4.3.5}$$

Finally, the termination semantics of the sequential combination of instructions $\mathit{stmt}_1\ \mathit{stmt}_2$, takes as input a ranking function $f \colon \mathcal{E} \rightharpoonup \mathbb{O}$, determines from

f the termination semantics $\tau_{\mathrm{Mt}}[\![\, stmt_2 \,]\!] f$ of $stmt_2$, and outputs the ranking function determined by the termination semantics of $stmt_1$ from $\tau_{\mathrm{Mt}}[\![\, stmt_2 \,]\!] f$:

$$\tau_{\mathrm{Mt}}[\![\, stmt_1 \;\; stmt_2 \,]\!] f \stackrel{\mathrm{def}}{=} \tau_{\mathrm{Mt}}[\![\, stmt_1 \,]\!](\tau_{\mathrm{Mt}}[\![\, stmt_2 \,]\!] f) \tag{4.3.6}$$

The termination semantics $\tau_{\mathrm{Mt}}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$ of a program *prog* is a ranking function whose domain represents the terminating environments, which is determined taking as input the zero function:

Definition 4.3.3 (Termination Semantics) *The* termination semantics $\tau_{Mt}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$ *of a program prog is:*

$$\tau_{Mt}[\![\, prog \,]\!] = \tau_{Mt}[\![\, stmt^{\,l} \,]\!] \stackrel{def}{=} \tau_{Mt}[\![\, stmt \,]\!](\lambda\rho.\ 0) \tag{4.3.7}$$

where the function $\tau_{Mt}[\![\, stmt \,]\!] \colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ *is the termination semantics of each program instruction stmt.*

Note that, as pointed out in Remark 3.2.2 and accordingly to Definition 3.2.5, possible run-time errors silently halting the program are ignored. More specifically, all environments leading to run-time errors are discarded and do not belong to the domain of the termination semantics of a program *prog*.

The termination semantics $\tau_{\mathrm{Mt}}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$ is usually not computable. In the next Chapter 5 and Chapter 6, we present sound decidable abstractions of $\tau_{\mathrm{Mt}}[\![\, prog \,]\!]$ by means of piecewise-defined functions. The soundness is related to the same *approximation order* defined in Equation 4.2.12 as:

$$f_1 \preccurlyeq f_2 \iff \mathrm{dom}(f_1) \supseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_2) : f_1(x) \leq f_2(x).$$

5

Piecewise-Defined Ranking Functions

In this chapter, we present a parameterized numerical abstract domain for effectively proving program termination by abstract interpretation of the definite termination semantics presented in Section 4.3. The domain is used to automatically synthesize piecewise-defined ranking functions and infer sufficient preconditions for program termination. The elements of the abstract domain are piecewise-defined partial functions represented by decision trees, where the decision nodes are labeled with linear constraints, and the leaf nodes belong to an auxiliary abstract domain for functions.

The abstract domain is parametric in the choice between the expressivity and the cost of the numerical abstract domain which underlies the linear constraints labeling the decision nodes, and the choice of the auxiliary abstract domain for the leaf nodes. We describe various instances based on the numerical abstract domains presented in Section 3.4 for the decision nodes, and affine functions for the leaf nodes.

Dans ce chapitre, nous présentons un domaine abstrait numérique paramétré pour prouver effectivement la terminaison de programmes par interprétation abstraite de la sémantique de terminaison présentée dans la Section 4.3. Le domaine est utilisé pour synthétiser automatiquement des fonctions de rang définies par morceaux et en déduire des conditions suffisantes pour la terminaison de programmes. Les éléments du domaine abstrait sont des fonctions partielles définies par morceaux représentées par des arbres de décision, où les nœuds de décision sont étiquetés avec des contraintes linéaires et les feuilles appartiennent à un domaine abstrait auxiliaire pour fonctions.

Le domaine abstrait est paramétrique dans le choix entre l'expressivité et le coût du domaine abstrait numérique qui sous-tend les contraintes linéaires

étiquetant les nœuds de décision, et le choix du domaine abstrait auxiliaire pour les feuilles. Nous décrivons différentes instances sur la base des domaines numériques abstraits présentés dans la Section 3.4 pour les nœuds de décision, et les fonctions affines pour les nœuds feuilles.

5.1 Piecewise-Defined Ranking Functions

In order to abstract the termination semantics presented in Section 4.3, we consider the following concretization-based abstraction:

$$\langle \mathcal{E} \rightharpoonup \mathbb{O}, \preccurlyeq \rangle \xleftarrow{\gamma_{\mathrm{T}}} \langle \mathcal{T}, \preccurlyeq_{\mathrm{T}} \rangle$$

which provides a sound decidable abstraction $\tau^{\natural}_{\mathrm{Mt}}[\![\, prog \,]\!] \in \mathcal{T}$ of the termination semantics of programs $\tau_{\mathrm{Mt}}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$ with respect to the approximation order defined in Equation 4.2.12:

$$f_1 \preccurlyeq f_2 \stackrel{\text{def}}{=} \mathrm{dom}(f_1) \supseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_2) : f_1(x) \leq f_2(x).$$

We have $\tau_{\mathrm{Mt}}[\![\, prog \,]\!] \preccurlyeq \gamma_{\mathrm{T}}(\tau^{\natural}_{\mathrm{Mt}}[\![\, prog \,]\!])$, meaning that the abstract termination semantics $\tau^{\natural}_{\mathrm{Mt}}[\![\, prog \,]\!]$ over-approximates the value of the ranking function $\tau_{\mathrm{Mt}}[\![\, prog \,]\!]$ and under-approximates its domain of definition $\mathrm{dom}(\tau_{\mathrm{Mt}}[\![\, prog \,]\!])$. In this way, an abstraction provides *sufficient preconditions* for program termination: if the abstract ranking function is defined on a program state, then all program execution traces branching from that state are terminating.

By pointwise lifting (cf. Equation 2.1.1) we obtain the following:

$$\langle \mathcal{L} \to (\mathcal{E} \rightharpoonup \mathbb{O}), \dot{\preccurlyeq} \rangle \xleftarrow{\dot{\gamma}_{\mathrm{T}}} \langle \mathcal{L} \to \mathcal{T}, \dot{\preccurlyeq}_{\mathrm{T}} \rangle$$

which provides a concretization-based abstraction of the definite termination semantics $\tau_{\mathrm{Mt}} \in \mathcal{L} \to (\mathcal{E} \rightharpoonup \mathbb{O})$ by partitioning with respect to the program control points. No approximation is made on $\mathcal{L}$. On the other hand, each program control point $l \in \mathcal{L}$ is associated with an element $t \in \mathcal{T}$.

Piecewise-Defined Ranking Functions. The elements of the abstract domain $\langle \mathcal{T}, \preccurlyeq_{\mathrm{T}} \rangle$ are *piecewise-defined* partial functions.

Their internal representation is inspired by the space partitioning trees [FKN80] developed in the context of 3D computer graphics and the use of decision trees in program analysis and verification [BCC+10, Jea02]: the piecewise-defined partial functions are represented by *decision trees*, where the decision

nodes are labeled with linear constraints, and the leaf nodes belong to an auxiliary abstract domain for functions. The decision trees recursively partition the space of possible values of the program variables inducing disjunctions into the auxiliary domain. The elements of the auxiliary domain are functions of the program variables, which provide upper bounds on the computational complexity of the program in terms of execution steps.

The partitioning is *dynamic*: during the analysis, partitions (resp. decision nodes and constraints) are split (resp. added) by tests, modified by assignments and joined (resp. removed) when merging control flows. In order to minimize the cost of the analysis, a widening limits the height of the decision trees and the number of maintained disjunctions.

The abstract domain is parametric in the choice between the expressivity and the cost of the numerical abstract domain which underlies the linear constraints labeling the decision nodes, and the choice of the auxiliary abstract domain for the leaf nodes. In this chapter, we propose various instances based on the numerical abstract domains presented in Section 3.4 for the decision nodes, and affine functions for the leaf nodes.

The following examples motivate the choice and illustrate the potential of piecewise-defined ranking functions.

Example 5.1.1

Let us consider the following program from [PR04a]:

$$\begin{array}{l} \texttt{while}\ ^{1}(x \geq 0)\ \texttt{do} \\ \quad ^{2}x := -2x + 10 \\ \texttt{od}^{3} \end{array}$$

The program is terminating since our program variables have integer values (cf. Section 3.1). In case we admitted non-integer values, the program would not terminate for $x = \frac{10}{3}$. However, the program does not have a linear ranking function (cf. [PR04a]). As a result, well-known methods to synthesize ranking functions like [PR04a, BMS05b], are not capable to guarantee its termination.

On the other hand, our method is not impaired from the fact that the program does not have a linear ranking function. The decision tree $t \in \mathcal{T}$ inferred at the initial program control point is depicted in Figure 5.1a. It represents the following piecewise-defined ranking function $\gamma_{\mathrm{T}}(t) \colon \mathcal{E} \rightharpoonup \mathbb{O}$, which proves that the program is terminating in at most nine program execution steps independently from the initial value for the program variable:

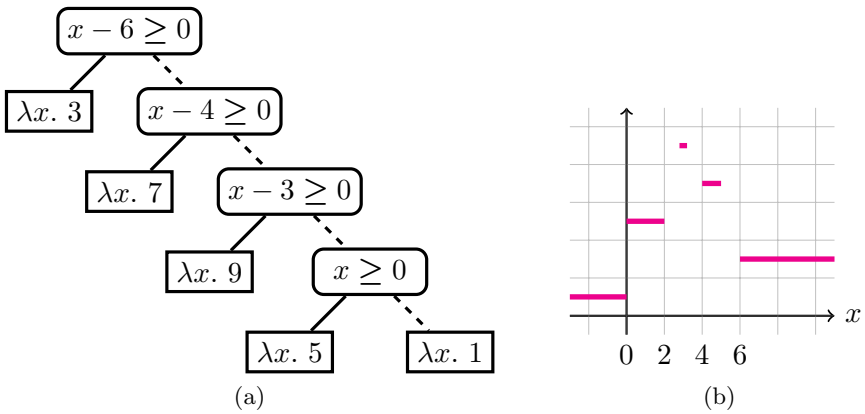


Figure 5.1: Decision tree representation (a) of the piecewise-defined ranking function (b) for the program of Example 5.1.1. The linear constraints are satisfied by their left subtree, while their right subtree satisfies their negation. The leaves of the tree represent partial functions whose domain is determined by the constraints satisfied along the path to the leaf node.

$$\gamma_{\mathrm{T}}(t) \quad \stackrel{\text{def}}{=} \quad \lambda\rho.\ \begin{cases} 1 & \rho(x) \leq -1 \\ 5 & 0 \leq \rho(x) \leq 2 \\ 9 & \rho(x) = 3 \\ 7 & 4 \leq \rho(x) \leq 5 \\ 3 & 6 \leq \rho(x) \end{cases}$$

The graphical representation of the ranking function is shown in Figure 5.1b.

The fully detailed analysis of the program uses interval constraints based on the intervals abstract domain (cf. Section 3.4.1) for the decision nodes, and a widening delay of five iterations. It is proposed in Example 5.3.4.

Example 5.1.2

Let us consider the following program from [UM14b]:

```
while ¹(r > 0) do
  ²r := r + x
  ³r := r − y
od⁴
```

At each loop iteration, the value of r is increased by the value of x and decreased by the value of y. The program is terminating if and only if $x < y$.

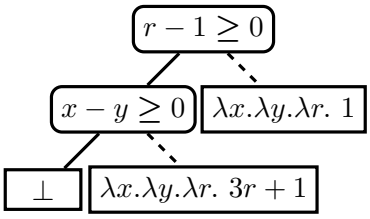


Figure 5.2: Decision tree representation of the piecewise-defined ranking function for the program of Example 5.1.2. The leaf with value $\perp$ explicitly represents the undefined piece of the ranking function determined by the constraints satisfied along the path to the leaf node.

Our method allows proving conditional termination. The decision tree $t \in \mathcal{T}$ inferred at the initial program control point is depicted in Figure 5.2. It represents the following piecewise-defined ranking function $\gamma_{\mathrm{T}}(t) \colon \mathcal{E} \rightharpoonup \mathbb{O}$:

$$\gamma_{\mathrm{T}}(t) \quad \stackrel{\text{def}}{=} \quad \lambda\rho. \begin{cases} 1 & \rho(r) \geq 1 \\ 3r + 1 & 0 \leq \rho(r) \wedge \rho(x) < \rho(y) \\ \text{undefined} & \text{otherwise} \end{cases}$$

which proves that the program is terminating in at most $3r + 1$ program execution steps if the initial value of the program variable x is smaller than the initial value of the program variable y. Note that the constraint $x < y$ does not explicitly appear in the program.

The fully detailed analysis of the program proposed in Example 5.1.2 uses polyhedral constraints based on the polyhedra abstract domain (cf. Section 3.4.2) for the decision nodes.

We emphasize that, as shown by the previous example, the partitioning induced by the decision trees is *semantic-based* rather than syntactic-based: the linear constraints labeling the decision nodes are automatically inferred by the analysis and do not necessarily appear in the program.

5.2 Decision Trees Abstract Domain

In the following, we give a more formal presentation of the decision trees abstract domain. To this end, we introduce the family of abstract domains $\mathbb{T}(\mathbb{D}, \mathbb{C}, \mathbb{F})$, parameterized by a family $\mathbb{C}$ of auxiliary abstract domains for the linear constraints labeling the decision nodes, and a family $\mathbb{F}$ of auxiliary abstract domains for the leaf nodes, both parameterized by a numerical abstract

domain $\mathbb{D}$ from Section 3.4. Adopting an OCaml terminology, each $\mathbb{T}$ is an abstract domain *functor*: a function mapping the parameter abstract domains $\mathbb{D}$, $\mathbb{C}$ and $\mathbb{F}$ into a new abstract domain $\mathbb{T}(\mathbb{D}, \mathbb{C}, \mathbb{F})$. $\mathbb{T}$ can be applied to various implementations of $\mathbb{D}$, $\mathbb{C}$ and $\mathbb{F}$ yielding the corresponding implementations of $\mathbb{T}(\mathbb{D}, \mathbb{C}, \mathbb{F})$, with no need for further programming effort. We first formally define the families $\mathbb{C}$, $\mathbb{F}$ and $\mathbb{T}$. Then, we present all abstract operators needed to manipulate decision nodes, leaf nodes, and decision trees.

5.2.1 Decision Trees

We first dive into some details on the families of auxiliary abstract domains $\mathbb{C}$ and $\mathbb{F}$. Then, we formally define the family of abstract domains $\mathbb{T}$.

Linear Constraints Auxiliary Abstract Domain. The family of auxiliary abstract domains $\mathbb{C}$ is itself a functor parameterized by $\mathbb{D}$, where $\mathbb{D}$ is any of the numerical abstract domains $\langle \mathcal{D}, \sqsubseteq_{\mathrm{D}} \rangle$. introduced in Section 3.4.

In the following, let $\mathcal{X} \stackrel{\text{def}}{=} \{X_1, \ldots, X_k\}$ be the set of program variables. The elements of the numerical abstract domains of Section 3.4 are equivalently represented as sets (i.e., conjunctions) of linear constraints of the form:

$$\begin{array}{cl} \pm X_i \geq c & \text{(intervals abstract domain)} \\ \pm X_i \pm X_j \geq c & \text{(octagons abstract domain)} \\ c_1 X_1 + \ldots + c_k X_k + c_{k+1} \geq 0 & \text{(polyhedra abstract domain)} \end{array}$$

where $c, c_1, \ldots, c_k, c_{k+1} \in \mathbb{Z}$. In the following, $\mathcal{C}_{\mathrm{B}} \stackrel{\text{def}}{=} \{\pm X_i \geq c \mid X_i \in \mathcal{X}, c \in \mathbb{Z}\}$ denotes the set of *interval constraints*, $\mathcal{C}_{\mathrm{O}} \stackrel{\text{def}}{=} \{\pm X_i \pm X_j \geq c \mid X_i, X_j \in \mathcal{X}, c \in \mathbb{Z}\}$ denotes the set of *octagonal constraints*, and $\mathcal{C}_{\mathrm{P}} \stackrel{\text{def}}{=} \{c_1 X_1 + \ldots + c_k X_k + c_{k+1} \geq 0 \mid X_1, \ldots, X_n \in \mathcal{X}, c_1, \ldots, c_k, c_{k+1} \in \mathbb{Z}\}$ denotes the set of *polyhedral constraints*. Let $\mathcal{C}$ be any of these sets of constraints. We have $\mathcal{C}_{\mathrm{B}} \subseteq \mathcal{C}_{\mathrm{O}} \subseteq \mathcal{C}_{\mathrm{P}}$. In particular, any interval constraint $\pm X_i \geq c$ is equivalent to the polyhedral constraint $0X_1 + \ldots \pm X_i + \ldots + 0X_k - c \geq 0$, and any octagonal constraint $\pm X_i \pm X_j \geq c$ is equivalent to the polyhedral constraint $0X_1 + \ldots \pm X_i + \ldots \pm X_j + \ldots 0X_k - c \geq 0$. Thus, in the following, we consider all linear constraints in $\mathcal{C}$ to have the form $c_1 X_1 + \ldots + c_k X_k + c_{k+1} \geq 0$. Moreover, in order to ensure a canonical representation of the linear constraints,

we require $\gcd(|c_1|, \ldots, |c_k|, |c_{k+1}|) = 1$:

$$\mathcal{C} \stackrel{\text{def}}{=} \left\{ c_1 X_1 + \ldots + c_k X_k + c_{k+1} \geq 0 \;\middle|\; \begin{array}{c} X_1, \ldots, X_k \in \mathcal{X}, \\ c_1, \ldots, c_k, c_{k+1} \in \mathbb{Z} \\ \gcd(|c_1|, \ldots, |c_k|, |c_{k+1}|) = 1 \end{array} \right\} \tag{5.2.1}$$

We can easily formalize the correspondence between the elements of the numerical abstract domains of Section 3.4 and sets of linear constraints as a Galois connection $\langle \mathcal{P}(\mathcal{C}), \sqsubseteq_{\mathrm{D}} \rangle \underset{\alpha_{\mathrm{C}}}{\overset{\gamma_{\mathrm{C}}}{\leftrightarrows}} \langle \mathcal{D}, \sqsubseteq_{\mathrm{D}} \rangle$, where the abstraction function $\alpha_{\mathrm{C}} \colon \mathcal{P}(\mathcal{C}) \to \mathcal{D}$ maps a set of interval (resp. octagonal, polyhedral) constraints to an interval (resp. an octagon, a polyhedron), and the concretization function $\gamma_{\mathrm{C}} \colon \mathcal{D} \to \mathcal{P}(\mathcal{C})$ maps an interval (resp. an octagon, a polyhedron) to a set of interval (resp. octagonal, polyhedral) constraints. In particular, we define $\gamma_{\mathrm{C}}(\top_{\mathrm{D}}) \stackrel{\text{def}}{=} \emptyset$ and $\gamma_{\mathrm{C}}(\bot_{\mathrm{D}}) \stackrel{\text{def}}{=} \{\bot_{\mathrm{C}}\}$, where $\bot_{\mathrm{C}}$ represents the unsatisfiable linear constraint $-1 \geq 0$.

The decision tree abstract domain is parametric in the choice between the expressivity and the cost of the set $\mathcal{C}$ of linear constraints chosen for labeling the decision nodes, which depends on the corresponding underlying numerical abstract domain $\langle \mathcal{D}, \sqsubseteq_{\mathrm{D}} \rangle$. As for boolean decision trees, where an ordering is imposed on all decision variables, we assume the set of program variables $\mathcal{X}$ to be totally ordered and we impose a *total order* $<_{\mathrm{C}}$ on $\mathcal{C}$. As an example, we define $<_{\mathrm{C}}$ to be the lexicographic order on the coefficients $c_1, \ldots, c_n$ and constant c_{k+1} of the linear constraints $c_1 X_1 + \ldots + c_k X_k + c_{k+1} \geq 0$:

$$\begin{aligned} a_1 X_1 + \ldots + a_k X_k + a_{k+1} \geq 0 \;\; <_{\mathrm{C}} \;\; b_1 X_1 + \ldots + b_k X_k + b_{k+1} \geq 0 \\ \iff \exists j > 0 : \forall i < j : (a_i = b_i) \wedge (a_j < b_j) \end{aligned} \tag{5.2.2}$$

Thus, any set $\mathcal{C}$ equipped with $<_{\mathrm{C}}$ forms a totally ordered set $\langle \mathcal{C}, <_{\mathrm{C}} \rangle$.

We define the *negation* $\neg c$ of a linear constraint c as follows:

$$\neg(c_1 X_1 + \ldots + c_k X_k + c_{k+1} \geq 0) \stackrel{\text{def}}{=} -c_1 X_1 - \ldots - c_k X_k - c_{k+1} - 1 \geq 0 \tag{5.2.3}$$

Note that, we decrement the value of the constant of the negated linear constraint because our program variables have integer values (cf. Section 3.1).

Example 5.2.1

Let $\mathcal{X} \stackrel{\text{def}}{=} \{x\}$ and let us consider the linear constraint $x - 2 \geq 0$. Its negation $\neg(x - 2 \geq 0)$ is the linear constraint $-x + 1 \geq 0$ (i.e., $x \leq 1$).

In order to ensure a *canonical representation* of the decision trees, we forbid a linear constraint c and its negation $\neg c$ from simultaneously appearing in a decision tree. As an example, between c and $\neg c$, we keep only the largest constraint with respect to the total order $<_{\mathrm{C}}$. We define the following equivalence relation between a linear constraint and its negation:

$$c_1 \equiv_{\mathrm{C}} c_2 \stackrel{\text{def}}{=} c_1 = \neg c_2 \wedge c_2 = \neg c_1 \tag{5.2.4}$$

Let $\mathbb{C}$ denote any totally ordered set $\langle \mathcal{C}/\equiv_{\mathrm{C}}, <_{\mathrm{C}} \rangle$.

Functions Auxiliary Abstract Domain. The family of abstract domains $\mathbb{F}$ is also a functor parameterized by any of the numerical abstract domains $\mathbb{D}$ introduced in Section 3.4. It is dedicated to the manipulation of the leaf nodes of a decision tree with respect to the set of linear constraints satisfied along their paths from the root of the decision tree.

The elements of these abstract domains belong to the following set:

$$\mathcal{F} \stackrel{\text{def}}{=} \{\bot_{\mathrm{F}}\} \cup \left(\mathbb{Z}^{|\mathcal{X}|} \to \mathbb{N}\right) \cup \{\top_{\mathrm{F}}\}. \tag{5.2.5}$$

which consists of *natural-valued* functions of the program variables, plus the element $\bot_{\mathrm{F}}$ and the element $\top_{\mathrm{F}}$, whose meaning will be explained shortly. In the following, the leaf nodes belonging to $\mathcal{F} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}$ and $\{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}$ are referred to as *defined* and *undefined* leaf nodes, respectively. Moreover, the undefined leaf nodes belonging to $\{\bot_{\mathrm{F}}\}$ are called $\bot_{\mathrm{F}}$-leaves and those belonging to $\{\top_{\mathrm{F}}\}$ are called $\top_{\mathrm{F}}$-leaves.

As an instance, in the following we consider *affine* functions:

$$\mathcal{F}_{\mathrm{A}} \stackrel{\text{def}}{=} \left\{ f \colon \mathbb{Z}^{|\mathcal{X}|} \to \mathbb{N} \;\middle|\; f(X_1, \ldots, X_k) = m_1 X_1 + \ldots + m_k X_k + q \right\} \cup \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}. \tag{5.2.6}$$

We now define a *computational order* $\sqsubseteq_{\mathrm{F}}$ and an *approximation order* $\preccurlyeq_{\mathrm{F}}$ where $\bot_{\mathrm{F}}$-leaves and $\top_{\mathrm{F}}$-leaves are comparable and incomparable, respectively. These orders are parameterized by a numerical abstract domain element $D \in \mathcal{D}$, which represents the linear constraints satisfied along the path to the compared leaf nodes. Intuitively, these orders abstract the computational order $\sqsubseteq$ defined in Equation 4.2.15 and the approximation order $\preccurlyeq$ defined in Equation 4.2.12, respectively. We have observed in Lemma 4.2.22 that $\sqsubseteq$ and $\preccurlyeq$ coincide when the ranking functions have the same domain. Analogously, the approximation order $\preccurlyeq_{\mathrm{F}}$ and the computational order $\sqsubseteq_{\mathrm{F}}$ between *defined*

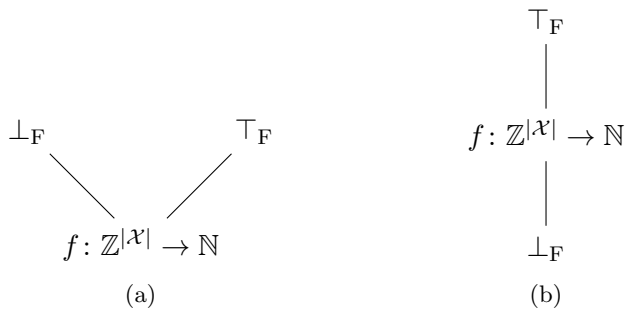


Figure 5.3: Hasse diagrams defining the approximation order $\preccurlyeq_{\mathrm{F}}[D]$ (a) and the computational order $\sqsubseteq_{\mathrm{F}}[D]$ (b) of the functions abstract domain.

leaf nodes are identical and defined as follows:

$$f_1 \preccurlyeq_{\mathrm{F}}[D]\ f_2 \iff \forall \rho \in \gamma_{\mathrm{D}}(D) : f_1(\rho(x_1), \ldots, \rho(x_k)) \leq f_2(\rho(x_1), \ldots, \rho(x_k)) \tag{5.2.7}$$

$$f_1 \sqsubseteq_{\mathrm{F}}[D]\ f_2 \iff \forall \rho \in \gamma_{\mathrm{D}}(D) : f_1(\rho(x_1), \ldots, \rho(x_k)) \leq f_2(\rho(x_1), \ldots, \rho(x_k)). \tag{5.2.8}$$

Instead, when one or both leaf nodes are undefined, the approximation and computational order are defined by the Hasse diagrams in Figure 5.3a and Figure 5.3b, respectively. Note that, as we mentioned, in the approximation order $\bot_{\mathrm{F}}$-leaves and $\top_{\mathrm{F}}$-leaves are incomparable.

A leaf node, together with its path from the root of the decision tree, represents a (piece of a) partial ordinal-valued functions of environments. We define the following concretization-based abstraction:

$$\langle \mathcal{E} \rightharpoonup \mathbb{O}, \preccurlyeq \rangle \xleftarrow{\gamma_{\mathrm{F}}[D]} \langle \mathcal{F}, \preccurlyeq_{\mathrm{F}} \rangle$$

where $D \in \mathcal{D}$ represents the path to the leaf node. The concretization function $\gamma_{\mathrm{F}} \colon \mathcal{D} \to \mathcal{F} \to (\mathcal{E} \rightharpoonup \mathbb{O})$ is defined as follows:

$$\begin{aligned} \gamma_{\mathrm{F}}[D]\bot_{\mathrm{F}} &\stackrel{\text{def}}{=} \dot{\emptyset} \\ \gamma_{\mathrm{F}}[D]f &\stackrel{\text{def}}{=} \lambda \rho \in \gamma_{\mathrm{D}}(D) : f(\rho(x_1), \ldots, \rho(x_k)) \\ \gamma_{\mathrm{F}}[D]\top_{\mathrm{F}} &\stackrel{\text{def}}{=} \dot{\emptyset} \end{aligned} \tag{5.2.9}$$

Note that, both γ_{F} and $\top_{\mathrm{F}}$ represent the totally undefined function.

The following result proves that, given $D \in \mathcal{D}$, $\gamma_{\mathrm{F}}[D]$ is monotonic:

Lemma 5.2.2 $\forall f_1, f_2 \in \mathcal{F} : f_1 \preccurlyeq_F [D]\ f_2 \Rightarrow \gamma_F[D]f_1 \preccurlyeq \gamma_F[D]f_2$.

Proof.
See Appendix A.3. ■

Note that, for now, we are approximating ordinal-valued functions by means of natural-valued functions. The next Chapter 6, will be dedicated to abstractions based on ordinal-valued functions.

Decision Trees Abstract Domain. We can now use the families of domains $\mathbb{D}$, $\mathbb{C}$ and $\mathbb{F}$ to build the decision trees abstract domains $\mathbb{T}(\mathbb{D}, \mathbb{C}, \mathbb{F})$.

The elements of these abstract domains belong to the following set:

$$\mathcal{T} \stackrel{\text{def}}{=} \{\text{LEAF} : f \mid f \in \mathcal{F}\} \cup \{\text{NODE}\{c\} : t_1; t_2 \mid c \in \mathcal{C}, t_1, t_2 \in \mathcal{T}\} \tag{5.2.10}$$

where $\mathcal{F}$ is defined in Equation 5.2.5 and $\mathcal{C}$ is defined in Equation 5.2.1. A *decision tree* $t \in \mathcal{T}$ is either a *leaf node* $\text{LEAF} : f$, with f an element of $\mathcal{F}$ (in the following denoted by $t.f$), or a *decision node* $\text{NODE}\{c\} : t_1; t_2$, such that c is a linear constraint in $\mathcal{C}$ (in the following denoted by $t.c$) and the left subtree t_1 and the right subtree t_2 (in the following denoted by $t.l$ and $t.r$, respectively) belong to $\mathcal{T}$. In addition, given a decision tree $\text{NODE}\{c\} : t_1; t_2$, we impose that the linear constraint $c \in \mathcal{C}$ is always the largest constraint, with respect to $<_{\text{C}}$, appearing in the tree. In the following, let $\bot_{\text{T}} \stackrel{\text{def}}{=} \text{LEAF} : \bot_{\text{F}}$.

A decision tree represents a partial ordinal-valued function of environments. We define the following concretization-based abstraction:

$$\langle \mathcal{E} \rightharpoonup \mathbb{O}, \preccurlyeq \rangle \xleftarrow{\gamma_{\text{T}}[D]} \langle \mathcal{T}, \preccurlyeq_{\text{T}} \rangle$$

where $D \in \mathcal{D}$ represent an over-approximation of the reachable environments. The concretization function $\gamma_{\text{T}} \colon \mathcal{D} \to \mathcal{T} \to (\mathcal{E} \rightharpoonup \mathbb{O})$ is defined as follows:

$$\gamma_{\text{T}}[D]t \stackrel{\text{def}}{=} \bar{\gamma}_{\text{T}}[\gamma_{\text{C}}(D)]t \tag{5.2.11}$$

where $\bar{\gamma}_{\text{T}} \colon \mathcal{D} \to \mathcal{T} \to (\mathcal{E} \rightharpoonup \mathbb{O})$. The function $\bar{\gamma}_{\text{T}} \colon \mathcal{D} \to \mathcal{T} \to (\mathcal{E} \rightharpoonup \mathbb{O})$ accumulates into a set $C \in \mathcal{P}(\mathcal{C})$ (initially equal to $\gamma_{\text{C}}(D)$) the linear constraints satisfied along the paths of the decision tree up to a leaf node, where the concretization function $\gamma_{\text{F}} \colon \mathcal{D} \to \mathcal{F} \to (\mathcal{E} \rightharpoonup \mathbb{O})$ (cf. Equation 5.2.9) returns a partially defined function over $\gamma_{\text{D}}(\alpha_{\text{C}}(C))$:

$$\bar{\gamma}_{\text{T}}[C]\text{LEAF} : f \stackrel{\text{def}}{=} \gamma_{\text{F}}[\alpha_{\text{C}}(C)]f$$

$$\bar{\gamma}_{\text{T}}[C]\text{NODE}\{c\} : t_1; t_2 \stackrel{\text{def}}{=} \bar{\gamma}_{\text{T}}[C \cup \{c\}]t_1 \dot{\cup} \bar{\gamma}_{\text{T}}[C \cup \{\neg c\}]t_2$$

The approximation order $\preccurlyeq_{\mathrm{T}}$ will be formally defined in the next section.

5.2.2 Binary Operators

In the following, we define the decision tree computational and approximation ordering. Moreover, we define *binary* operators for the computational and approximation join, and for the meet of decision trees.

The binary operators manipulate elements belonging to the following set:

$$\mathcal{T}_{\mathrm{NIL}} \stackrel{\mathrm{def}}{=} \{\mathrm{NIL}\} \cup \{\mathrm{LEAF} : f \mid f \in \mathcal{F}\} \cup \{\mathrm{NODE}\{c\} : t_1; t_2 \mid c \in \mathcal{C}, t_1, t_2 \in \mathcal{T}_{\mathrm{NIL}}\} \tag{5.2.12}$$

A *partial decision tree* $t \in \mathcal{T}_{\mathrm{NIL}}$ is either an *empty* tree NIL, or a leaf node *leaf node* $\mathrm{LEAF} : f$, with f an element of $\mathcal{F}$, or a *decision node* $\mathrm{NODE}\{c\} : t_1; t_2$, such that c is a linear constraint in $\mathcal{C}$ and the left subtree t_1 and the right subtree t_2 belong to $\mathcal{T}_{\mathrm{NIL}}$. Note that $\mathcal{T} \subseteq \mathcal{T}_{\mathrm{NIL}}$. Intuitively, the special element NIL represent the absence of information regarding some partition of the domain of the ranking function. Note that, an empty tree NIL is different from an undefined leaf node $\mathrm{LEAF} : \bot_{\mathrm{F}}$ or $\mathrm{LEAF} : \top_{\mathrm{F}}$, since an undefined leaf node provides the information that the ranking function is undefined. The NIL nodes serve an algorithmic purpose. They allow carving out parts of decision trees and stitching them up on disjoint domains. In particular, NIL nodes are only temporary. They are inserted in intermediate trees during computations but disappear before the abstract operators return.

Tree Unification. The decision tree orderings and binary operators are based on Algorithm 1 for *tree unification*: the main function UNIFICATION, given an over-approximation $D \in \mathcal{D}$ of the reachable environments and two decision trees $t_1, t_2 \in \mathcal{T}_{\mathrm{NIL}}$, calls the auxiliary UNIFICATION-AUX to find a common refinement for the trees.

The function UNIFICATION-AUX accumulates into a set $C \in \mathcal{P}(\mathcal{C})$ (initially equal to $\gamma_{\mathrm{C}}(D)$, cf. Line 36) the linear constraints encountered along the paths of the decision trees (cf. Lines 12-13, Lines 21-22, Lines 31-32), possibly adding decision nodes (cf. Line 14, Line 23) or removing constraints that are redundant (cf. Line 7, Line 16, Line 26) or whose negation is redundant (cf. Line 9, Line 18, Line 28) with respect to C.

The redundancy check is performed by the function isRedundant which, given a linear constraint $c \in \mathcal{C}$ and a set of linear constraints $C \in \mathcal{P}(\mathcal{C})$, tests the inclusion of the corresponding numerical abstract domain elements $\alpha_{\mathrm{C}}(C) \in \mathcal{D}$ and $\alpha_{\mathrm{C}}(\{c\}) \in \mathcal{D}$ (cf. Line 2).

Algorithm 1 : Tree Unification

```
1:  function isRedundant(c, C)                                  ▷ c ∈ 𝒞, C ∈ 𝒫(𝒞)
2:      return α_C(C) ⊑_D α_C({c})
3:
4:  function unification-aux(t_1, t_2, C)                       ▷ t_1, t_2 ∈ 𝒯_NIL, C ∈ 𝒫(𝒞)
5:      if ¬isNode(t_1) ∧ ¬isNode(t_2) then return (t_1, t_2)
6:      else if ¬isNode(t_1) ∨ (isNode(t_1) ∧ isNode(t_2) ∧ t_1.c <_C t_2.c) then
7:          if isRedundant(t_2.c, C) then
8:              return unification-aux(t_1, t_2.l, C)
9:          else if isRedundant(¬t_2.c, C) then
10:             return unification-aux(t_1, t_2.r, C)
11:         else                                  ▷ t_2.c can be added to t_1 and kept in t_2
12:             (l_1, l_2) ← unification-aux(t_1, t_2.l, C ∪ {t_2.c})
13:             (r_1, r_2) ← unification-aux(t_1, t_2.r, C ∪ {¬t_2.c})
14:             return (node{t_2.c} : l_1; r_1, node{t_2.c} : l_2; r_2)
15:     else if ¬isNode(t_2) ∨ (isNode(t_1) ∧ isNode(t_2) ∧ t_2.c <_C t_1.c) then
16:         if isRedundant(t_1.c, C) then
17:             return unification-aux(t_1.l, t_2, C)
18:         else if isRedundant(¬t_1.c, C) then
19:             return unification-aux(t_1.r, t_2, C)
20:         else                                  ▷ t_1.c can be kept in t_1 and added to t_2
21:             (l_1, l_2) ← unification-aux(t_1.l, t_2, C ∪ {t_1.c})
22:             (r_1, r_2) ← unification-aux(t_1.r, t_2, C ∪ {¬t_1.c})
23:             return (node{t_1.c} : l_1; r_1, node{t_1.c} : l_2; r_2)
24:     else if isNode(t_1) ∧ isNode(t_2) then
25:         c ← t_1.c                                 ▷ t_1.c and t_2.c are equal
26:         if isRedundant(c, C) then
27:             return unification-aux(t_1.l, t_2.l, C)
28:         else if isRedundant(¬c, C) then
29:             return unification-aux(t_1.r, t_2.r, C)
30:         else                                      ▷ c can be kept in t_1 and t_2
31:             (l_1, l_2) ← unification-aux(t_1.l, t_2.l, C ∪ {c})
32:             (r_1, r_2) ← unification-aux(t_1.r, t_2.r, C ∪ {¬c})
33:             return (node{c} : l_1; r_1, node{c} : l_2; r_2)
34:
35: function unification(D, t_1, t_2)                           ▷ D ∈ 𝒟, t_1, t_2 ∈ 𝒯_NIL
36:     return unification-aux(t_1, t_2, γ_C(D))
```

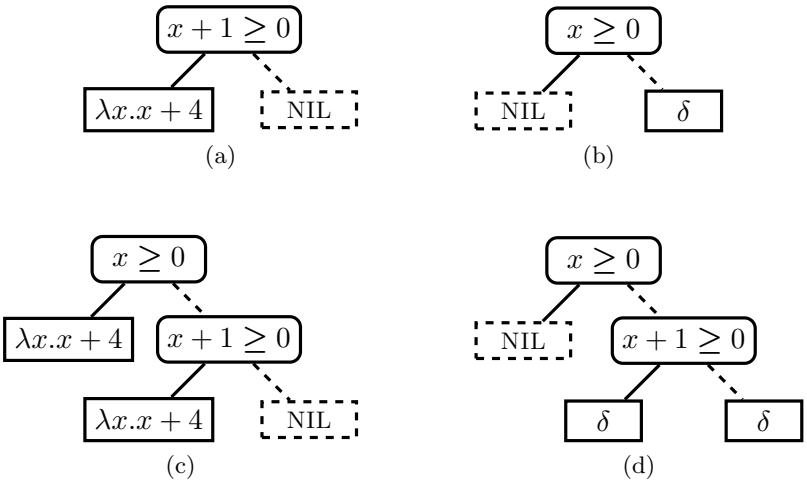


Figure 5.4: Tree unification of the decision trees (a) and (b) of Example 5.2.3. The resulting decision trees are represented in (c) and (d), respectively.

Note that the tree unification does not loose any information. Then, the binary operations are carried out "leaf-wise" on the unified decision trees.

Example 5.2.3

Let $\mathcal{X} \stackrel{\text{def}}{=} \{x\}$, let $\mathbb{D}$ be the intervals abstract domain $\langle \mathcal{B}, \sqsubseteq_{\mathrm{B}} \rangle$, let $\mathbb{C}$ be the interval constraints auxiliary abstract domain $\langle \mathcal{C}_{\mathrm{B}}/\equiv_{\mathrm{C}}, <_{\mathrm{C}} \rangle$, and let $\mathbb{F}$ be the affine functions auxiliary abstract domain $\langle \mathcal{F}_{\mathrm{A}}, \preccurlyeq_{\mathrm{F}} \rangle$. We consider the decision trees $t_1 \in \mathcal{T}_{\mathrm{NIL}}$ represented in Figure 5.4a and $t_2 \in \mathcal{T}_{\mathrm{NIL}}$ represented in Figure 5.4b, where $\delta \in \mathcal{T}$ is a leaf node. Let $d \stackrel{\text{def}}{=} \top_{\mathrm{B}}$ and let $C \stackrel{\text{def}}{=} \gamma_{\mathrm{C}}(d) = \emptyset$.

The function UNIFICATION-AUX adds a decision node for $x \geq 0$ to t_1 (cf. Line 14) and removes the redundant constraint $x+1 \geq 0$ from the resulting left subtree (cf. Line 8). Moreover, UNIFICATION-AUX adds a decision node for $x+1 \geq 0$ to the right subtree of t_2 (cf. Line 23). The result of the tree unification are the decision trees represented in Figure 5.4c and Figure 5.4d, respectively.

Ordering. The decision tree ordering is implemented by Algorithm 2: the function ORDER is parameterized by the choice of the ordering $\trianglelefteq$ between leaf nodes and, given a sound over-approximation $D \in \mathcal{D}$ of the reachable environments and two decision trees $t_1, t_2 \in \mathcal{T}$, calls the function UNIFICATION for tree unification (cf. Line 11) and then calls the auxiliary function ORDER-AUX. The function ORDER-AUX accumulates into a set $C \in \mathcal{P}(\mathcal{C})$ (initially

Algorithm 2 : Tree Order

```
1:  function ORDER-AUX(⊴, t1, t2, C)        ▷ t1, t2 ∈ 𝒯, C ∈ 𝒫(𝒞), ⊴ ∈ {≼F, ⊑F}
2:      if ISLEAF(t1) ∧ ISLEAF(t2) then
3:          return LEAF : t1.f ⊴ [αC(C)] t2.f
4:      else if ISNODE(t1) ∧ ISNODE(t2) then
5:          c ← t1.c
6:          l ← ORDER-AUX(t1.l, t2.l, C ∪ {c})
7:          r ← ORDER-AUX(t1.r, t2.r, C ∪ {¬c})
8:          return l ∧ r
9:
10: function ORDER(⊴, D, t1, t2)            ▷ D ∈ 𝒟, t1, t2 ∈ 𝒯, ⊴ ∈ {≼F, ⊑F}
11:     (t1, t2) ← UNIFICATION(D, t1, t2)
12:     return ORDER-AUX(⊴, t1, t2, γC(D))
```

Algorithm 3 : Tree Approximation Order

```
1: function A-ORDER(D, t1, t2)              ▷ D ∈ 𝒟, t1, t2 ∈ 𝒯
2:     return ORDER(≼F, D, t1, t2)
```

equal to $\gamma_{\mathrm{C}}(D)$, cf. Line 12) the linear constraints encountered along the paths of the decision trees (cf. Lines 6-7) up to the leaf nodes, which are compared by means of the chosen ordering $\trianglelefteq$ (cf. Line 3).

In particular, the approximation ordering $\preccurlyeq_{\mathrm{T}}$ and the computational ordering $\sqsubseteq_{\mathrm{T}}$ are implemented by Algorithm 3 and Algorithm 4, respectively: the functions A-ORDER of Algorithm 3 and C-ORDER of Algorithm 4 call the function ORDER of Algorithm 2 choosing respectively the approximation order $\preccurlyeq_{\mathrm{F}}$ (cf. Equation 5.2.7 and Figure 5.3a) and the computational order $\sqsubseteq_{\mathrm{F}}$ (cf. Equation 5.2.8 and Figure 5.3b).

The following result proves that, given $D \in \mathcal{D}$, the concretization function $\gamma_{\mathrm{T}}[D]$ is monotonic with respect to the approximation order $\preccurlyeq_{\mathrm{T}}[D]$:

Lemma 5.2.4 $\forall t_1, t_2 \in \mathcal{T} : t_1 \preccurlyeq_T [D]\ t_2 \Rightarrow \gamma_T[D] f_1 \preccurlyeq \gamma_T[D] f_2$.

Proof.
See Appendix A.3. ■

Join. The join of decision trees represents a ranking function defined over the *union* of their partitions. It is implemented by Algorithm 5: the function

Algorithm 4 : Tree Computational Order

```
function C-ORDER(D, t1, t2)                    ▷ D ∈ 𝒟, t1, t2 ∈ 𝒯
    return ORDER(⊑_F, D, t1, t2)
```

Algorithm 5 : Tree Join

```
1:  function JOIN-AUX(⊕, t1, t2, C)   ▷ t1, t2 ∈ 𝒯_NIL, C ∈ 𝒫(𝒞), ⊕ ∈ {ϒ_F, ⊔_F}
2:      if ISNIL(t1) then return t2
3:      else if ISNIL(t2) then return t1
4:      else if ISLEAF(t1) ∧ ISLEAF(t2) then
5:          return LEAF : t1.f ⊕[α_C(C)] t2.f
6:      else if ISNODE(t1) ∧ ISNODE(t2) then
7:          c ← t1.c
8:          (l1, l2) ← JOIN-AUX(t1.l, t2.l, C ∪ {c})
9:          (r1, r2) ← JOIN-AUX(t1.r, t2.r, C ∪ {¬c})
10:         return (NODE{c} : l1; r1, NODE{c} : l2; r2)
11:
12: function JOIN(⊕, D, t1, t2)          ▷ D ∈ 𝒟, t1, t2 ∈ 𝒯_NIL, ⊕ ∈ {ϒ_F, ⊔_F}
13:     (t1, t2) ← UNIFICATION(D, t1, t2)
14:     return JOIN-AUX(⊕, t1, t2, γ_C(D))
```

JOIN is parameterized by the choice of the join $\oplus$ between leaf nodes, which will be presented shortly. Given a sound over-approximation $D \in \mathcal{D}$ of the reachable environments and two partial decision trees $t_1, t_2 \in \mathcal{T}_{\text{NIL}}$, JOIN first calls UNIFICATION (cf. Line 13) and then calls the auxiliary function JOIN-AUX. The latter collects the set $C \in \mathcal{P}(\mathcal{C})$ (initially equal to $\gamma_{\text{C}}(D)$, cf. Line 14) of linear constraints encountered up to the leaf nodes (cf. Lines 8-9), which are joined by the chosen operator $\oplus$ (cf. Line 5). In case it encounters an empty tree, JOIN-AUX favors the possibly non-empty one (cf. Lines 2-3).

We define the *approximation join* operator $\curlyvee_{\text{F}}$ and the *computational join* operator $\sqcup_{\text{F}}$ between leaf nodes. The approximation join is the least upper bound for the approximation order (cf. Figure 5.3a and Equation 5.2.7), and the computational join is the least upper bound for the computational order (cf. Figure 5.3b and Equation 5.2.8). These operators are parameterized by a numerical abstraction $D \in \mathcal{D}$, which represents the linear constraints satisfied along the path to the leaf nodes (cf. Line 5 of Algorithm 5).

The approximation and computational join differ when joining defined and

undefined leaf nodes. In this case, the approximation join is defined as follows:

$$\begin{aligned}
\bot_{\mathrm{F}} \curlyvee_{\mathrm{F}}[D]\ f &\stackrel{\mathrm{def}}{=} \bot_{\mathrm{F}} && f \in \mathcal{F} \setminus \{\top_{\mathrm{F}}\} \\
f \curlyvee_{\mathrm{F}}[D]\ \bot_{\mathrm{F}} &\stackrel{\mathrm{def}}{=} \bot_{\mathrm{F}} && f \in \mathcal{F} \setminus \{\top_{\mathrm{F}}\} \\
\top_{\mathrm{F}} \curlyvee_{\mathrm{F}}[D]\ f &\stackrel{\mathrm{def}}{=} \top_{\mathrm{F}} && f \in \mathcal{F} \setminus \{\bot_{\mathrm{F}}\} \\
f \curlyvee_{\mathrm{F}}[D]\ \top_{\mathrm{F}} &\stackrel{\mathrm{def}}{=} \top_{\mathrm{F}} && f \in \mathcal{F} \setminus \{\bot_{\mathrm{F}}\}
\end{aligned} \tag{5.2.13}$$

and the computational join is defined as follows:

$$\begin{aligned}
\bot_{\mathrm{F}} \sqcup_{\mathrm{F}}[D]\ f &\stackrel{\mathrm{def}}{=} f && f \in \mathcal{F} \\
f \sqcup_{\mathrm{F}}[D]\ \bot_{\mathrm{F}} &\stackrel{\mathrm{def}}{=} f && f \in \mathcal{F} \\
\top_{\mathrm{F}} \sqcup_{\mathrm{F}}[D]\ f &\stackrel{\mathrm{def}}{=} \top_{\mathrm{F}} && f \in \mathcal{F} \\
f \sqcup_{\mathrm{F}}[D]\ \top_{\mathrm{F}} &\stackrel{\mathrm{def}}{=} \top_{\mathrm{F}} && f \in \mathcal{F}
\end{aligned} \tag{5.2.14}$$

In particular, note that the approximation join is undefined when joining $\bot_{\mathrm{F}}$-leaves and $\top_{\mathrm{F}}$-leaves and always favors the undefined leaf nodes.

Instead, given two *defined* leaf nodes $f_1, f_2 \in \mathcal{F} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}$, their approximation join $f_1 \curlyvee_{\mathrm{F}}[D]\ f_2$ and their computational join $f_1 \sqcup_{\mathrm{F}}[D]\ f_2$ coincide and they are defined as their least upper bound $f \in \mathcal{F} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}$ or, when such least upper bound does not exist, $\top_{\mathrm{F}}$:

$$f_1 \curlyvee_{\mathrm{F}}[D]\ f_2 \stackrel{\mathrm{def}}{=} \begin{cases} f & f \in \mathcal{F} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\} \\ \top_{\mathrm{F}} & \text{otherwise} \end{cases} \tag{5.2.15}$$

$$f_1 \sqcup_{\mathrm{F}}[D]\ f_2 \stackrel{\mathrm{def}}{=} \begin{cases} f & f \in \mathcal{F} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\} \\ \top_{\mathrm{F}} & \text{otherwise} \end{cases} \tag{5.2.16}$$

where $f \stackrel{\mathrm{def}}{=} \lambda\rho \in \gamma_{\mathrm{D}}(D).\ \max\{f_1(\rho(X_1), \ldots, \rho(X_k)), f_2(\rho(X_1), \ldots, \rho(X_k))\}$.

As an instance, let us consider two affine functions $f_1, f_2 \in \mathcal{F}_{\mathrm{A}} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}$. Let $V \notin \mathcal{X}$ be a special variable not appearing in any program. We define the *hypograph* of a given affine function $f \in \mathcal{F}_{\mathrm{A}} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}$ within a given numerical abstraction $D \in \mathcal{D}$ as $f[D]{\downarrow} \stackrel{\mathrm{def}}{=} \{(X_1, \ldots, X_k, V) \mid \gamma_{\mathrm{C}}(D), V \leq f(X_1, \ldots, X_k)\}$. The approximation join $f_1 \curlyvee_{\mathrm{F}}[D]\ f_2$ and computational join $f_1 \sqcup_{\mathrm{F}}[D]\ f_2$ of f_1 and f_2 are identical and defined as the affine function $f \in \mathcal{F}_{\mathrm{A}} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}$, whose hypograph $f[D]{\downarrow}$ is the *convex-hull* of the hypographs $f_1[D]{\downarrow}$ and $f_2[D]{\downarrow}$ of f_1 and f_2, respectively; when such function does not exist, the result is $\top_{\mathrm{F}}$:

$$f_1 \curlyvee_{\mathrm{F}}[D]\ f_2 \stackrel{\mathrm{def}}{=} \begin{cases} f & f[D]{\downarrow} = \textit{convex-hull}\{f_1[D]{\downarrow}, f_2[D]{\downarrow}\} \\ \top_{\mathrm{F}} & \text{otherwise} \end{cases} \tag{5.2.17}$$

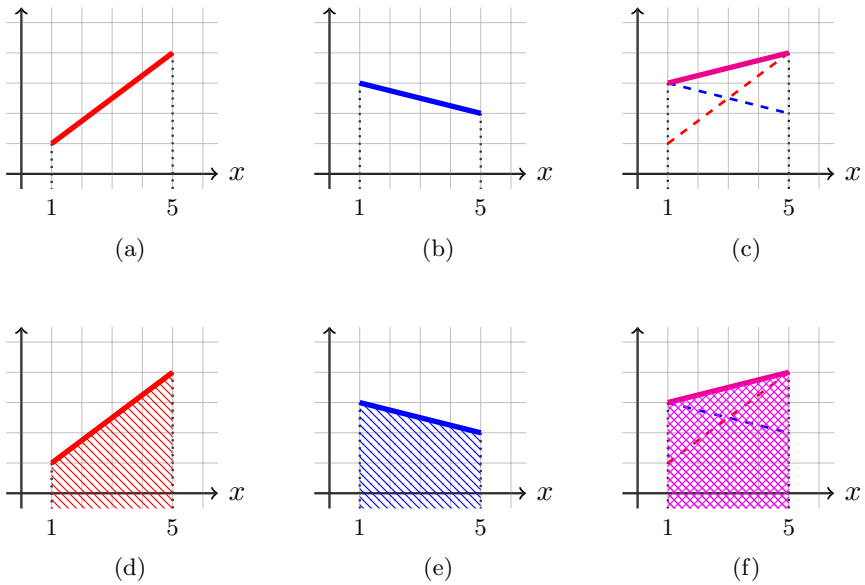


Figure 5.5: Example of join of two affine functions of one variable, shown in (a) and (b), respectively. The result is shown in (c). In (d), (e), and (f) are respectively shown the hypographs of (a), (b), and (c).

$$f_1 \sqcup_{\mathrm{F}}[D]\ f_2 \stackrel{\text{def}}{=} \begin{cases} f & f[D]{\downarrow} = \textit{convex-hull}\{f_1[D]{\downarrow}, f_2[D]{\downarrow}\} \\ \top_{\mathrm{F}} & \text{otherwise} \end{cases} \tag{5.2.18}$$

To clarify, let us consider the following example:

Example 5.2.5

Two affine functions $f_1, f_2 \in \mathcal{F}_{\mathrm{A}} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}$ are shown in Figure 5.5a and Figure 5.5b, respectively. Their domain of definition $D \in \mathcal{D}$ is represented by the set of linear constraints $C \stackrel{\text{def}}{=} \gamma_{\mathrm{C}}(D) = \{x - 1 \geq 0, -x + 5 \geq 0\}$. In Figure 5.5d and Figure 5.5e are represented the hypographs $f_1[D]{\downarrow}$ and $f_2[D]{\downarrow}$ of f_1 and f_2, respectively. Their convex-hull is shown in Figure 5.5e, and the result of the join is the affine function $f \in \mathcal{F}_{\mathrm{A}} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}$ shown in Figure 5.5c.

Example 5.2.6

Let $\mathcal{X} = \{x\}$, and let $f_1 \stackrel{\text{def}}{=} \lambda x.\ x$ and $f_2 \stackrel{\text{def}}{=} \lambda x.\ -x$ be two affine functions, whose domain of definition $D \in \mathcal{D}$ is represented by the empty set of linear constraints $C \stackrel{\text{def}}{=} \gamma_{\mathrm{C}}(D) = \emptyset$. Then, since there is no affine function which is

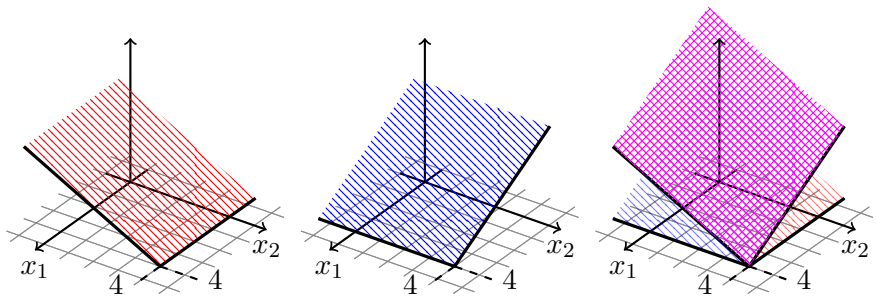


Figure 5.6: Example of join of two affine functions of two variables, shown in (a) and (b), respectively. The result is shown in (c).

Algorithm 6 : Tree Approximation Join

1: **function** A-JOIN(D, t_1, t_2) ▷ $D \in \mathcal{D}$, $t_1, t_2 \in \mathcal{T}_{\text{NIL}}$
2: **return** JOIN$(\curlyvee_{\text{F}}, D, t_1, t_2)$

the least upper bound of f_1 and f_2, the result of the join is $\top_{\text{F}}$.

Another example of join of affine functions is proposed in Figure 5.6.

The approximation join $\curlyvee_{\text{T}}$ and the computational join $\sqcup_{\text{T}}$ of decision trees are implemented by Algorithm 6 and Algorithm 7, respectively: the functions A-JOIN of Algorithm 6 and C-JOIN of Algorithm 7 call the function JOIN of Algorithm 5 choosing respectively the approximation join $\curlyvee_{\text{F}}$ and the computational join $\sqcup_{\text{F}}$ between leaf nodes.

Example 5.2.7

Let $\mathcal{X} \stackrel{\text{def}}{=} \{x\}$, let $\mathbb{D}$ be the intervals abstract domain $\langle \mathcal{B}, \sqsubseteq_{\text{B}} \rangle$, let $\mathbb{C}$ be the interval constraints auxiliary abstract domain $\langle \mathcal{C}_{\text{B}}/\equiv_{\text{C}}, <_{\text{C}} \rangle$, and let $\mathbb{F}$ be the affine functions auxiliary abstract domain $\langle \mathcal{F}_{\text{A}}, \preccurlyeq_{\text{F}} \rangle$. We consider the decision trees $t_1 \in \mathcal{T}$ represented in Figure 5.4a and $t_2 \in \mathcal{T}$ represented in Figure 5.4b, where $\delta \in \mathcal{T}$ is a leaf node. Let $d \stackrel{\text{def}}{=} \top_{\text{B}}$ and let $C \stackrel{\text{def}}{=} \gamma_{\text{C}}(d) = \emptyset$.

From Example 5.2.3, the result of the tree unification are the decision trees represented in Figure 5.4c and Figure 5.4d, respectively. The decision tree returned by the function A-JOIN-AUX is represented in Figure 5.7.

Algorithm 7 : Tree Computational Join

1: **function** C-JOIN(D, t_1, t_2) ▷ $D \in \mathcal{D}$, $t_1, t_2 \in \mathcal{T}_{\text{NIL}}$
2: **return** JOIN$(\sqcup_{\text{F}}, D, t_1, t_2)$

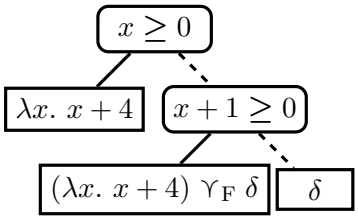


Figure 5.7: Tree approximation join of Figure 5.4a and Figure 5.4b.

Meet. The meet of decision trees represents a ranking function defined over the *intersection* of their partitions. It is implemented by Algorithm 8: the function MEET, given a sound over-approximation $D \in \mathcal{D}$ of the reachable environments and two partial decision trees $t_1, t_2 \in \mathcal{T}_{\text{NIL}}$, calls UNIFICATION (cf. Line 12) and then calls the auxiliary function JOIN-AUX. The latter collects the set $C \in \mathcal{P}(\mathcal{C})$ (initially equal to $\gamma_{\text{C}}(D)$, cf. Line 13) of linear constraints encountered up to the leaf nodes (cf. Lines 8-9), which are joined by the approximation join operator $\curlyvee_{\text{F}}$ (cf. Line 4). However, unlike Algorithm 6, MEET-AUX favors empty trees over non-empty trees (cf. Line 2).

5.2.3 Unary Operators

We now define the *unary* operators for handling skip instructions, backward variable assignments and program tests on decision trees.

Skip. The step operator $\text{STEP}_{\text{T}} \colon \mathcal{T} \to \mathcal{T}$ for handling skip instructions is implemented by Algorithm 9: the function STEP, given a decision tree $t \in \mathcal{T}$, descends along the paths of the decision tree (cf. Line 5) up to a leaf node, where the leaves step operator STEP_{F} is invoked (cf. Line 3).

The operator $\text{STEP}_{\text{F}} \colon \mathcal{F} \to \mathcal{F}$, given a function $f \in \mathcal{F} \setminus \{\bot_{\text{F}}, \top_{\text{F}}\}$, simply increments its constant to take into account that one more execution step is needed before termination; undefined leaf nodes are left unaltered:

$$\begin{aligned} \text{STEP}_{\text{F}}(\bot_{\text{F}}) &\stackrel{\text{def}}{=} \bot_{\text{F}} \\ \text{STEP}_{\text{F}}(f) &\stackrel{\text{def}}{=} \lambda X_1, \ldots, X_k.\ f(X_1, \ldots, X_k) + 1 \\ \text{STEP}_{\text{F}}(\top_{\text{F}}) &\stackrel{\text{def}}{=} \top_{\text{F}} \end{aligned} \tag{5.2.19}$$

Algorithm 8 : Tree Meet

```
1:  function MEET-AUX(t1, t2, C)                          ▷ t1, t2 ∈ T_NIL, C ∈ P(C)
2:      if ISNIL(t1) ∨ ISNIL(t2) then return NIL
3:      else if ISLEAF(t1) ∧ ISLEAF(t2) then
4:          return LEAF : t1.f ϒ_F[α_C(C)] t2.f
5:      else if ISNODE(t1) ∧ ISNODE(t2) then
6:          c ← t1.c
7:          (l1, l2) ← MEET-AUX(t1.l, t2.l, C ∪ {c})
8:          (r1, r2) ← MEET-AUX(t1.r, t2.r, C ∪ {¬c})
9:          return (NODE{c} : l1; r1, NODE{c} : l2; r2)
10:
11: function MEET(D, t1, t2)                              ▷ D ∈ D, t1, t2 ∈ T_NIL
12:     (t1, t2) ← UNIFICATION(D, t1, t2)
13:     return MEET-AUX(t1, t2, γ_C(D))
```

Algorithm 9 : Tree Step

```
1: function STEP(t)                                       ▷ t ∈ T
2:     if ISLEAF(t) then
3:         return LEAF : STEP_F(t.f)
4:     else if ISNODE(t) then
5:         return NODE{t.c} : STEP(t.l); STEP(t.r)
```

The following result proves that for a skip instruction $^l\texttt{skip}$, given a sound over-approximation $R \in \mathcal{D}$ of $\tau_{\mathrm{I}}(l)$ and a sound over-approximation $D \in \mathcal{D}$ of $\tau_{\mathrm{I}}(f[\![\,^l\texttt{skip}\,]\!])$, the step operator $\textsc{step}_{\mathrm{T}}$ is a sound over-approximation of the termination semantics $\tau_{\mathrm{Mt}}[\![\,^l\texttt{skip}\,]\!]$ defined in Equation 4.3.1:

Lemma 5.2.8 $\tau_{Mt}[\![\,^l\texttt{skip}\,]\!]\gamma_T[D]t \preccurlyeq \gamma_T[R]\textsc{step}_T(t)$.

Proof.
See Appendix A.3. ■

Tree Pruning. The remaining decision tree unary operators rely on Algorithm 10 for pruning a decision tree with respect to a given set of linear constraints. Only the subtrees whose paths from the root of the decision tree satisfy these constraints are preserved, while the other subtrees are pruned

Algorithm 10 : Tree Pruning

```
1:  function PRUNE(t, C, J)                                   ▷ t ∈ 𝒯, C, J ∈ 𝒫(𝒞)
2:      if ISEMPTY(J) then
3:          if ISLEAF(t) then return t
4:          else if ISNODE(t) then
5:              if ISREDUNDANT(t.c, C) then return PRUNE(t.l, C, J)
6:              else if ISREDUNDANT(¬j, C) then return PRUNE(t.r, C, J)
7:              else                                          ▷ t.c can be kept in t
8:                  l ← PRUNE(t.l, C ∪ {t.c}, J)
9:                  r ← PRUNE(t.r, C ∪ {¬t.c}, J)
10:                 return NODE{t.c} : l, r
11:     else if ¬ISEMPTY(J) then
12:         j ← max J          ▷ j is the largest linear constraint appearing in J
13:         if ISLEAF(t) ∨ (ISNODE(t) ∧ (t.c <_C j ∨ t.c <_C ¬j) then
14:             if ISREDUNDANT(j, C) then return PRUNE(t, C, J \ {j})
15:             else if ISREDUNDANT(¬j, C) then return NIL
16:             else if ¬j <_C j then                         ▷ j can be added to t
17:                 return NODE{j} : PRUNE(t, C ∪ {j}, J \ {j}), NIL
18:             else if j <_C ¬j then                         ▷ ¬j can be added to t
19:                 return NODE{¬j} : NIL, PRUNE(t, C ∪ {j}, J \ {j})
20:         else if ISNODE(t) ∧ j <_C t.c ∧ ¬j <_C t.c then
21:             if ISREDUNDANT(t.c, C) then return PRUNE(t.l, C, J)
22:             else if ISREDUNDANT(¬j, C) then return PRUNE(t.r, C, J)
23:             else                                          ▷ t.c can be kept in t
24:                 l ← PRUNE(t.l, C ∪ {t.c}, J)
25:                 r ← PRUNE(t.r, C ∪ {¬t.c}, J)
26:                 return NODE{t.c} : l, r
27:         else if ISNODE(t) ∧ ¬j <_C j then                 ▷ t.c and j are equal
28:             if ISREDUNDANT(j, C) then return PRUNE(t.l, C, J \ {j})
29:             else if ISREDUNDANT(¬j, C) then return NIL
30:             elsereturn NODE{j} : PRUNE(t.l, C ∪ {j}, J \ {j}), NIL
31:         else if ISNODE(t) ∧ j <_C ¬j then                 ▷ t.c and ¬j are equal
32:             if ISREDUNDANT(j, C) then return PRUNE(t.r, C, J \ {j})
33:             else if ISREDUNDANT(¬j, C) then return NIL
34:             else return NODE{¬j} : NIL, PRUNE(t.r, C ∪ {j}, J \ {j})
```

and substituted with empty trees. The function PRUNE takes as input a decision tree $t \in \mathcal{T}$, a set $C \in \mathcal{P}(\mathcal{C})$ of linear constraints representing an over-approximation of the reachable environments, and a set $J \in \mathcal{P}(\mathcal{C})$ of linear constraints that need to be added to the decision tree in order to prune it. When J is empty (cf. Line 2), PRUNE accumulates in C the linear constraints encountered along the paths (cf. Lines 8-9) up to a leaf node (cf. Line 3), possibly removing constraints that are redundant (cf. Line 5) or whose negation is redundant (cf. Line 6) with respect to C. When J is not empty (cf. Line 11), the linear constraints J are added to the decision tree in descending order with respect to $<_{\mathrm{C}}$. Note that not all constraints in J are in canonical form, that is, $J \notin \mathcal{P}(\mathcal{C}/\equiv_{\mathrm{C}})$: at each iteration a linear constraint $j \in \mathcal{C}$ is extracted from J (cf. Line 12), which is the largest constraint in J with respect to the constraints in canonical form.

Example 5.2.9

Let $J \stackrel{\text{def}}{=} \{x - 1 \geq 0, -x + 5 \geq 0\}$. Note that, $-x + 5 \geq 0$ is not in canonical form, since its negation $x - 6 \geq 0$ is larger with respect to $<_{\mathrm{C}}$. However, when comparing $x - 1 \geq 0$ and $-x + 5 \geq 0$, their canonical forms are compared and thus $\max J$ is the linear constraint $-x + 5 \geq 0$.

Then, the function PRUNE possibly adds a decision node for the linear constraint j or its negation $\neg j$ (cf. Lines 13-19), or continues the descent along the paths of the decision tree (cf. Lines 20-34). In the first case, PRUNE tests j and $\neg j$ for redundancy with respect to C (cf. Lines 14-15): when $\neg j$ is redundant with respect to C the whole decision tree is pruned (cf. Line 15); otherwise, PRUNE adds a decision node for j while pruning its right subtree (cf. Line 17), if j is already in canonical form (cf. Line 16), or it adds a decision node for for $\neg j$ while pruning its left subtree (cf. Line 19), if $\neg j$ is the canonical form of j (cf. Line 18). In the second case, PRUNE accumulates in C the encountered linear constraints (cf. Lines 24-25), possibly removing redundant decision nodes (cf. Lines 21-22) and pruning the decision tree when the encountered linear constraints coincide with those appearing in J (cf. Lines 27-30) or their negations (cf. Lines 31-34).

Example 5.2.10

Let $\mathcal{X} \stackrel{\text{def}}{=} \{x\}$, let $\mathbb{D}$ be the intervals abstract domain $\langle \mathcal{B}, \sqsubseteq_{\mathrm{B}} \rangle$, let $\mathbb{C}$ be the interval constraints auxiliary abstract domain $\langle \mathcal{C}_{\mathrm{B}}/\equiv_{\mathrm{C}}, <_{\mathrm{C}} \rangle$, and let $\mathbb{F}$ be the affine functions auxiliary abstract domain $\langle \mathcal{F}_{\mathrm{A}}, \preccurlyeq_{\mathrm{F}} \rangle$. We consider the decision tree represented in Figure 5.7 from Example 5.2.7, and the set of constraints $J \stackrel{\text{def}}{=} \{-x + 1 \geq 0\}$. Let $d \stackrel{\text{def}}{=} \top_{\mathrm{B}}$ and let $C \stackrel{\text{def}}{=} \gamma_{\mathrm{C}}(d) = \emptyset$.

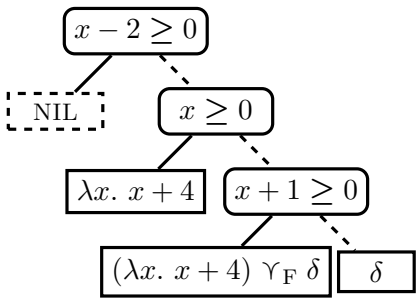


Figure 5.8: Tree pruning of Figure 5.7 with respect to $\{-x + 2 \geq 0\}$.

The function PRUNE adds a decision node for the negation $x - 2 \geq 0$ of $-x + 1 \geq 0$, since $-x + 1 \geq 0$ is not in canonical form (cf. Line 18), and prunes the left subtree (cf. Line 19). Then, it continues the descent along the paths of the decision tree, without removing decision nodes. The result is the decision tree represented in Figure 5.8.

Assignments. A variable assignment might impact some linear constraints within the decision nodes as well as some functions within the leaf nodes.

We define the operator B-ASSIGN$_{\mathrm{C}}$ to handle backward variable assignments by manipulating linear constraints within the decision nodes by means of the underlying numerical abstract domain $\langle \mathcal{D}, \sqsubseteq_{\mathrm{D}} \rangle$:

$$\text{B-ASSIGN}_{\mathrm{C}}[\![\, X := \mathit{aexp} \,]\!] D \stackrel{\text{def}}{=} \lambda c.\ \alpha_{\mathrm{C}}(\text{B-ASSIGN}_{\mathrm{D}}[\![\, X := \mathit{aexp} \,]\!] D(\gamma_{\mathrm{C}}(\{c\}))) \tag{5.2.20}$$

The operator B-ASSIGN$_{\mathrm{C}}[\![\, X := \mathit{aexp} \,]\!] \colon \mathcal{D} \to \mathcal{C} \to \mathcal{P}(\mathcal{C})$ takes as input a numerical abstraction $D \in \mathcal{D}$ of the reachable environments at the initial control point of the instruction and a linear constraint $c \in \mathcal{C}$, and produces a set $C \in \mathcal{P}(\mathcal{C})$ of linear constraints that need to be substituted to c in the decision tree. It is often the case that the set C contains a single linear constraint. However, when variable assignments and program tests cannot be exactly represented in the numerical abstract domain, the output set C contains multiple linear constraints, as shown by the following example.

Example 5.2.11

Let $\mathcal{X} \stackrel{\text{def}}{=} \{x, y\}$. We consider, as numerical abstract domain $\mathbb{D}$, the intervals abstract domain $\langle \mathcal{B}, \sqsubseteq_{\mathrm{B}} \rangle$. Let $c \in \mathcal{C}_{\mathrm{B}}$ be the linear constraint $x \geq 0$ and let

$D \in \mathcal{B}$ be the interval $\langle y, [0, +\infty] \rangle$, which is isomorphic to the set of constraints $\{y \geq 0\}$. The backward assignment $x := x - y$ produces the set of linear constraints $C \stackrel{\text{def}}{=} \text{B-ASSIGN}_{\text{C}}[\![x := x - y]\!] D(c) = \{x \geq 0, y \geq 0\}$, since the linear constraint $x - y \geq 0$, obtained by replacing x with $x - y$ in c, cannot be exactly represented in the intervals abstract domain.

We now define the operator $\text{B-ASSIGN}_{\text{F}}$ to handle backward variable assignments within the leaf nodes of a decision tree. The operator $\text{B-ASSIGN}_{\text{F}}[\![X := \mathit{aexp}]\!] \colon \mathcal{D} \to \mathcal{D} \to \mathcal{F} \to \mathcal{F}$, given a numerical abstraction $d \in \mathcal{D}$ of the reachable environments at the initial control point of the instructions, a numerical abstraction $D \in \mathcal{D}$ of the linear constraints accumulated along the path to the leaf node, and a function $f \in \mathcal{F} \setminus \{\bot_{\text{F}}, \top_{\text{F}}\}$, substitutes the arithmetic expression *aexp* to the variable X with the function, and increments the constant of the resulting function to take into account that one more program execution step is needed before termination; when such function does not exist, the result is $\top_{\text{F}}$; undefined leaf nodes are left unaltered:

$$\begin{aligned}
\text{B-ASSIGN}_{\text{F}}[\![X := \mathit{aexp}]\!] d[D] \bot_{\text{F}} &\stackrel{\text{def}}{=} \bot_{\text{F}} \\
\text{B-ASSIGN}_{\text{F}}[\![X := \mathit{aexp}]\!] d[D] f &\stackrel{\text{def}}{=} \begin{cases} F & F \in \mathcal{F} \setminus \{\bot_{\text{F}}, \top_{\text{F}}\} \\ \top_{\text{F}} & \text{otherwise} \end{cases} \\
\text{B-ASSIGN}_{\text{F}}[\![X := \mathit{aexp}]\!] d[D] \top_{\text{F}} &\stackrel{\text{def}}{=} \top_{\text{F}}
\end{aligned} \tag{5.2.21}$$

where $F(X_1, \ldots, X, \ldots, X_k) \stackrel{\text{def}}{=} \max\{f(\rho(X_1), \ldots, v, \ldots, \rho(X_k)) + 1 \mid \rho \in \gamma_{\text{D}}(R), v \in [\![\mathit{aexp}]\!]\rho\}$ and $R \stackrel{\text{def}}{=} \text{B-ASSIGN}_{\text{D}}[\![X := \mathit{aexp}]\!] d(D)$. Note that, all possible outcomes of the backward assignment are taken into account when substituting the arithmetic expression *aexp* to the variable X.

As an instance, given an affine function $f \in \mathcal{F}_{\text{A}} \setminus \{\bot_{\text{F}}, \top_{\text{F}}\}$, the result of the backward assignment is the affine function $f' \in \mathcal{F}_{\text{A}} \setminus \{\bot_{\text{F}}, \top_{\text{F}}\}$ whose hypograph $f'[R]{\downarrow}$ within R is the *convex-hull* of the hypograph $F[R]{\downarrow}$ of F; when such affine function does not exist, the result is $\top_{\text{F}}$:

$$\text{B-ASSIGN}_{\text{F}}[\![X := \mathit{aexp}]\!] d[D] f \stackrel{\text{def}}{=} \begin{cases} f' & f'[R]{\downarrow} = \mathit{convex\text{-}hull}\{F[R]{\downarrow}\} \\ \top_{\text{F}} & \text{otherwise} \end{cases} \tag{5.2.22}$$

To clarify, consider the following example:

Algorithm 11 : Tree Assignment

```
1:  function ASSIGN-AUX⟦ X := aexp ⟧(D, t, C)        ▷ D ∈ 𝒟, t ∈ 𝒯, C ∈ 𝒫(𝒞)
2:      if ISLEAF(t) then
3:          return LEAF : B-ASSIGN_F⟦ X := aexp ⟧D[α_D(C)]t.f
4:      else if ISNODE(t) then
5:          I ← B-ASSIGN_C⟦ X := aexp ⟧d(t.c)
6:          J ← B-ASSIGN_C⟦ X := aexp ⟧d(¬t.c)
7:          if ISEMPTY(I) ∧ ISEMPTY(J) then
8:              t_1 ← ASSIGN-AUX⟦ X := aexp ⟧(d, t.l, C)
9:              t_2 ← ASSIGN-AUX⟦ X := aexp ⟧(d, t.r, C)
10:             return A-JOIN(α_C(C), t_1, t_2)
11:         else if ISEMPTY(I) ∧ ⊥_C ∈ J then
12:             return ASSIGN-AUX⟦ X := aexp ⟧(d, t.l, C)
13:         else if ⊥_C ∈ I ∧ ISEMPTY(J) then
14:             return ASSIGN-AUX⟦ X := aexp ⟧(d, t.r, C)
15:         else if (¬ISEMPTY(I) ∧ ⊥_C ∉ I ∧ ¬ISEMPTY(J) ∧ ⊥_C ∉ J then
16:             l ← ASSIGN-AUX⟦ X := aexp ⟧(d, t.l, C ∪ {t.c})
17:             t_1 ← PRUNE(l, γ_C(d), I)
18:             r ← ASSIGN-AUX⟦ X := aexp ⟧(d, t.r, C ∪ {¬t.c})
19:             t_2 ← PRUNE(r, γ_C(d), J)
20:             return A-JOIN(α_C(C), t_1, t_2)
21:         else return FAIL("invalid argument")
22:
23: function ASSIGN⟦ X := aexp ⟧(D, t)                ▷ D ∈ 𝒟, t ∈ 𝒯
24:     return ASSIGN-AUX⟦ X := aexp ⟧(D, t, γ_C(D))
```

Example 5.2.12

Let $\mathcal{X} \stackrel{\text{def}}{=} \{x\}$ and let $\mathbb{D}$ be the intervals abstract domain $\langle \mathcal{B}, \sqsubseteq_{\mathrm{B}} \rangle$. We consider the affine function $f \in \mathcal{F}_{\mathrm{A}} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}$ defined as $f(x) \stackrel{\text{def}}{=} x + 1$ and the backward assignment $x := x + [1, 2]$. Let $D \in \mathcal{B}$ be the interval $\langle x, [1, 2] \rangle$ and let $d \in \mathcal{B}$ be defined as $d \stackrel{\text{def}}{=} \top_{\mathrm{B}}$. The backward assignment produces the interval $R \stackrel{\text{def}}{=} \langle x, [-1, 1] \rangle$ and the affine function $f' \in \mathcal{F}_{\mathrm{A}} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}$ defined as $f'(x) \stackrel{\text{def}}{=} x + 4$, since substituting the expression $x + [1, 2]$ to the variable x within f gives $f(x + [1, 2]) + 1 = x + [1, 2] + 1 + 1 = x + [3, 4]$ and $\max\{3, 4\} = 4$.

The operator $\text{B-ASSIGN}_{\mathrm{T}}[\![X := aexp]\!] \colon \mathcal{D} \to \mathcal{T} \to \mathcal{T}$ for handling backward assignments within decision trees is implemented by Algorithm 11: the

function ASSIGN$[\![X := aexp]\!]$, given a sound over-approximation $D \in \mathcal{D}$ of the reachable environments before the assignment and a decision tree $t \in \mathcal{T}$, calls the auxiliary function ASSIGN-AUX$[\![X := aexp]\!]$, which performs the assignment on each linear constraint along the paths in the decision tree (cf. Lines 4-21), and accumulates the encountered constraints into a set $C \in \mathcal{P}(\mathcal{C})$ (initially equal to $\gamma_{\mathrm{C}}(D)$, cf. Line 24), up to the leaf nodes, where the assignment is performed by the operator B-ASSIGN$_{\mathrm{F}}$ defined in Equation 5.2.21 (cf. Line 3).

In particular, for each linear constraint $c \in \mathcal{C}$ appearing in the decision trees, the auxiliary function ASSIGN-AUX$[\![X := aexp]\!]$ performs the assignment on both c and its negation $\neg c$ (cf. Lines 5-6) by means of the backward assignment operator B-ASSIGN$_{\mathrm{C}}$ defined in Equation 5.2.20. Then, ASSIGN-AUX$[\![X := aexp]\!]$ possibly (cf. Line 15) calls PRUNE to add the resulting sets of constraints $I \in \mathcal{P}(\mathcal{C})$ and $J \in \mathcal{P}(\mathcal{C})$ to the decision tree (cf. Lines 17-19). In case both I and J are empty (cf. Line 7), it means that neither c nor $\neg c$ exist anymore and thus the subtrees of the decision tree are joined by the approximation join A-JOIN (cf. Line 10). Note that, the function PRUNE introduces empty trees. However, they disappear when the subtrees are joined. Indeed, since I and J are sets of constraints resulting from the assignment on *complementary* linear constraints, they identify adjacent (or, due to non-determinism, overlapping) partitions. In case I is empty and J is an unsatisfiable set of constraints (cf. Line 11), it means that $\neg c$ is no longer satisfiable and thus only the left subtree of the decision tree is kept (cf. Line 12). Similarly, in case I is an unsatisfiable set of constraints and J is empty (cf. Line 13), it means that c is no longer satisfiable and thus only the right subtree of the decision tree is kept (cf. Line 14). When both I and J are unsatisfiable sets of constraints, an error is raised (cf. Line 21).

Example 5.2.13

Let $\mathcal{X} \stackrel{\text{def}}{=} \{x\}$, let $\mathbb{D}$ be the intervals abstract domain $\langle \mathcal{B}, \sqsubseteq_{\mathrm{B}} \rangle$, let $\mathbb{C}$ be the interval constraints auxiliary abstract domain $\langle \mathcal{C}_{\mathrm{B}}/\equiv_{\mathrm{C}}, <_{\mathrm{C}} \rangle$, and let $\mathbb{F}$ be the affine functions auxiliary abstract domain $\langle \mathcal{F}_{\mathrm{A}}, \preccurlyeq_{\mathrm{F}} \rangle$. We consider the decision tree represented in Figure 5.9a, where $\alpha, \beta \in \mathcal{T}$ are leaf nodes, and the backward assignment $x := x + [1, 2]$. Let $d \stackrel{\text{def}}{=} \top_{\mathrm{B}}$.

The function ASSIGN-AUX collects the set $C \stackrel{\text{def}}{=} \{-x + 2 \geq 0, x - 1 \geq 0\}$ of encountered linear constraints up to the leaf node LEAF : $\lambda x.x + 1$, where B-ASSIGN$_{\mathrm{F}}$ performs the assignment (cf. Line 3). From Example 5.2.12, the result of the assignment is the leaf node LEAF : $\lambda x.x + 4$. Similarly, let γ and δ be the results of the assignment to α and γ, respectively.

Then, the function ASSIGN-AUX performs the assignment on $x - 1 \geq 0$ and its negation $-x \geq 0$ (cf. Lines 5-6) yielding the sets of constraints

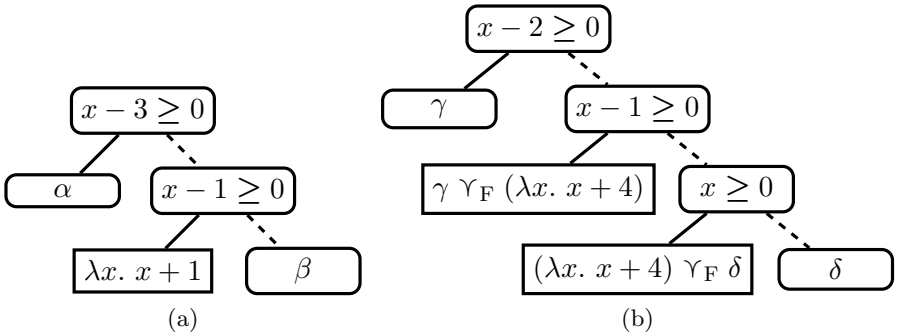


Figure 5.9: Tree assignment on the decision tree (a) of Example 5.2.13. The resulting decision tree is represented in (b).

$I \stackrel{\text{def}}{=} \{x + 1 \geq 0\}$ and $J \stackrel{\text{def}}{=} \{-x - 1 \geq 0\}$, respectively. Note that, because of non-determinism within the variable assignment, the partitions identified by I and J overlap when $x = -1$. The leaf node LEAF : $\lambda x.x + 4$ is pruned with I (cf. Line 17) and the leaf node δ is pruned with J (cf. Line 19); the resulting decision trees respectively represented in Figure 5.4a and Figure 5.4b are joined by the approximation join (cf. Line 20) yielding the decision tree represented in Figure 5.7 (cf. Example 5.2.7).

Finally, ASSIGN-AUX performs the assignment on $x-3 \geq 0$ and its negation $-x+2 \geq 0$ yielding the sets of constraints $I \stackrel{\text{def}}{=} \{x-1 \geq 0\}$ and $J \stackrel{\text{def}}{=} \{-x+1 \geq 0\}$, respectively. Note that, because of non-determinism, the partitions identified by I and J overlap when $x = 1$. The leaf node γ is pruned with I and the decision tree represented in Figure 5.7 is pruned with J; the resulting decision trees $\mathrm{NODE}\{x - 1 \geq 0\} : \gamma; \mathrm{NIL}$ and Figure 5.8 (cf. Example 5.2.10) are joined by the approximation join yielding the decision tree represented in Figure 5.9.

In absence of run-time errors, the following result proves that for a variable assignment ${}^{l}X := \mathit{aexp}$, given a sound over-approximation $R \in \mathcal{D}$ of $\tau_{\mathrm{I}}(l)$ and a sound over-approximation $D \in \mathcal{D}$ of $\tau_{\mathrm{I}}(f[\![{}^{l}X := \mathit{aexp}]\!])$, the backward assignment operator $\text{B-ASSIGN}_{\mathrm{T}}[\![X := \mathit{aexp}]\!]$ is a sound over-approximation of $\tau_{\mathrm{Mt}}[\![{}^{l}X := \mathit{aexp}]\!]$ defined in Equation 4.3.2:

Lemma 5.2.14 $\tau_{Mt}[\![{}^{l}X := \mathit{aexp}]\!]\gamma_D[D]t \preccurlyeq \gamma_T[R]((\text{B-ASSIGN}_T[\![X := \mathit{aexp}]\!]R)(t))$.

Proof.
See Appendix A.3. ■

Algorithm 12 : Tree Filter

```
1:  function FILTER⟦ bexp ⟧(D, t)                                  ▷ D ∈ 𝒟, t ∈ 𝒯
2:      switch bexp do
3:          case ? : return t
4:          case not bexp :
5:              return FILTER⟦ ¬bexp ⟧(D, t)
6:          case bexp_1 and bexp_2 :
7:              return MEET(D, FILTER⟦ bexp_1 ⟧(D, t), FILTER⟦ bexp_2 ⟧(D, t))
8:          case bexp_1 or bexp_2 :
9:              return A-JOIN(D, FILTER⟦ bexp_1 ⟧(D, t), FILTER⟦ bexp_2 ⟧(D, t))
10:         case aexp_1 ⋈ aexp_2 :
11:             J ← FILTER_C⟦ aexp_1 ⋈ aexp_2 ⟧D
12:             return PRUNE(STEP(t), γ_C(D), J)
```

Tests. In case of a program test, all paths that are feasible in the decision tree are preserved, while all paths that for sure can never be followed according to the tested condition are discarded.

We define the operator $\text{FILTER}_\text{C}[\![\, bexp \,]\!] \colon \mathcal{D} \to \mathcal{P}(\mathcal{C})$ which takes as input a numerical abstraction $D \in \mathcal{D}$ and, by means of the underlying numerical abstract domain $\langle \mathcal{D}, \sqsubseteq_\text{D} \rangle$, produces a set $C \in \mathcal{P}(\mathcal{C})$ of linear constraints that need to be added to prune the decision tree:

$$\text{FILTER}_\text{C}[\![\, bexp \,]\!]D \stackrel{\text{def}}{=} \alpha_\text{C}(\text{FILTER}_\text{D}[\![\, bexp \,]\!]D) \tag{5.2.23}$$

The operator $\text{FILTER}_\text{T}[\![\, bexp \,]\!] \colon \mathcal{D} \to \mathcal{T} \to \mathcal{T}$ for handling tests within decision trees is implemented by Algorithm 12: the function $\text{FILTER}[\![\, bexp \,]\!]$, given a sound over-approximation $D \in \mathcal{D}$ of the reachable environments before the test and a decision tree $t \in \mathcal{T}$, reasons by induction on the structure of the boolean expression *bexp*. In particular, when *bexp* is a conjunction of two boolean expression $bexp_1$ and $bexp_2$ (cf. Line 6), the resulting decision trees are merged by the function MEET defined in Algorithm 8 (cf. Line 7). Similarly, when *bexp* is a disjunction of two boolean expression $bexp_1$ and $bexp_2$ (cf. Line 7), the resulting decision trees are merged by the approximation join A-JOIN defined in Algorithm 6 (cf. Line 9). Instead, when *bexp* is a comparison of arithmetic expressions $aexp_1 \bowtie aexp_2$ (cf. Line 10), the function STEP (cf. Algorithm 9) is invoked (cf. Line 12). Then, the resulting decision tree is pruned (cf. Line 12) with the set of constraints J (cf. Line 11) produced by the operator FILTER_C defined in Equation 5.2.23.

In absence of run-time errors, the following result provides a sound over-approximation of $\tau_{\mathrm{Mt}}[\![\,\texttt{if}\ ^{l}\mathit{bexp}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \texttt{fi}\,]\!]$ defined in Equation 4.3.3, given a sound over-approximation $R \in \mathcal{D}$ of $\tau_{\mathrm{I}}(l)$ and a sound over-approximation $D \in \mathcal{D}$ of $\tau_{\mathrm{I}}(f[\![\,\texttt{if}\ ^{l}\mathit{bexp}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \texttt{fi}\,]\!])$:

Lemma 5.2.15 *Let* $F_1^\natural[t] \stackrel{def}{=} (\textsc{filter}_T[\![\,\mathit{bexp}\,]\!]R)(\tau_{Mt}^\natural[\![\,\mathit{stmt}_1\,]\!]t)$ *and* $F_2^\natural[t] \stackrel{def}{=} (\textsc{filter}_T[\![\,\texttt{not}\ \mathit{bexp}\,]\!]R)(\tau_{Mt}^\natural[\![\,\mathit{stmt}_2\,]\!]t)$*. Then, for all* $t \in \mathcal{T}$*, we have:*

$$\tau_{Mt}[\![\,\texttt{if}\ ^{l}\mathit{bexp}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \texttt{fi}\,]\!]\gamma_T[D]t \preccurlyeq \gamma_T[R](F_1^\natural[t] \curlyvee_T[R] F_2^\natural[t])$$

Proof.
See Appendix A.3. ■

Note that, $\textsc{filter}_{\mathrm{T}}$ introduces empty trees. However, they disappear when the decision trees $F_1^\natural[t]$ and $F_2^\natural[t]$ are joined. Indeed, $F_1^\natural[t]$ and $F_2^\natural[t]$ are obtained from *complementary* boolean expressions.

Similarly, the next result provides, for a loop $\texttt{while}\ ^{l}\mathit{bexp}\ \texttt{do}\ \mathit{stmt}\ \texttt{od}$, a sound over-approximation $\phi_{\mathrm{Mt}}^\natural$ of ϕ_{Mt} defined in Equation 4.3.5, given sound over-approximations $R \in \mathcal{D}$ of $\tau_{\mathrm{I}}(l)$ and $D \in \mathcal{D}$ of $\tau_{\mathrm{I}}(f[\![\,\texttt{while}\ ^{l}\mathit{bexp}\ \texttt{do}\ \mathit{stmt}\ \texttt{od}\,]\!])$:

Lemma 5.2.16 *Let* $F_1^\natural[x] \stackrel{def}{=} (\textsc{filter}_T[\![\,\mathit{bexp}\,]\!]R)(\tau_{Mt}^\natural[\![\,\mathit{stmt}\,]\!]x)$ *and* $F_2^\natural[t] \stackrel{def}{=} (\textsc{filter}_T[\![\,\texttt{not}\ \mathit{bexp}\,]\!]R)(t)$*. Then, given* $t \in \mathcal{T}$*, for all* $x \in \mathcal{T}$ *we have:*

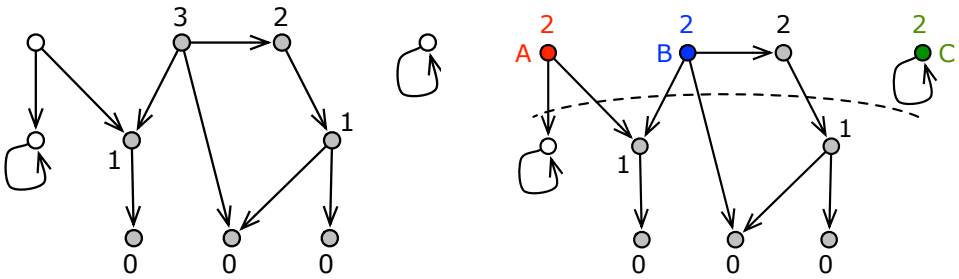
$$\phi_{Mt}(\gamma_D[R]x) \preccurlyeq \gamma_T[R](\phi_{Mt}^\natural(x))$$

where $\phi_{Mt}^\natural(x) \stackrel{def}{=} F_1^\natural[x] \curlyvee_T[R] F_2^\natural[t]$*.*

Proof.
See Appendix A.3. ■

5.2.4 Widening

The widening operator ∇_{T} requires a more thorough discussion. The widening is allowed more freedom than the other operators, in the sense that it is temporary allowed to *under-approximate* the value of the termination semantics τ_{Mt} (cf. Section 4.3) or *over-approximate* its domain of definition, or both — in contrast with the approximation order $\preccurlyeq$ (cf. Equation 4.2.12). This is necessary in order to extrapolate a ranking function over the environments



(a) Most precise ranking function. (b) Unsound abstraction.

Figure 5.10: Unsound abstraction (b) of a most precise ranking function (a).

on which it is not yet defined. The only requirement is that, when the iteration sequence with widening is stable for the computational order, its limit is a sound abstraction of the termination semantics with respect to the approximation order. In the following, we discuss in detail how the widening guarantees the soundness of the analysis.

As running example, let us consider Figure 5.10. In Figure 5.10a we depict a transition system and the value of the termination semantics for the well-founded part of its transition relation. In Figure 5.10b we represent the concretization of a possible iterate of the analysis: we assume that the first iterate has individuated the states marked with value zero, the second iterate has individuated the states marked with value one, and the widening at the third iterate has extrapolated the ranking function over the states marked with value two. In this case the abstraction both under-approximates the value of the termination semantics (on the second state from the left — case B) and over-approximates its domain of definition (including the first and the last state from the left — case A and C, respectively). In case A, the non-terminating loop is *outside* the domain of definition of the unsound abstract function, while in case C the loop is *inside*. The analysis continues iterating until all these discrepancies are solved and, in the following, we explain and justify why this works in general.

For a loop `while` ^{l}*bexp* `do` *stmt* `od`, given a sound over-approximation $R \in \mathcal{D}$ of $\tau_{\mathrm{I}}(l)$, we define the iteration sequence with widening as follows:

$$
\begin{aligned}
y_0 &\stackrel{\text{def}}{=} \bot_{\mathrm{T}} \\
y_{n+1} &\stackrel{\text{def}}{=} \begin{cases} y_n & \phi^{\natural}_{\mathrm{Mt}}(y_n) \sqsubseteq_{\mathrm{T}} [R]\, y_n \wedge \phi^{\natural}_{\mathrm{Mt}}(y_n) \preccurlyeq_{\mathrm{T}} [R]\, y_n \\ y_n \triangledown_{\mathrm{T}} \phi^{\natural}_{\mathrm{Mt}}(y_n) & \text{otherwise} \end{cases}
\end{aligned}
\tag{5.2.24}
$$

Algorithm 13 : Tree Widening

1: **function** WIDEN(D,t_1,t_2) ▷ $D \in \mathcal{D}$, $t_1, t_2 \in \mathcal{T}$
2: $t_2 \leftarrow$ CASEA(D, t_1, t_2) ▷ check for case A
3: $(t_1, t_2) \leftarrow$ LEFT-UNIFICATION$(\sqcup_{\mathrm{F}}, D, t_1, t_2)$ ▷ domain widening
4: $t_2 \leftarrow$ CASEBORC(D, t_1, t_2) ▷ check for case B or case C
5: **return** WIDEN-AUX$(t_1, t_1, t_2, \gamma_{\mathrm{C}}(D))$ ▷ value widening

Algorithm 14 : Check Case A

function CASEA-AUX(t_1, t_2, C) ▷ $t_1, t_2 \in \mathcal{T}$, $C \in \mathcal{P}(\mathcal{C})$
  if ISLEAF$(t_1) \wedge$ ISLEAF(t_2) **then**
    if $t_1.f \sqsubseteq_{\mathrm{F}} [\alpha_{\mathrm{C}}(C)]\ t_2.f$ **then return** $t_2.f$
    else return LEAF : $\top_{\mathrm{F}}$
  else if ISNODE$(t_1) \wedge$ ISNODE(t_2) **then**
    $l \leftarrow$ CASEA-AUX$(t_1.l, t_2.l, C \cup \{t_2.c\})$
    $r \leftarrow$ CASEA-AUX$(t_1.r, t_2.r, C \cup \{\neg t_2.c\})$
    return NODE$\{t_2.c\} : l; r$

function CASEA(D, t_1, t_2) ▷ $D \in \mathcal{D}$, $t_1, t_2 \in \mathcal{T}$
  $(t_1, t_2) \leftarrow$ UNIFICATION(D, t_1, t_2)
  return CASEA-AUX$(t_1, t_2, \gamma_{\mathrm{C}}(D))$

In the following, its limit is denoted by $\mathrm{lfp}^{\natural}\ \phi^{\natural}_{\mathrm{Mt}}$. Note that, the usual condition for halting the iterations $\phi^{\natural}_{\mathrm{Mt}}(y_n) \sqsubseteq_{\mathrm{T}} [R]\ y_n$ has been strengthened to $\phi^{\natural}_{\mathrm{Mt}}(y_n) \sqsubseteq_{\mathrm{T}} [R]\ y_n \wedge \phi^{\natural}_{\mathrm{Mt}}(y_n) \preccurlyeq_{\mathrm{T}} [R]\ y_n$ in order to force the iterations to continue in case of discrepancies as in Figure 5.10b. Indeed, in situations like case A and case B, the usual halting condition $\phi^{\natural}_{\mathrm{Mt}}(y_n) \sqsubseteq_{\mathrm{T}} [R]\ y_n$ can be satisfied but the iteration sequence should not halt.

The widening $\triangledown_{\mathrm{T}}$ is implemented by Algorithm 13: the function WIDEN, given a given a sound over-approximation $D \in \mathcal{D}$ of the reachable environments and two decision trees $t_1, t_2 \in \mathcal{T}$, calls in order the functions CASEA (cf. Line 2), LEFT-UNIFICATION (cf. Line 3), CASEBORC (cf. Line 4), and WIDEN-AUX (cf. Line 4). In the following, we detail all these functions.

Check for Case A. First, the widening $\triangledown_{\mathrm{T}}$ has to detect situations like case A in Figure 5.10b, where at some iterate y_i in Equation 5.2.24 the domain of the termination semantics has been over-approximated including environments from which a non-terminating loop is reachable.

The following result proves in situations like case A that, given $t \in \mathcal{T}$,

a sound over-approximations $R \in \mathcal{D}$ of $\tau_{\mathrm{I}}(l)$ and a sound-overapproximation $D \in \mathcal{D}$ of $\tau_{\mathrm{I}}(f[\![\texttt{while}\ ^{l} bexp\ \texttt{do}\ stmt\ \texttt{od}]\!])$, an additional iterate of $\phi^{\natural}_{\mathrm{Mt}}$ removes (a subset of) the incriminated environments from the abstraction:

Lemma 5.2.17 *Let* $w \stackrel{def}{=} \tau_{Mt}[\![\texttt{while}\ ^{l} bexp\ \texttt{do}\ stmt\ \texttt{od}]\!]\gamma_T[D]t$ *and, for some iterate* y_i*, let* $\mathrm{dom}(\gamma_T[R]y_i) \setminus \mathrm{dom}(w) \neq \emptyset$*. Then, in case A of Figure 5.10b, we have* $\mathrm{dom}(\gamma_T[R]\phi^{\natural}_{Mt}(y_i)) \setminus \mathrm{dom}(w) \subset \mathrm{dom}(\gamma_T[R]y_i) \setminus \mathrm{dom}(w)$*.*

Proof.
Let $\mathrm{dom}(\gamma_{\mathrm{T}}[R]y_i) \setminus \mathrm{dom}(w) \neq \emptyset$, for some iterate y_i. It means that there exists at least an environment $\rho \in \mathrm{dom}(\gamma_{\mathrm{T}}[R]y_i)$ such that the state $\langle l, \rho\rangle \in \Sigma$ belongs to a non-terminating program execution trace $\sigma \in \Sigma^{\omega}$. We assumed to be in case A of Figure 5.10b. Thus, let $\langle l', \rho'\rangle \in \Sigma$ where $\rho' \notin \mathrm{dom}(\gamma_{\mathrm{T}}[R]y_i)$ be a state reachable from $\langle l, \rho\rangle$ on σ. Without loss of generality, we can assume that $\langle l', \rho'\rangle$ is the immediate successor of $\langle l, \rho\rangle$: $\langle\langle l, \rho\rangle, \langle l', \rho'\rangle\rangle \in \tau$. Thus, by definition of ϕ (cf. Equation 4.3.5), we have $\rho \notin \mathrm{dom}(\phi(\gamma_{\mathrm{T}}[R]y_i))$ and, by Lemma 5.2.16 and by definition of the approximaiton order $\preccurlyeq$ (cf. Equation 4.2.12), we have $\mathrm{dom}(\gamma_{\mathrm{T}}[R]\phi^{\natural}_{\mathrm{Mt}}(y_i)) \subseteq \mathrm{dom}(\phi(\gamma_{\mathrm{T}}[R]y_i))$ which implies $\rho \notin \mathrm{dom}(\gamma_{\mathrm{T}}[R]\phi^{\natural}_{\mathrm{Mt}}(y_i))$. This concludes the proof, for case A, that $\mathrm{dom}(\gamma_{\mathrm{T}}[R]\phi^{\natural}_{\mathrm{Mt}}(y_i)) \setminus \mathrm{dom}(w) \subset \mathrm{dom}(\gamma_{\mathrm{T}}[R]y_i) \setminus \mathrm{dom}(w)$. ■

Therefore note that, in situations like case A, the iterate $\phi^{\natural}_{\mathrm{Mt}}(y_i)$ is a decision tree with more undefined leaf nodes than y_i, that is $y_i \preccurlyeq_{\mathrm{T}} [R]\ \phi^{\natural}_{\mathrm{Mt}}(y_i)$. Moreover, when the newly added undefined leaf nodes are $\bot_{\mathrm{F}}$-leaves, we have $\phi^{\natural}_{\mathrm{Mt}}(y_i) \sqsubseteq_{\mathrm{T}} [R]\ y_i$. Thus, the widening $\triangledown_{\mathrm{T}}$ has to detect whether some defined leaf node in y_i has become a $\bot_{\mathrm{F}}$-leaf in $\phi^{\natural}_{\mathrm{Mt}}(y_i)$ and, in case, turn it into a $\top_{\mathrm{F}}$-leaf in order to have $y_i \sqsubseteq_{\mathrm{T}} [R]\ \phi^{\natural}_{\mathrm{Mt}}(y_i)$ but also to prevent successive iterates from mistakenly including again the same environment in their domain. This check is implemented by Algorithm 14: the main function CASEA, given a sound over-approximation $D \in \mathcal{D}$ of the reachable environments and two decision trees $t_1, t_2 \in \mathcal{T}$, calls UNIFICATION for tree unification (cf. Line 11) and then calls the auxiliary function CASEA-AUX. The latter collects the set $C \in \mathcal{P}(\mathcal{C})$ (initially equal to $\gamma_{\mathrm{C}}(D)$, cf. Line 12) of the linear constraints encountered up to the leaf nodes (cf. Lines 6-7), which are compared by the computational order $\sqsubseteq_{\mathrm{F}} [\alpha_{\mathrm{C}}(C)]$ defined in Equation 5.2.8 and Figure 5.3b (cf. Line 3) and, in case, turned into a $\top_{\mathrm{F}}$-leaf (cf. Line 4).

Domain Widening - Left Unification. In order to ensure convergence, the widening $\triangledown_{\mathrm{T}}$ uses Algorithm 15 to limit the size of the decision trees,

Algorithm 15 : Tree Left Unification

```
1: function LEFT-UNIFICATION-AUX(⊕, t1, t2, C)
2:                                  ▷ t1, t2 ∈ 𝒯, C ∈ 𝒫(𝒞), ⊕ ∈ {ϒF, ⊔F}
3:     if isLeaf(t1) ∧ isLeaf(t2) then return (t1, t2)
4:     else if isLeaf(t1) ∨ (isNode(t1) ∧ isNode(t2) ∧ t1.c <C t2.c) then
5:         if isRedundant(t2.c, C) then
6:             return LEFT-UNIFICATION-AUX(t1, t2.l, C)
7:         else if isRedundant(¬t2.c, C) then
8:             return LEFT-UNIFICATION-AUX(t1, t2.r, C)
9:         else                         ▷ t2.c should be removed from t2
10:            return LEFT-UNIFICATION-AUX(t1, t2, JOIN(⊕, αC(C), t2.l, t2.r))
11:    else if isLeaf(t2) ∨ (isNode(t1) ∧ isNode(t2) ∧ t2.c <C t1.c) then
12:        if isRedundant(t1.c, C) then
13:            return LEFT-UNIFICATION-AUX(t1.l, t2, C)
14:        else if isRedundant(¬t1.c, C) then
15:            return LEFT-UNIFICATION-AUX(t1.r, t2, C)
16:        else                         ▷ t1.c can be kept in t1 and added to t2
17:            (l1, l2) ← LEFT-UNIFICATION-AUX(t1.l, t2, C ∪ {t1.c})
18:            (r1, r2) ← LEFT-UNIFICATION-AUX(t1.r, t2, C ∪ {¬t1.c})
19:            return (NODE{t1.c} : l1; r1, NODE{t1.c} : l2; r2)
20:    else if isNode(t1) ∧ isNode(t2) then
21:        c ← t1.c                     ▷ t1.c and t2.c are equal
22:        if isRedundant(c, C) then
23:            return LEFT-UNIFICATION-AUX(t1.l, t2.l, C)
24:        else if isRedundant(¬c, C) then
25:            return LEFT-UNIFICATION-AUX(t1.r, t2.r, C)
26:        else                         ▷ c can be kept in t1 and t2
27:            (l1, l2) ← LEFT-UNIFICATION-AUX(t1.l, t2.l, C ∪ {c})
28:            (r1, r2) ← LEFT-UNIFICATION-AUX(t1.r, t2.r, C ∪ {¬c})
29:            return (NODE{c} : l1; r1, NODE{c} : l2; r2)
30:
31: function LEFT-UNIFICATION(⊕, D, t1, t2)
32:                                  ▷ D ∈ 𝒟, t1, t2 ∈ 𝒯, ⊕ ∈ {ϒF, ⊔F}
33:    return LEFT-UNIFICATION-AUX(t1, t2, γC(D))
```

and thus avoid infinite sequences of partition refinements. Algorithm 15 is a slight modification of Algorithm 1: the main function LEFT-UNIFICAITON is parameterized by the choice of the join $\oplus$ between leaf nodes and, given a sound over-approximation $D \in \mathcal{D}$ of the reachable environments and two decision trees $t_1, t_2 \in \mathcal{T}$, calls the auxiliary function LEFT-UNIFICATION-AUX in order to force the structure of t_1 on t_2. Note that in this way, unlike Algorithm 1, Algorithm 15 might loose information.

The function LEFT-UNIFICATION-AUX accumulates into a set $C \in \mathcal{P}(\mathcal{C})$ (initially equal to $\gamma_{\mathrm{C}}(D)$, cf. Line 33) the linear constraints encountered along the paths in the first decision trees (cf. Lines 17-18, Lines 27-28), possibly adding decision nodes to the second tree (cf. Line 19) or removing decision nodes from the second tree (cf. Line 10), or removing constraints that are redundant (cf. Line 5, Line 12, Line 22) or whose negation is redundant (cf. Line 7, Line 14, Line 24) with respect to C. When removing a decision node from the second tree, the left and right subtree are joined by means of JOIN (cf. Line 10): in order to extrapolate the domain of the ranking function over environments on which it is not yet defined, the widening $\triangledown_{\mathrm{T}}$ invokes LEFT-UNIFICATION choosing the computational join $\sqcup_{\mathrm{F}}$ (cf. Line 3 of Algorithm 13). For this reason, since $\bot_{\mathrm{F}}$-leaves might disappear when joining subtrees, it is important to check for situations like case A before the left unification.

Remark 5.2.18 Note that, in order to limit the loss of precision, it is possible to adapt the idea presented in [CC92c] to design a domain widening parametric in a finite set of *thresholds*. Is is also possible to integrate the state-of-the-art precise widening operators proposed in [BHRZ05]. In [DU15], we used *conflict-driven learning* to obtain an improvement of the precision that is similar in spirit to using a more precise domain widening. We plan to investigate further possibilities for the domain widening as part of our future work. ■

Check for Case B and C. Next, the widening $\triangledown_{\mathrm{T}}$ has to detect situations like case B in Figure 5.10b, where at some iterate y_i the value of the termination semantics has been under-approximated, and situations like case C in Figure 5.10b, where the domain of the termination semantics has been over-approximating including a non-terminating loop.

The following result proves in situations like case B that, given $t \in \mathcal{T}$, a sound over-approximations $R \in \mathcal{D}$ of $\tau_{\mathrm{I}}(l)$ and a sound-overapproximation $D \in \mathcal{D}$ of $\tau_{\mathrm{I}}(f[\![\,\texttt{while}\ ^{l}\mathit{bexp}\ \texttt{do}\ \mathit{stmt}\ \texttt{od}\,]\!])$, an additional iterate of $\phi^{\natural}_{\mathrm{Mt}}$ strictly increases the value of the abstraction for the incriminated environments:

Algorithm 16 : Check Case B or C

```
1:  function CASEBORC-AUX(t1, t2, C)                    ▷ t1, t2 ∈ 𝒯, C ∈ 𝒫(𝒞)
2:      if ISLEAF(t1) ∧ ISLEAF(t2) then
3:          if ISDEFINED(t1.f) ∧ ¬(t2.f ≼F [αC(C)] t1.f) then
4:              return LEAF : ⊤F
5:          else return t2.f
6:      else if ISNODE(t1) ∧ ISNODE(t2) then
7:          l ← CASEBORC-AUX(t1.l, t2.l, C ∪ {t2.c})
8:          r ← CASEBORC-AUX(t1.r, t2.r, C ∪ {¬t2.c})
9:          return NODE{t2.c} : l; r
10:
11: function CASEBORC(D, t1, t2)                        ▷ D ∈ 𝒟, t1, t2 ∈ 𝒯
12:     return CASEBORC-AUX(t1, t2, γC(D))
```

Lemma 5.2.19 *Let* $w \stackrel{def}{=} \tau_{Mt}[\![\texttt{while}\ ^{l}bexp\ \texttt{do}\ stmt\ \texttt{od}]\!]\gamma_T[D]t$ *and, for some iterate* y_i*, let* $\gamma_T[R]y_i(\rho) < w(\rho)$*, for some* $\rho \in \mathrm{dom}(w) \cap \mathrm{dom}(\gamma_T[R]y_i)$*. For all* $\rho \in \mathrm{dom}(\gamma_T[R]\phi^{\natural}_{Mt}(y_i)) \setminus \mathrm{dom}(w)$*, we have* $\gamma_T[R]y_i(\rho) < \gamma_T[R]\phi^{\natural}_{Mt}(y_i)(\rho)$*.*

Proof.
For some iterate y_i, let $\gamma_{\mathrm{T}}[R]y_i(\rho) < w(\rho)$, for some environment $\rho \in \mathrm{dom}(w) \cap \mathrm{dom}(\gamma_{\mathrm{T}}[R]y_i)$. It means that the state $\langle l, \rho\rangle \in \Sigma$ belongs to a program execution trace $\sigma \in \Sigma^+$ whose length from $\langle l, \rho\rangle$ is greater than $\gamma_{\mathrm{T}}[R]y_i(\rho)$. Let $\langle l', \rho'\rangle \in \Sigma$ where $\rho' \in \mathrm{dom}(w) \cap \mathrm{dom}(\gamma_{\mathrm{T}}[R]y_i)$ and $w(\rho') \leq \gamma_{\mathrm{T}}[R]y_i(\rho')$, be a state reachable from $\langle l, \rho\rangle$ on σ. Without loss of generality, we can assume that σ from $\langle l, \rho\rangle$ is the longest program execution trace from $\langle l, \rho\rangle$ and that $\langle l', \rho'\rangle$ is the immediate successor of $\langle l, \rho\rangle$: $\langle\langle l, \rho\rangle, \langle l', \rho'\rangle\rangle \in \tau$. Thus, by definition of ϕ (cf. Equation 4.3.5), by Lemma 5.2.16 and by definition of $\preccurlyeq$ (cf. Equation 4.2.12), we have $w(\rho) \leq \phi(\gamma_{\mathrm{T}}[R]y_i)(\rho) \leq \gamma_{\mathrm{T}}[R]\phi^{\natural}_{\mathrm{Mt}}(y_i)(\rho)$. This concludes the proof that $\gamma_{\mathrm{T}}[R]y_i(\rho) < \gamma_{\mathrm{T}}[R]\phi^{\natural}_{\mathrm{Mt}}(y_i)(\rho)$. ■

The following result proves in situations like case C that, given $t \in \mathcal{T}$, a sound over-approximation $R \in \mathcal{D}$ of $\tau_{\mathrm{I}}(l)$ and a sound-overapproximation $D \in \mathcal{D}$ of $\tau_{\mathrm{I}}(f[\![\texttt{while}\ ^{l}bexp\ \texttt{do}\ stmt\ \texttt{od}]\!])$, an additional iterate of $\phi^{\natural}_{\mathrm{Mt}}$ strictly increases the value of the abstraction for the incriminated environments:

Lemma 5.2.20 *Let* $w \stackrel{def}{=} \tau_{Mt}[\![\texttt{while}\ ^{l}bexp\ \texttt{do}\ stmt\ \texttt{od}]\!]\gamma_T[D]t$ *and, for some iterate* y_i*, let* $\mathrm{dom}(\gamma_T[R]y_i)\setminus\mathrm{dom}(w) \neq \emptyset$*. Then, for all* $\rho \in \mathrm{dom}(\gamma_T[R]\phi^{\natural}_{Mt}(y_i))\setminus \mathrm{dom}(w)$ *in case C of Figure 5.10b, we have* $\gamma_T[R]y_i(\rho) < \gamma_T[R]\phi^{\natural}_{Mt}(y_i)(\rho)$*.*

Proof.
Let $\mathrm{dom}(\gamma_{\mathrm{T}}[R]y_i) \setminus \mathrm{dom}(w) \neq \emptyset$, for some iterate y_i. It means that there exists at least an environment $\rho \in \mathrm{dom}(\gamma_{\mathrm{T}}[R]y_i)$ such that the state $\langle l, \rho \rangle \in \Sigma$ belongs to a non-terminating program execution trace $\sigma \in \Sigma^{\omega}$. We assume to be in case C of Figure 5.10b. For all $\rho \in \mathrm{dom}(\gamma_{\mathrm{T}}[R]y_i) \setminus \mathrm{dom}(w) \neq \emptyset$, by definition of ϕ (cf. Equation 4.3.5), by Lemma 5.2.16, and by definition of $\preccurlyeq$ (cf. Equation 4.2.12), we have $\gamma_{\mathrm{T}}[R]y_i(\rho) < \phi(\gamma_{\mathrm{T}}[R]y_i)(\rho) \leq \gamma_{\mathrm{T}}[R]\phi^{\natural}_{\mathrm{Mt}}(y_i)(\rho)$. This concludes the proof, for case C. ■

Note that, an additional iterate of $\phi^{\natural}_{\mathrm{Mt}}$ is not able to distinguish between an under-approximation of the value of the termination semantics as in case B, and an over-approximation of its domain of definition as in case C.

Therefore, the widening $\triangledown_{\mathrm{T}}$ has to detect whether the value of some defined leaf node in y_i has increased in $\phi^{\natural}_{\mathrm{Mt}}(y_i)$ and, in such case, turn it into a $\top_{\mathrm{F}}$-leaf in order to prevent an indefinite growth. This check is implemented by Algorithm 16: the main function CASEBORC, given a sound over-approximation $D \in \mathcal{D}$ of the reachable environments and two decision trees $t_1, t_2 \in \mathcal{T}$, calls the auxiliary function CASEBORC-AUX, which collects into a set $C \in \mathcal{P}(\mathcal{C})$ (initially equal to $\gamma_{\mathrm{C}}(D)$, cf. Line 12) the linear constraints encountered along the paths up to the leaf nodes (cf. Lines 7-8), which are compared (cf. Line 3) and, in case, turned into a $\top_{\mathrm{F}}$-leaf (cf. Line 4).

Remark 5.2.21 Note that, it is also possible to allow the ranking function to increase using a finite set of *thresholds* instead of returning directly $\top_{\mathrm{F}}$. In this way, in situations like case C, it simply delays the iteration that must return $\top_{\mathrm{F}}$. However, in situations like case B, it may discover the correct ranking function and avoid losing precision. ■

Value Widening Finally, the widening $\triangledown_{\mathrm{T}}$ extrapolates the value of the ranking function over the environments on which it was not defined before the domain widening. The heuristic that we chose consists in widening leaf nodes with respect to their *adjacent* leaf nodes. The rationale being that programs often loop over consecutive values of a variable, to infer the shape of the ranking function over the environments on which it was not defined we use the information available in adjacent partitions of its domain of definition.

We define an extrapolation operator $\blacktriangledown_{\mathrm{F}}[D_1, D_2]$ between two defined leaf nodes $f_1, f_2 \in \mathcal{F} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}$, given the numerical abstractions $D_1 \in \mathcal{D}$ and $D_2 \in \mathcal{D}$ representing their path from the root of the decision tree.

In particular, given two affine functions $f_1, f_2 \in \mathcal{F}_\mathrm{A} \setminus \{\bot_\mathrm{F}, \top_\mathrm{F}\}$, the operator extrapolates the affine function $f \in \mathcal{F}_\mathrm{A} \setminus \{\bot_\mathrm{F}, \top_\mathrm{F}\}$, whose hypograph $f[D_2]{\downarrow}$ within D_2 is the *convex-hull* of the hypographs $f'[D_2]{\downarrow}$ of the affine functions $f' \in \mathcal{F}_\mathrm{A} \setminus \{\bot_\mathrm{F}, \top_\mathrm{F}\}$ lying on the boundaries of *convex-hull*$\{f_1[D_1]{\downarrow}, f_2[D_2]{\downarrow}\}$; when such affine function does not exist, the result is $\top_\mathrm{F}$:

$$f_1 \blacktriangledown_\mathrm{F}[D_1, D_2]\ f_2 \stackrel{\text{def}}{=} \begin{cases} f & f[D_2]{\downarrow} = \textit{convex-hull}\{f'[D_2]{\downarrow} \mid f' \in F\} \\ \top_\mathrm{F} & \text{otherwise} \end{cases} \tag{5.2.25}$$

where $F \stackrel{\text{def}}{=} \{f \in \mathcal{F}_\mathrm{A} \setminus \{\bot_\mathrm{F}, \top_\mathrm{F}\} \mid f[D_1 \sqcup_\mathrm{D} D_2]{\downarrow} \supseteq \textit{convex-hull}\{f_1[D_1]{\downarrow}, f_2[D_2]{\downarrow}\}\}$ is the minimal set of affine functions whose hypographs within $D_1 \sqcup_\mathrm{D} D_2$ determine *convex-hull*$\{f_1[D_1]{\downarrow}, f_2[D_2]{\downarrow}\}$, that is, $\forall f \in F : \nexists f' \in F : f[D_1 \sqcup_\mathrm{D} D_2]{\downarrow} \supset f'[D_1 \sqcup_\mathrm{D} D_2]{\downarrow} \supseteq \textit{convex-hull}\{f_1[D_1]{\downarrow}, f_2[D_2]{\downarrow}\}$ and $\bigcap_{f \in F} f[D_1 \sqcup_\mathrm{D} D_2]{\downarrow} = \textit{convex-hull}\{f_1[D_1]{\downarrow}, f_2[D_2]{\downarrow}\}$.

To clarify, let us consider the following example:

Example 5.2.22

Two affine functions $f_1, f_2 \in \mathcal{F}_\mathrm{A} \setminus \{\bot_\mathrm{F}, \top_\mathrm{F}\}$ are shown in Figure 5.11a and Figure 5.11b, respectively. Their domains of definition $D_1 \in \mathcal{D}$ and $D_2 \in \mathcal{D}$ are represented by the set of linear constraints $C_1 \stackrel{\text{def}}{=} \gamma_\mathrm{C}(D_1) = \{x - 6 \geq 0, -x + 10 \geq 0\}$ and $C_2 \stackrel{\text{def}}{=} \gamma_\mathrm{C}(D_2) = \{-x + 5 \geq 0\}$. In Figure 5.11d and Figure 5.11e are represented the hypographs $f_1[D_1]{\downarrow}$ and $f_2[D_2]{\downarrow}$ of f_1 and f_2, respectively. Their convex-hull is shown in Figure 5.11e, and the result of the extrapolation is the affine function $f \in \mathcal{F}_\mathrm{A} \setminus \{\bot_\mathrm{F}, \top_\mathrm{F}\}$ shown in Figure 5.11c.

The heuristic for widening adjacent leaf nodes within decision trees is implemented by Algorithm 17: the function WIDEN-AUX in Algorithm 17 is given as input two decision trees $t_1, t_2 \in \mathcal{T}$, a copy $t \in \mathcal{T}$ of t_1 and a set $C \in \mathcal{P}(\mathcal{C})$ initially equal to $\gamma_\mathrm{C}(D)$, where $D \in \mathcal{D}$ is a sound over-approximation of the reachable environments (cf. Line 5 of Algorithm 13). It accumulates into C the linear constraints accumulated along the paths up to the leaf nodes (cf. Lines 36-37). The leaf nodes defined in t_2 and undefined in t_1 (cf. Line 29) are extrapolated with respect to their adjacent *defined* leaf nodes (cf. Line 31) by means of the extrapolation operator $\blacktriangledown_\mathrm{F}$ (cf. Line 32).

The adjacent leaf nodes are determined by the function ADJACENT (cf. Line 31). Note that, not all linear constraints appearing in a decision tree necessarily appear along a path in the decision tree. For this reason, given a decision tree $t \in \mathcal{T}$ and a set $C \in \mathcal{P}(\mathcal{C})$ of linear constraints encountered along

Algorithm 17 : Tree Value Widening

```
1:  function CONSTRAINTS(t)                                          ▷ t ∈ 𝒯
2:      if ¬ISNODE(t) then return ∅
3:      else if ISNODE(t) then
4:          return CONSTRAINTS(t.l) ∪ {t.c} ∪ CONSTRAINTS(t.r)
5:
6:  function LEAF(t, C, J)                                           ▷ t ∈ 𝒯, C, J ∈ 𝒫(𝒞)
7:      if ¬ISNODE(t) then
8:          if ISLEAF(t) ∧ ISDEFINED(t.f) then return {(α_C(C), t)}
9:          else return ∅
10:     else if ISNODE(t) then
11:         j ← max J          ▷ j is the largest linear constraint appearing in J
12:         if t.c <_C j then return LEAF(t, C, J \ {j})
13:         else if j <_C t.c then return LEAF(t.r, C ∪ {¬t.c}, J \ {j})
14:         else return LEAF(t.l, C ∪ {t.c}, J \ {j})
15:
16: function ADJACENT(t, C)                                          ▷ t ∈ 𝒯, C ∈ 𝒫(𝒞)
17:     K ← C
18:     for all c ∈ CONSTRAINTS(t) do
19:         if c ∉ K ∧ ¬c ∉ K then
20:             if ISREDUNDANT(c, K) then K ← K ∪ {c}
21:             else if ISREDUNDANT(¬c, K) then K ← K ∪ {¬c}
22:     A ← ∅
23:     for all c ∈ C do A ← A ∪ LEAF(t, ∅, K \ {c} ∪ {¬c})
24:     return A
25:
26: function WIDEN-AUX(t, t_1, t_2, C)                               ▷ t, t_1, t_2 ∈ 𝒯, C ∈ 𝒫(𝒞)
27:     if ISNIL(t_1) ∨ ISNIL(t_2) then return FAIL("invalid argument")
28:     else if ISLEAF(t_1) ∧ ISLEAF(t_2) then
29:         if ¬ISDEFINED(t_1.f) ∧ ISDEFINED(t_2.f) then
30:             f ← t_2.f
31:             for all (d_1, f_1) ∈ ADJACENT(t, C) do
32:                 f ← (f_1 ▼_F[d_1, α_C(C)] t_2.f) ⊔_F[α_C(C)] f
33:             return f
34:         else return t_2.f
35:     else if ISNODE(t_1) ∧ ISNODE(t_2) then
36:         l ← WIDEN-AUX(t, t_1.l, t_2.l, C ∪ {t_2.c})
37:         r ← WIDEN-AUX(t, t_1.r, t_2.r, C ∪ {¬t_2.c})
38:         return NODE{t_2.c} : l; r
```

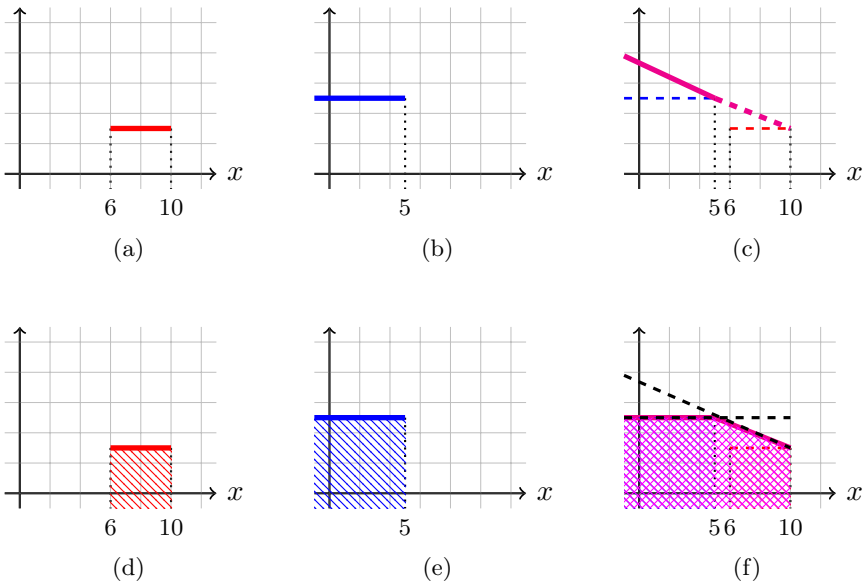


Figure 5.11: Example of extrapolation between two affine functions of one variable, shown in (a) and (b), respectively. The hypographs of (a) and (b) are respectively shown in (d) and (e). Their convex-hull is shown in (f) together with the affine functions lying on its boundaries. The result of the extrapolation is shown in (c).

the path to a leaf node, the function adds to a set $K \in \mathcal{P}(\mathcal{C})$ (initially equal to C, cf. Line 17) all the redundant constraints (cf. Lines 20-21) that appear in the decision tree (cf. Line 18) but are missing from the path represented by C (cf. Line 19). Then, the adjacent leaf nodes are sought negating one by one in K the linear constraint in C, and are collected into a set A (cf. Line 23).

The search for a leaf node determined by a set of constraints $J \in \mathcal{P}(\mathcal{C})$ is conducted by the function LEAF: the function extracts from J the linear constraints in decreasing order (cf. Line 11) and uses them to choose a path in the decision tree while accumulating into an initial empty set $C \in \mathcal{P}(\mathcal{C})$ the linear constraints effectively encountered (cf. Lines 12-14). In case the path leads to a defined leaf node, the function returns a set containing the leaf node together with the numerical abstract domain $\alpha_{\mathrm{C}}(C) \in \mathcal{D}$ representing its path (cf. Line 8). Otherwise, the function returns an empty set (cf. Line 9).

The linear constraints appearing in a decision tree are determined by the function CONSTRAINTS by visiting in-order the decision tree (cf. Line 4).

Remark 5.2.23 Note that, besides establishing relationships between adjacent leaf nodes, other value widening heuristics are possible. An example would be establishing relationships between leaf nodes based on the parity of some variable, or based on numerical relationships between variables. We plan to investigate these possibilities as part of our future work. ■

The following result provides, for a loop `while` ^{l}*bexp* `do` *stmt* `od`, a sound over-approximation of $\tau_{\mathrm{Mt}}[\![$ `while` ^{l}*bexp* `do` *stmt* `od` $]\!]$ defined in Equation 4.3.4, given a decision tree $t \in \mathcal{T}$, a sound over-approximation $R \in \mathcal{D}$ of $\tau_{\mathrm{I}}(l)$, and a sound over-approximation $D \in \mathcal{D}$ of $\tau_{\mathrm{I}}(f[\![$ `while` ^{l}*bexp* `do` *stmt* `od` $]\!])$:

Lemma 5.2.24 *Let* $\phi^{\natural}_{Mt}(x) \stackrel{def}{=} F^{\natural}_1[x] \curlyvee_T [R]\ F^{\natural}_2[t]$ *as defined in Lemma 5.2.16 for any given* $t \in \mathcal{T}$*. Then, we have:*

$$\tau_{Mt}[\![\texttt{while}\ ^l bexp\ \texttt{do}\ stmt\ \texttt{od}]\!]\gamma_T[D]t \preccurlyeq \gamma_T[R](\mathrm{lfp}^{\natural}\ \phi^{\natural}_{Mt})$$

where $\mathrm{lfp}^{\natural}\ \phi^{\natural}_{Mt}$ *is the limit of the iteration sequence with widening:* $y_0 \stackrel{def}{=} \bot_T$, $y_{n+1} \stackrel{def}{=} y_n \triangledown \phi^{\natural}_{Mt}(y_n)$ *(cf. Equation 5.2.24).*

Proof.
See Appendix A.3. ■

5.3 Abstract Definite Termination Semantics

The operators of the decision trees abstract domains can now be used to define the abstract definite termination semantics.

Note that, pure backward analysis is blind with respect to the program initial states. For this reason, in the following, we assume to have, for each program control point $l \in \mathcal{L}$, a sound numerical over-approximation $R \in \mathcal{D}$ of the reachable environments $\tau_{\mathrm{I}}(l) \in \mathcal{P}(\mathcal{E})$: $\tau_{\mathrm{I}}(l) \subseteq \gamma_{\mathrm{D}}(R)$ (cf. Section 3.4).

In Figure 5.12 we define the semantics $\tau^{\natural}_{\mathrm{Mt}}[\![\,stmt]\!]\colon \mathcal{T} \to \mathcal{T}$, for each program instruction *stmt*. Each function $\tau^{\natural}_{\mathrm{Mt}}[\![\,stmt]\!]\colon \mathcal{T} \to \mathcal{T}$ takes as input a decision tree over-approximating the ranking function corresponding to the final control point of the instruction, and outputs a decision tree defined over a subset of the reachable environments at $i[\![\,stmt]\!]$, which over-approximates the ranking function corresponding to the initial control point of the instruction.

The abstract termination semantics $\tau^{\natural}_{\mathrm{Mt}}[\![\,prog]\!] \in \mathcal{T}$ of a program *prog* outputs the decision tree over-approximating the ranking function corresponding

$$
\begin{array}{l}
\tau^{\natural}_{\mathrm{Mt}}[\![\, {}^{l}\mathtt{skip} \,]\!]t \stackrel{\mathrm{def}}{=} \mathrm{STEP}_{\mathrm{T}}(t) \\
\tau^{\natural}_{\mathrm{Mt}}[\![\, {}^{l}X := aexp \,]\!]t \stackrel{\mathrm{def}}{=} (\text{B-ASSIGN}_{\mathrm{T}}[\![\, X := aexp \,]\!]R)(t) \\
\tau^{\natural}_{\mathrm{Mt}}[\![\, \mathtt{if}\ {}^{l}bexp\ \mathtt{then}\ stmt_1\ \mathtt{else}\ stmt_2\ \mathtt{fi} \,]\!]t \stackrel{\mathrm{def}}{=} F^{\natural}_1[t] \curlyvee_{\mathrm{T}}[R]\ F^{\natural}_2[f] \\
\qquad F^{\natural}_1[t] \stackrel{\mathrm{def}}{=} (\mathrm{FILTER}_{\mathrm{T}}[\![\, bexp \,]\!]R)(\tau^{\natural}_{\mathrm{Mt}}[\![\, stmt_1 \,]\!]t) \\
\qquad F^{\natural}_2[t] \stackrel{\mathrm{def}}{=} (\mathrm{FILTER}_{\mathrm{T}}[\![\, \mathtt{not}\ bexp \,]\!]R)(\tau^{\natural}_{\mathrm{Mt}}[\![\, stmt_2 \,]\!]t) \\
\tau^{\natural}_{\mathrm{Mt}}[\![\, \mathtt{while}\ {}^{l}bexp\ \mathtt{do}\ stmt\ \mathtt{od} \,]\!]t \stackrel{\mathrm{def}}{=} \mathrm{lfp}^{\natural}\ \phi^{\natural}_{\mathrm{Mt}} \\
\qquad \phi^{\natural}_{\mathrm{Mt}}(x) \stackrel{\mathrm{def}}{=} F^{\natural}[x] \curlyvee_{\mathrm{T}}[R]\ (\mathrm{FILTER}_{\mathrm{T}}[\![\, \mathtt{not}\ bexp \,]\!]R)(t) \\
\qquad F^{\natural}[x] \stackrel{\mathrm{def}}{=} (\mathrm{FILTER}_{\mathrm{T}}[\![\, bexp \,]\!]R)(\tau^{\natural}_{\mathrm{Mt}}[\![\, stmt \,]\!]x) \\
\tau^{\natural}_{\mathrm{Mt}}[\![\, stmt_1\ stmt_2 \,]\!]t \stackrel{\mathrm{def}}{=} \tau^{\natural}_{\mathrm{Mt}}[\![\, stmt_1 \,]\!](\tau^{\natural}_{\mathrm{Mt}}[\![\, stmt_2 \,]\!]t)
\end{array}
$$

Figure 5.12: Abstract termination semantics of instructions *stmt*.

to the initial program control point $i[\![\, prog \,]\!] \in \mathcal{L}$. It is defined taking as input the leaf node $\mathrm{LEAF} : \lambda X_1, \ldots, X_k.\ 0$ as:

Definition 5.3.1 (Abstract Termination Semantics) *The* abstract termination semantics $\tau^{\natural}_{Mt}[\![\, prog \,]\!] \in \mathcal{T}$ *of a program prog is:*

$$\tau^{\natural}_{Mt}[\![\, prog \,]\!] = \tau^{\natural}_{Mt}[\![\, stmt\ {}^{l} \,]\!] \stackrel{def}{=} \tau^{\natural}_{Mt}[\![\, stmt \,]\!]\mathrm{LEAF} : \lambda X_1, \ldots, X_k.\ 0 \qquad (5.3.1)$$

where the abstract termination semantics $\tau^{\natural}_{Mt}[\![\, stmt \,]\!] \in \mathcal{T} \to \mathcal{T}$ *of each program instruction stmt is defined in Figure 5.12.*

The following result proves the soundness of the abstract termination semantics $\tau^{\natural}_{\mathrm{Mt}}[\![\, prog \,]\!] \in \mathcal{T}$ with respect to the termination semantics $\tau_{\mathrm{Mt}}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$, given a sound numerical over-approximation $R \in \mathcal{D}$ of the reachable environments $\tau_{\mathrm{I}}(i[\![\, prog \,]\!])$:

Theorem 5.3.2 $\tau_{Mt}[\![\, prog \,]\!] \preccurlyeq \gamma_T[R]\tau^{\natural}_{Mt}[\![\, prog \,]\!]$

Proof (Sketch).
The proof follows from the soundness of the operators of the decision trees abstract domain (cf. Lemma 5.2.8, Lemma 5.2.14, Lemma 5.2.15, Lemma 5.2.16, and Lemma 5.2.24) used for the definition of $\tau^{\natural}_{\mathrm{Mt}}[\![\, prog \,]\!] \in \mathcal{T}$. ■

In particular, the abstract termination semantics provides *sufficient preconditions* for ensuring definite termination of a program for a given over-approximation $R \in \mathcal{D}$ of the set of initial states $\mathcal{I} \subseteq \Sigma$:

Corollary 5.3.3 *A program must terminate for execution traces starting from a given set of initial states* $\gamma_D(R)$ *if* $\gamma_D(R) \subseteq \mathrm{dom}(\gamma_T[R]\tau^{\natural}_{Mt}[\![\, prog \,]\!])$.

Examples. In the following, we recall the examples introduced at the beginning of the chapter and we present the fully detailed analyses of the program using the abstract domain of decision trees.

Example 5.3.4

Let us consider again the program of Example 5.1.1 [PR04a]:

$$\begin{array}{l} \texttt{while}\ ^{1}(x \geq 0)\ \texttt{do} \\ \quad ^{2}x := -2x + 10 \\ \texttt{od}^{3} \end{array}$$

We present the analysis of the program using interval constraints based on the *intervals abstract domain* (cf. Section 3.4.1) for the decision nodes, and *affine functions* for the leaf nodes (cf. Equation 5.2.6).

The starting point is the zero function at the program final control point:

$$\mathbf{3}: \quad \textsc{leaf} : \lambda x.\ 0$$

The ranking function is then propagated backwards towards the program initial control point taking the loop into account:

$$\mathbf{1}: \quad \textsc{node}\{x \geq 0\} : (\textsc{leaf} : \bot_{\mathrm{F}}); (\textsc{leaf} : \lambda x.\ 1)$$

The first iterate of the loop is able to conclude that the program terminates in at most one program step if the loop condition $x \geq 0$ is not satisfied. Then, at program control point **2**, the operator $\textsc{assign}_{\mathrm{T}}$ replaces the program variable x with the expression $-2x + 10$ within the decision tree:

$$\mathbf{2}: \quad \textsc{node}\{x - 6 \geq 0\} : (\textsc{leaf} : \lambda x.\ 2); (\textsc{leaf} : \bot_{\mathrm{F}})$$

Note that, $\textsc{assign}_{\mathrm{T}}$ has also increased the value of the ranking function in order to count one more program execution step to termination. The second iterate of the loop concludes that the program terminates in at most three execution steps, when $x \geq 6$ and thus the loop is executed only once, and in at most one program step, when the loop is not entered:

$$\begin{array}{ll} \mathbf{1}: & \textsc{node}\{x - 6 \geq 0\} : \\ & \quad \textsc{leaf} : \lambda x.\ 3; \\ & \quad \textsc{node}\{x \geq 0\} : (\textsc{leaf} : \bot_{\mathrm{F}}); (\textsc{leaf} : \lambda x.\ 1) \end{array}$$

In this particular case, there is no need for convergence acceleration and the analysis is rather precise: at the sixth iteration, a fixpoint is reached providing the decision tree shown in Figure 5.1.

In the following example, the widening is required for convergence.

Example 5.3.5

Let us consider again the program of Example 5.1.2 [UM14b]:

$$\begin{array}{l} \texttt{while}\ ^{1}(r > 0)\ \texttt{do} \\ \quad ^{2}r := r + x \\ \quad ^{3}r := r - y \\ \texttt{od}^{4} \end{array}$$

We present the analysis of the program using polyhedral constraints based on the *polyhedra abstract domain* (cf. Section 3.4.2) for the decision nodes, and *affine functions* for the leaf nodes (cf. Equation 5.2.6).

The starting point is the zero function at the program final control point:

$$\mathbf{3}: \quad \text{LEAF} : \lambda x.\lambda y.\lambda r.\ 0$$

The ranking function is then propagated backwards towards the program initial control point taking the loop into account:

$$\mathbf{1}: \quad \text{NODE}\{r - 1 \geq 0\} : (\text{LEAF} : \bot_{\text{F}}); (\text{LEAF} : \lambda x.\lambda y.\lambda r.\ 1)$$

The first iterate of the loop is able to conclude that the program terminates in at most one program step if the loop condition $r > 0$ is not satisfied. Then, at program control point **3**, the operator ASSIGN_{T} replaces the program variable r with the expression $r - y$ within the decision tree:

$$\mathbf{3}: \quad \text{NODE}\{y - r \geq 0\} : (\text{LEAF} : \lambda x.\lambda y.\lambda r.\ 2); (\text{LEAF} : \bot_{\text{F}})$$

and at program control point **2**, it replaces r with $r + x$:

$$\mathbf{2}: \quad \text{NODE}\{x - y + r - 1 \geq 0\} : (\text{LEAF} : \bot_{\text{F}}); (\text{LEAF} : \lambda x.\lambda y.\lambda r.\ 3)$$

The second iterate of the loop concludes that the program terminates in at most four execution steps, when the loop is executed only once, and in at most one program step, when the loop is not entered:

$$\begin{array}{ll} \mathbf{1}: & \text{NODE}\{r - 1 \geq 0\} : \\ & \quad \text{NODE}\{x - y + r - 1 \geq 0\} : (\text{LEAF} : \bot_{\text{F}}); (\text{LEAF} : \lambda x.\lambda y.\lambda r.\ 4); \\ & \quad \text{LEAF} : \lambda x.\lambda y.\lambda r.\ 1 \end{array}$$

The third iterate concludes that the program terminates in at most seven execution steps when the loop is executed twice, in at most four steps when the loop is executed once, and in at most one step when the loop is not entered:

$$
\begin{array}{ll}
\mathbf{1}: & \textsc{node}\{r-1 \geq 0\}: \\
& \quad \textsc{node}\{x-y+r-1 \geq 0\}: \\
& \quad\quad \textsc{node}\{2x-2y+r-1 \geq 0\}: (\textsc{leaf}: \bot_{\mathrm{F}}); (\textsc{leaf}: \lambda x.\lambda y.\lambda r.\ 7); \\
& \quad\quad \textsc{leaf}: \lambda x.\lambda y.\lambda r.\ 4 \\
& \quad \textsc{leaf}: \lambda x.\lambda y.\lambda r.\ 1
\end{array}
$$

The widening that we defined in Section 5.2.4 extrapolates the ranking function on the partitions over which it is not yet defined:

$$
\begin{array}{ll}
\mathbf{1}: & \textsc{node}\{r-1 \geq 0\}: \\
& \quad \textsc{node}\{x-y+r-1 \geq 0\}: (\textsc{leaf}: \lambda x.\lambda y.\lambda r.\ 7); (\textsc{leaf}: \lambda x.\lambda y.\lambda r.\ 4); \\
& \quad \textsc{leaf}: \lambda x.\lambda y.\lambda r.\ 1
\end{array}
$$

Note, in particular, that the ranking function is now *temporarily* defined even when $x \geq y$ and the program does not terminate. Indeed, the fourth iterate of the while loop identifies a situation like case C in Figure 5.10b:

$$
\begin{array}{ll}
\mathbf{1}: & \textsc{node}\{r-1 \geq 0\}: \\
& \quad \textsc{node}\{x-y+r-1 \geq 0\}: (\textsc{leaf}: \lambda x.\lambda y.\lambda r.\ 10); (\textsc{leaf}: \lambda x.\lambda y.\lambda r.\ 4); \\
& \quad \textsc{leaf}: \lambda x.\lambda y.\lambda r.\ 1
\end{array}
$$

and the widening corrects the ranking function yielding a fixpoint:

$$
\begin{array}{ll}
\mathbf{1}: & \textsc{node}\{r-1 \geq 0\}: \\
& \quad \textsc{node}\{x-y+r-1 \geq 0\}: (\textsc{leaf}: \top_{\mathrm{F}}); (\textsc{leaf}: \lambda x.\lambda y.\lambda r.\ 4); \\
& \quad \textsc{leaf}: \lambda x.\lambda y.\lambda r.\ 1
\end{array}
$$

The fixpoint represents the following piecewise-defined ranking function:

$$
\lambda\rho. \begin{cases} 1 & \rho(r) \geq 1 \\ 4 & 0 \leq \rho(r) \wedge \rho(r) \leq \rho(y) - \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}
$$

which proves that the program is terminating in at most 4 program execution steps if the initial value of the program variable r is smaller than or equal to the difference between the initial value of the program variable y and the initial value of the program variable x.

In Remark 5.2.18, we mentioned various possibilities to improve the precision of the widening operator. In particular, an adaptation of the domain widening using the *evolving rays* technique proposed in [BHRZ05] yields the more precise decision tree shown in Figure 5.2. The same result can be obtained using conflict-driven learning [DU15].

5.4 Related Work

We conclude the chapter with a discussion of the most relevant related work.

Decision Trees. The use of (binary) decision trees (Binary Decision Diagrams [Bry86], in particular) for verification has been devoted a large body of work, especially in the area of timed-systems and hybrid-system verification [Jea02, LPWY99, MLAH99, etc.].

In this thesis, we focus on common program analysis applications and, in this sense, our decision trees abstract domain is mostly related to the ones presented in [CCM10, GC10a]: both ours and these abstract domains are based on decision trees extended with linear constraints. However, the abstract domains proposed in [CCM10, GC10a] are designed for the disjunctive refinement of numerical abstract domains, while our abstract domain is designed specifically in order manipulate ranking functions. Moreover, while our abstract domain is based on *binary* decision trees, in [CCM10] the choices at the decision nodes may differ at each node and their number is not bounded a priori.

In general, despite all the available alternatives [BCC+10, CCM10, GR98, GC10a, GC10b, SISG06, etc.], it seems to us that in the literature there is no disjunctive abstract domain well-suited for program termination. A first (minor) reason is the fact that most of the existing disjunctive abstract domains are designed specifically for forward analyses while ranking functions are inferred through backward analysis (cf. Section 5.3). However, the main reason is that adapting existing widening operators to ranking functions is not obvious due to the coexistence of an approximation and computational ordering in the termination semantics domain (cf. Section 4.3 and Section 5.2.4).

Termination. In the recent past, termination analysis has benefited from many research advances and powerful termination provers have emerged over the years [BCF13, CPR06, GSKT06, HHLP13, LQC15].

Many results in this area [BCC+07, CPR06, etc.] are based on the transition invariants method introduced in [PR04b]. In particular, the Terminator prover [CPR06] is based on an algorithm for the iterative construction of transition invariants. This algorithm searches within a program for single paths representing potential counterexamples to termination, computes a ranking function for each one of them individually (as in [PR04a]), and combines the obtained ranking functions into a single termination argument. Our approach differs in that it aims at proving termination for all program paths at the same time, without resorting to counterexample-guided analysis. In particular, we emphasize that our approach is able to deal with arbitrary control

structures, and thus it is not limited to simple loops as [PR04a] or to non-nested loops as [BMS05a]. Moreover, it avoids the cost of explicit checking for the well-foundedness of the termination argument. The approach presented in [TSWK11] shares similar motivations, but prefers loop summarization to iterative fixpoint computation with widening, as considered in this thesis.

The majority of the literature is based on the *indirect* use of invariants for proving termination [ADFG10, BCC+07, BCF13, CS02, LORCR13, etc.]. On the other hand, our approach infers and manipulates ranking functions *directly* as invariants associated to program control points. In this sense, [ADFG10] is the closest approach to ours: the invariants are pre-computed, but each program point is assigned with a ranking function (that also provides information on the execution time in terms of execution steps), as in our approach.

The strength of our approach is being an abstract interpretation of a *complete* semantics for termination. For any given terminating program, it is always possible to design an abstraction able to prove its termination. In particular, this is stronger than fixing a priori an incomplete reasoning method that can miss terminating programs. For example, various methods to synthesize ranking functions based on linear programming [ADFG10, CS01, PR04a, etc.] are complete for programs with *rational*-valued variables, but not with integer-valued variables. Indeed, as we have seen in Example 5.1.1 at the beginning of the chapter, there are programs that terminate over the integers but do not terminate over the rationals.

Finally, in the literature, we found only few works that have addressed the problem of automatically inferring *preconditions* for program termination. In [CGLA+08], the authors proposed a method based on preconditions generating ranking functions from potential ranking functions, while our preconditions are inherently obtained from the inferred ranking functions as the set of programs state for which the ranking function is defined. Thus, our preconditions are derived by *under-approximation* of the set of terminating states as opposed to the approaches presented in [GG13, Mas14] where the preconditions are derived by (complementing an) over-approximation of the non-terminating states.

The abstract domain that we have presented in this chapter has been instantiated only with affine functions, while several methods in the literature use more powerful lexicographic ranking functions. This limitation will be addressed in the next chapter Chapter 6 using ordinal-valued ranking functions. We postpone the comparison between our work and related work based on lexicographic ranking functions to the end of the next chapter.

6

Ordinal-Valued Ranking Functions

In this chapter, we address the limitation to natural-valued functions of the decision trees abstract domain presented in Chapter 5. In particular, we propose a functions auxiliary abstract domain based on ordinal-valued functions. More specifically, these functions are polynomials in ω, where the polynomial coefficients are natural-valued functions of the program variables. The abstract domain is parametric in the choice of the maximum degree of the polynomials, and the types of functions used as polynomial coefficients.

Dans ce chapitre, nous levons la limitation à des fonctions à valeur naturelle du domaine abstrait d'arbres de décision présentés au Chapitre 5. En particulier, nous proposons un domaine abstrait auxiliaire de fonctions basé sur des fonctions à valeurs dans les ordinaux. Plus précisément, ces fonctions sont basées sur des polynômes en ω, où les coefficients des polynômes sont des fonctions à valeur naturelle des variables du programme. Le domaine est paramétrique dans le choix du degré maximum des polynômes, et des types de fonctions utilisées comme coefficients des polynômes.

6.1 Ordinal-Valued Ranking Functions

In many cases, a single natural-valued ranking function is not sufficient. In particular, this is the case in the presence of unbounded non-determinism, as we have seen in Example 4.1.2. In order to further motivate the need for ordinal-valued ranking functions, we propose the following example.

Example 6.1.1

Let us consider the following program from [CPR11]:

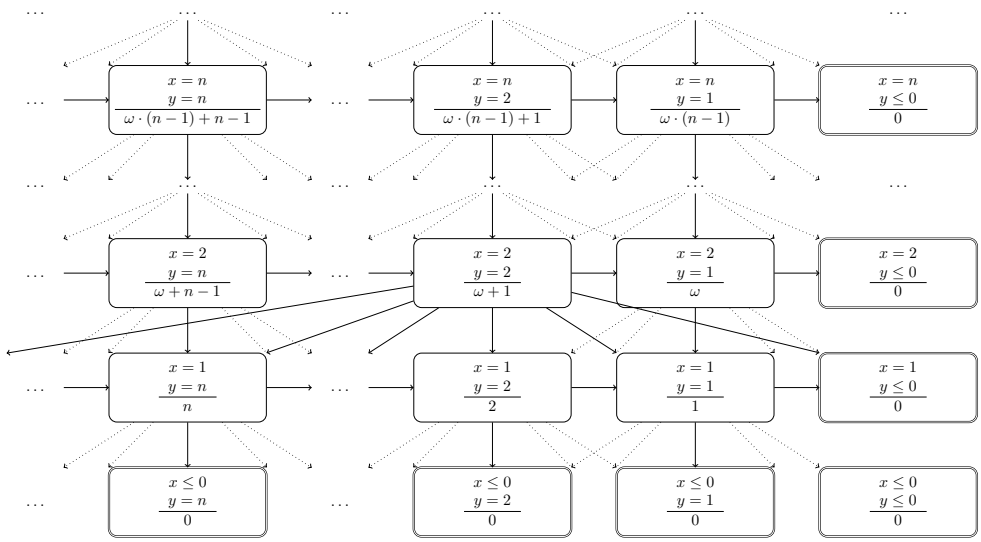


Figure 6.1: Transitions between states at program control point **1** for the program of Example 6.1.1. There is an edge from any node where y has value $k > 0$ (and $y > 0$) to all nodes where y has value $k - 1$ (and y has any value). In every node we indicate the maximum number of loop iterations needed to reach a blocking state with no successors.

```
while 1(0 < x ∧ 0 < y) do
  if 2( ? ) then
    3x := x − 1
    4y := ?
  else
    5y := y − 1
  fi
od6
```

Each loop iteration, either decrements the value of y, or decrements the value of x and resets the value of y, until either program variable becomes less than or equal to zero. There is a non-deterministic choice between the branches of the conditional `if` instruction at program control point **2**, and the value of the variable y is chosen non-deterministically at program control point **4**. The program always terminates, whatever the initial values for x and y, and whatever the non-deterministic choices during execution.

In the graph of Figure 6.1, each node represents a possible state of the program at program control point **1**, and each edge represents a loop iteration.

The nodes with a double outline are blocking states with no successor. We define a ranking function for the program following the intuition behind the definite termination semantics of Section 4.2.2: we start from the blocking states, where we assign value 0 to the function; then, we follow the edges backwards, and for each state that we encounter (whose successors belong to the domain of the function) we define the value of the ranking function as the maximum of all values of the function plus 1 for all successors of the state. Hence, we need a transfinite value whenever we encounter a state leading through unbounded non-determinism to program executions of arbitrary length. In particular, in this case, we need ordinal numbers for all states where $x > 1$ and $y > 0$

The analysis of the program using our decision trees abstract domain extended with ordinal-valued functions is proposed in Example 6.4.4

Lexicographic Ranking Functions. It is also possible to prove the termination of the program of Example 6.1.1 using a lexicographic ranking function (x, y). Indeed, a lexicographic tuple $(f_n, \ldots, f_1, f_0)$ of natural numbers is an isomorphic representation of the ordinal $\omega^n \cdot f_n + \cdots + \omega \cdot f_1 + f_0$ [MP96]. However, reasoning directly with lexicographic ranking functions, poses the additional difficulty of finding an appropriate lexicographic order. Existing methods [BMS05a, CSZ13, etc.] use heuristics to explore the space of possible orders, which grows very large with the number of program variables. Instead, the coefficients $f_n, \ldots, f_1, f_0$ (and thus their order) of our ordinal-valued ranking functions are automatically inferred by the analysis. Moreover, there exist programs for which there does not exist a lexicographic ranking function but there is a piecewise-defined ordinal-valued ranking function as considered in this chapter, and we will provide an example in Example 6.4.5. We refer to Section 6.5 at end of the chapter for further discussion on the comparison between lexicographic and ordinal-valued ranking functions.

6.2 Ordinal Arithmetic

The theory of ordinals was introduced by Georg Cantor as the core of his set theory [Can95, Can97]. We recall from Chapter 2 that the smallest ordinal is denoted by 0. The successor of an ordinal α is denoted by $\alpha+1$, or equivalently, by $\mathrm{succ}(\alpha)$. A limit ordinal is an ordinal which is neither 0 nor a successor ordinal. In the following, we provide the definition and some properties of addition, multiplication and exponentiation on ordinals [Kun80].

Addition. Ordinal addition is defined by transfinite induction:

$$\begin{aligned} \alpha + 0 &\stackrel{\text{def}}{=} \alpha && \text{(zero case)} \\ \alpha + (\beta + 1) &\stackrel{\text{def}}{=} (\alpha + \beta) + 1 && \text{(successor case)} \\ \alpha + \beta &\stackrel{\text{def}}{=} \bigcup_{\gamma<\beta} (\alpha + \gamma) && \text{(limit case)} \end{aligned} \tag{6.2.1}$$

Ordinal addition generalizes the addition of natural numbers. It is associative, i.e. $(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma)$, but not commutative, e.g. $1 + \omega = \omega \neq \omega + 1$.

Multiplication. Ordinal multiplication is also defined inductively:

$$\begin{aligned} \alpha \cdot 0 &\stackrel{\text{def}}{=} 0 && \text{(zero case)} \\ \alpha \cdot (\beta + 1) &\stackrel{\text{def}}{=} (\alpha \cdot \beta) + \alpha && \text{(successor case)} \\ \alpha \cdot \beta &\stackrel{\text{def}}{=} \bigcup_{\gamma<\beta} (\alpha \cdot \gamma) && \text{(limit case)} \end{aligned} \tag{6.2.2}$$

Ordinal multiplication generalizes the multiplication of natural numbers. It is associative, i.e. $(\alpha \cdot \beta) \cdot \gamma = \alpha \cdot (\beta \cdot \gamma)$, and left distributive, i.e. $\alpha \cdot (\beta + \gamma) = (\alpha \cdot \beta) + (\alpha \cdot \gamma)$. However, commutativity does not hold, e.g. $2 \cdot \omega = \omega \neq \omega \cdot 2$, and neither does right distributivity, e.g. $(\omega + 1) \cdot \omega = \omega \cdot \omega \neq \omega \cdot \omega + \omega$.

Exponentiation. We define ordinal exponentiation by transfinite induction:

$$\begin{aligned} \alpha^0 &\stackrel{\text{def}}{=} 1 && \text{(zero case)} \\ \alpha^{\beta+1} &\stackrel{\text{def}}{=} (\alpha^\beta) \cdot \alpha && \text{(successor case)} \\ \alpha^\beta &\stackrel{\text{def}}{=} \bigcup_{\gamma<\beta} (\alpha^\gamma) && \text{(limit case)} \end{aligned} \tag{6.2.3}$$

Cantor Normal Form. Using ordinal arithmetic, we can build all ordinals up to ε_0 (i.e. the smallest ordinal such that $\varepsilon_0 = \omega^{\varepsilon_0}$):

$$0, 1, 2, \ldots, \omega, \omega + 1, \omega + 2, \ldots, \omega \cdot 2, \omega \cdot 2 + 1, \omega \cdot 2 + 2, \ldots, \omega^2, \ldots, \omega^3, \ldots, \omega^\omega, \ldots$$

In the following, we use the representation of ordinals based on Cantor Normal Form [Kun80], i.e. every ordinal $\alpha > 0$ can be uniquely written as

$$\omega^{\beta_1} \cdot n_1 + \cdots + \omega^{\beta_k} \cdot n_k$$

where k is a natural number, the coefficients $n_1, \ldots, n_k$ are positive integers and the exponents $\beta_1 > \beta_2 > \cdots > \beta_k \geq 0$ are ordinal numbers. Throughout the rest of the thesis we will consider ordinal numbers only up to ω^ω.

6.3 Decision Trees Abstract Domain

In the previous Chapter 5 we have presented the decision trees abstract domain $\mathbb{T}(\mathbb{D}, \mathbb{C}, \mathbb{F})$, parameterized by a numerical abstract domain $\mathbb{D}$, an auxiliary abstract domain $\mathbb{C}$ for the decision nodes, and an auxiliary abstract domain $\mathbb{F}$ based on *natural-valued* functions for the leaf nodes (cf. Section 5.2). In the following, we address the limitations of $\mathbb{F}$ by means of auxiliary abstract domain $\mathbb{W}(\mathbb{F})$ based on *ordinal-valued* functions. Thus, the decision trees abstract domain $\mathbb{T}(\mathbb{D}, \mathbb{C}, \mathbb{F})$ is lifted to $\mathbb{T}(\mathbb{D}, \mathbb{C}, \mathbb{W}(\mathbb{F}))$.

6.3.1 Decision Trees

We now formally define the family of auxiliary abstract domains $\mathbb{W}$. Then, we illustrate its integration within the family of abstract domains $\mathbb{T}$.

Ordinal-Valued Functions Auxiliary Abstract Domain. The family of abstract domains $\mathbb{W}$ is a functor which lifts any auxiliary abstract domain $\mathbb{F}$ introduced in Section 5.2 to *ordinal-valued* functions. As $\mathbb{F}$, it is dedicated to the manipulation of the leaf nodes of a decision tree with respect to the linear constraints satisfied along their paths from the root of the decision tree.

Let $\mathcal{X} \stackrel{\text{def}}{=} \{X_1, \ldots, X_k\}$ be the set of program variables. The elements of these abstract domains belong to the following set:

$$\mathcal{W} \stackrel{\text{def}}{=} \{\bot_{\mathrm{W}}\} \cup \left\{ \sum_i \omega^i \cdot f_i \;\middle|\; f_i \in \mathcal{F} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\} \right\} \cup \{\top_{\mathrm{W}}\} \tag{6.3.1}$$

where $\mathcal{F}$ is defined in Equation 5.2.5, which consists of *ordinal-valued* functions of the program variables, plus the counterparts $\bot_{\mathrm{W}}$ and $\top_{\mathrm{W}}$ of $\bot_{\mathrm{F}}$ and $\top_{\mathrm{F}}$, respectively. More specifically, the ordinal-valued functions are polynomials in ω (that is, ordinals in Cantor Normal Form), where the coefficients are natural-valued functions of the program variables belonging to $\mathcal{F} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}$. The maximum degree of the polynomials, in the following denoted by M, is another parameter of the abstract domain.

The *computational order* $\sqsubseteq_{\mathrm{W}}$ and the *approximation order* $\preccurlyeq_{\mathrm{W}}$ are parameterized by a numerical abstract domain element $D \in \mathcal{D}$, which represents the linear constraints satisfied along the path to the leaf nodes. According to Lemma 4.2.22, they coincide when the compared leaf nodes are both *defined*:

$$\begin{aligned} &\sum_i \omega^i \cdot f_{i_1} \preccurlyeq_{\mathrm{W}} [D] \sum_i \omega^i \cdot f_{i_2} \iff \forall \rho \in \gamma_{\mathrm{D}}(D) : \\ &\qquad \sum_i \omega^i \cdot f_{i_1}(\rho(X_1), \ldots, \rho(X_k)) \leq \sum_i \omega^i \cdot f_{i_2}(\rho(X_1), \ldots, \rho(X_k)) \end{aligned} \tag{6.3.2}$$

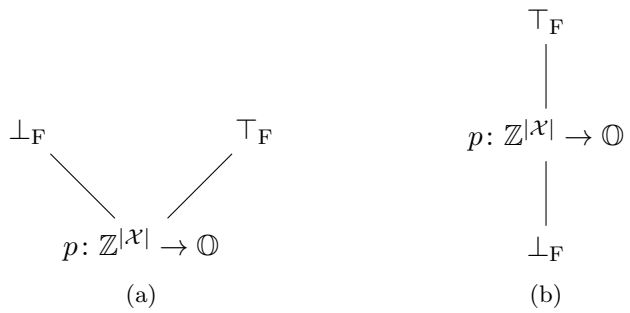


Figure 6.2: Hasse diagrams for approximation order $\preccurlyeq_{\mathrm{F}}[D]$ (a) and computational order $\sqsubseteq_{\mathrm{F}}[D]$ (b) of the ordinal-valued functions abstract domain.

$$\begin{aligned}\sum_i \omega^i \cdot f_{i_1} \sqsubseteq_{\mathrm{W}}[D] \sum_i \omega^i \cdot f_{i_2} \iff \forall \rho \in \gamma_{\mathrm{D}}(D) :\\ \sum_i \omega^i \cdot f_{i_1}(\rho(X_1), \ldots, \rho(X_k)) \leq \sum_i \omega^i \cdot f_{i_2}(\rho(X_1), \ldots, \rho(X_k))\end{aligned} \tag{6.3.3}$$

Instead, when one or both leaf nodes are undefined, the approximation and computational orders are defined by the Hasse diagrams in Figure 6.2a and Figure 6.2b, respectively. Note that, as in Figure 5.3a and Figure 5.3b, $\bot_{\mathrm{W}}$-leaves and $\top_{\mathrm{W}}$-leaves are incomparable and comparable, respectively.

We define the following concretization-based abstraction:

$$\langle \mathcal{E} \rightharpoonup \mathbb{O}, \preccurlyeq \rangle \xleftarrow{\gamma_{\mathrm{W}}[D]} \langle \mathcal{W}, \preccurlyeq_{\mathrm{W}} \rangle$$

where $D \in \mathcal{D}$ represents the path to the leaf node. The concretization function $\gamma_{\mathrm{W}} \colon \mathcal{D} \to \mathcal{W} \to (\mathcal{E} \rightharpoonup \mathbb{O})$ is defined as follows:

$$\begin{aligned}\gamma_{\mathrm{W}}[D]\bot_{\mathrm{W}} &\stackrel{\text{def}}{=} \dot{\emptyset}\\ \gamma_{\mathrm{W}}[D]\sum_i \omega^i \cdot f_i &\stackrel{\text{def}}{=} \lambda \rho \in \gamma_{\mathrm{D}}(D) : \sum_i \omega^i \cdot f_i(\rho(X_1), \ldots, \rho(X_k))\\ \gamma_{\mathrm{W}}[D]\top_{\mathrm{W}} &\stackrel{\text{def}}{=} \dot{\emptyset}\end{aligned} \tag{6.3.4}$$

As in Equation 5.2.9, $\bot_{\mathrm{W}}$ and $\top_{\mathrm{W}}$ represent the totally undefined function.

The following result proves that, given $D \in \mathcal{D}$, $\gamma_{\mathrm{W}}[D]$ is monotonic:

Lemma 6.3.1 $\forall p_1, p_2 \in \mathcal{W} : p_1 \preccurlyeq_W [D]\ p_2 \Rightarrow \gamma_W[D]p_1 \preccurlyeq \gamma_W[D]p_2$.

Proof.
See Appendix A.4. ■

Decision Trees Abstract Domain. We can now use the family of abstract domains $\mathbb{W}$ to lift the decision trees abstract domain to $\mathbb{T}(\mathbb{D}, \mathbb{C}, \mathbb{W}(\mathbb{F}))$.

Specifically, the decision trees now belong to the following set:

$$\mathcal{T} \stackrel{\text{def}}{=} \{\text{LEAF} : p \mid p \in \mathcal{W}\} \cup \{\text{NODE}\{c\} : t_1; t_2 \mid c \in \mathcal{C}, t_1, t_2 \in \mathcal{T}\} \tag{6.3.5}$$

where $\mathcal{W}$ is defined in Equation 6.3.1 and $\mathcal{C}$ is defined in Equation 5.2.1.

In the following, we lift to ordinal-valued functions all the abstract domain operators, focusing on the operators used to manipulate leaf nodes.

6.3.2 Binary Operators

The need for ordinals arising from non-deterministic boolean expressions (cf. Example 6.1.1) gets reflected into the *binary* operator for the computational and approximation join of decision tree.

Join. The join of decision trees is parameterized by the choice of the join between leaf nodes and yields a ranking function defined over the *union* of their partitions (cf. Algorithm 5). We lift Algorithm 5 to ordinal-valued functions by simply defining the join between ordinal-valued leaf nodes.

The approximation join $\curlyvee_{\mathrm{W}}$ and the computational join $\sqcup_{\mathrm{W}}$ are parameterized by a numerical abstraction $D \in \mathcal{D}$, which represents the linear constraints satisfied along the path to both leaf nodes, after the decision tree unification (cf. Line 5 of Algorithm 5). They differ when joining defined and undefined leaf nodes. Specifically, the approximation join is defined as follows:

$$\begin{aligned}
\bot_{\mathrm{W}} \curlyvee_{\mathrm{W}}[D]\, p &\stackrel{\text{def}}{=} \bot_{\mathrm{W}} && p \in \mathcal{W} \setminus \{\top_{\mathrm{W}}\} \\
p \curlyvee_{\mathrm{W}}[D]\, \bot_{\mathrm{W}} &\stackrel{\text{def}}{=} \bot_{\mathrm{W}} && p \in \mathcal{W} \setminus \{\top_{\mathrm{W}}\} \\
\top_{\mathrm{F}} \curlyvee_{\mathrm{W}}[D]\, f &\stackrel{\text{def}}{=} \top_{\mathrm{W}} && p \in \mathcal{W} \setminus \{\bot_{\mathrm{W}}\} \\
p \curlyvee_{\mathrm{W}}[D]\, \top_{\mathrm{W}} &\stackrel{\text{def}}{=} \top_{\mathrm{W}} && p \in \mathcal{W} \setminus \{\bot_{\mathrm{W}}\}
\end{aligned} \tag{6.3.6}$$

and the computational join is defined as follows:

$$\begin{aligned}
\bot_{\mathrm{W}} \sqcup_{\mathrm{W}}[D]\, p &\stackrel{\text{def}}{=} p && p \in \mathcal{W} \\
p \sqcup_{\mathrm{W}}[D]\, \bot_{\mathrm{W}} &\stackrel{\text{def}}{=} p && p \in \mathcal{W} \\
\top_{\mathrm{W}} \sqcup_{\mathrm{W}}[D]\, p &\stackrel{\text{def}}{=} \top_{\mathrm{F}} && p \in \mathcal{W} \\
p \sqcup_{\mathrm{W}}[D]\, \top_{\mathrm{W}} &\stackrel{\text{def}}{=} \top_{\mathrm{F}} && p \in \mathcal{W}
\end{aligned} \tag{6.3.7}$$

Algorithm 18 : Ordinal-Valued Function Join

```
1:  function W-JOIN(⊕, D, p1, p2)
2:                          ▷ D ∈ 𝒟, p1, p2 ∈ 𝒲 \ {⊥W, ⊤W}, ⊕ ∈ {ϒF, ⊔F}
3:      r ← Σ_i ω^i · (λX1, ..., Xk. 0)
4:      i ← 0
5:      carry ← false
6:      while i < M do          ▷ join carried out in ascending powers of ω
7:          f ← p1[i] ⊕[D] p2[i]
8:          if f = ⊤F then
9:              carry ← true
10:         else
11:             if carry then
12:                 r[i] ← STEPF(f)
13:                 carry ← false
14:             else
15:                 r[i] ← f
16:         i ← i + 1
17:     if ¬carry then
18:         return r
19:     else                    ▷ maximum degree of the polynomial exceeded
20:         return ⊤W
```

In particular, as in Equation 5.2.13, the approximation join is undefined when joining $\bot_{\mathrm{W}}$-leaves and $\top_{\mathrm{W}}$-leaves and always favors the undefined leaf nodes.

Instead, the approximation and computational join between defined leaf nodes coincide. Algorithm 18 is a generic implementation parameterized by the choice of the join $\oplus$ between natural-valued leaf nodes. Given an ordinal valued function $p \stackrel{\text{def}}{=} \sum_i \omega^i \cdot f_i \in \mathcal{W} \setminus \{\bot_{\mathrm{W}}, \top_{\mathrm{W}}\}$, we write $p[i]$ to denote the coefficient f_i. The join of two given ordinal-valued functions $p_1, p_2 \in \mathcal{W} \setminus \{\bot_{\mathrm{W}}, \top_{\mathrm{W}}\}$ is carried out in ascending powers of ω, joining the coefficients of terms with the same power of ω (cf. Line 7), up to the maximum degree M (cf. Line 6). In case the join of natural-valued leaf nodes yields $\top_{\mathrm{F}}$ (cf. Line 8), the function W-JOIN sets the coefficient to equal the zero function (cf. Line 3) and propagates a carry of one execution step (cf. Line 9 and Lines 12) to the unification of terms with next higher degree; unless the maximum degree M has been reached, in which case W-JOIN returns $\top_{\mathrm{W}}$ (cf. Line 20).

To clarify, let us consider the following example:

Algorithm 19 : Ordinal-Valued Function Approximation Join

```
1: function A-W-JOIN(D, p1, p2)          ▷ D ∈ 𝒟, p1, p2 ∈ 𝒲 \ {⊥W, ⊤W}
2:     return W-JOIN(ϒF, D, p1, p2)
```

Algorithm 20 : Ordinal-Valued Function Computational Join

```
1: function C-W-JOIN(D, p1, p2)          ▷ D ∈ 𝒟, p1, p2 ∈ 𝒲 \ {⊥W, ⊤W}
2:     return W-JOIN(⊔F, p1, p2, C)
```

Example 6.3.2

Let $\mathcal{X} \stackrel{\text{def}}{=} \{x, y\}$, let $\mathbb{D}$ be the intervals abstract domain $\langle \mathcal{B}, \sqsubseteq_{\mathrm{B}} \rangle$, and let $\mathbb{F}$ be the affine functions auxiliary abstract domain $\langle \mathcal{F}_{\mathrm{A}}, \preccurlyeq_{\mathrm{F}} \rangle$. We consider the ordinal-valued functions $p_1 \stackrel{\text{def}}{=} \omega \cdot x + y$ and $p_2 \stackrel{\text{def}}{=} \omega \cdot (x-1) - y$ and we assume that their domain of definition $D \in \mathcal{D}$ is defined as $D \stackrel{\text{def}}{=} \top_{\mathrm{B}}$. The join of p_1 and p_2 is carried out in ascending powers of ω starting from the coefficients $f_{1_0} \stackrel{\text{def}}{=} y$ and $f_{2_0} \stackrel{\text{def}}{=} -y$. However, there does not exists a *natural-valued* affine function which is the least upper bound of f_{1_0} and f_{2_0} within D. Thus, we force their join to equal zero and we carry one to the join of $f_{1_1} \stackrel{\text{def}}{=} x - 1$ and $f_{2_1} \stackrel{\text{def}}{=} x$ which becomes $x+1$ (i.e., x after the join and $x+1$ after propagating the carry). Thus, the result of the join of p_1 and p_2 is $r \stackrel{\text{def}}{=} \omega \cdot (x+1)$.

Intuitively, whenever natural-valued functions are not sufficient, W-JOIN naturally resorts to ordinal numbers: given two terms $\omega^k \cdot f_1$ and $\omega^k \cdot f_2$, forcing their join to equal zero and carrying one to the terms with next higher degree is equivalent to considering their join to be equal to ω and applying the limit case of ordinal multiplication (cf. Equation 6.2.2): $\omega^k \cdot \omega = \omega^{k+1} \cdot 1 + \omega^k \cdot 0 = \omega^{k+1}$.

The approximation join $\curlyvee_{\mathrm{W}}$ and the computational join $\sqcup_{\mathrm{W}}$ between defined leaf nodes are implemented by Algorithm 19 and Algorithm 20, respectively: the functions A-W-JOIN of Algorithm 19 and C-W-JOIN of Algorithm 20 call the function W-JOIN of Algorithm 18 choosing respectively the approximation join $\curlyvee_{\mathrm{F}}$ and the computational join $\sqcup_{\mathrm{F}}$ between natural-valued leaf nodes.

6.3.3 Unary Operators

In the following, we lift to ordinal-valued functions the *unary* operators for handling skip instructions and backward variable assignments. In particular, the assignment operator reflects the need for ordinals arising from non-deterministic assignments (cf. Example 4.1.2 and Example 6.1.1).

Algorithm 21 : Ordinal-Valued Function Assignment

```
1:  function W-ASSIGN⟦X := aexp⟧(d, D, p)
2:                                          ▷ d, D ∈ 𝒟, p ∈ 𝒲 \ {⊥_W, ⊤_W}
3:      r ← Σ_{i=1}^{M} ω^i · (λX_1, ..., X_k. 0)
4:      i ← 0
5:      carry ← false
6:      while i < M do    ▷ assignment carried out in ascending powers of ω
7:          f ← B-ASSIGN_F⟦X := aexp⟧d[D]p[i]
8:          if f = ⊤_F then
9:              carry ← true
10:         else
11:             if carry then
12:                 r[i] ← STEP_F(f)
13:                 carry ← false
14:             else
15:                 r[i] ← f
16:         i ← i + 1
17:     if ¬carry then
18:         return STEP_W(r)
19:     else                  ▷ maximum degree of the polynomial exceeded
20:         return ⊤_W
```

Skip. We lift the step operator for handling skip instructions (cf. Algorithm 9) defining the step operator $\textsc{step}_{\mathrm{W}}$ for ordinal-valued leaves.

The operator $\textsc{step}_{\mathrm{W}} \colon \mathcal{W} \to \mathcal{W}$, given an ordinal-valued function $p \in \mathcal{W} \setminus \{\bot_{\mathrm{W}}, \top_{\mathrm{W}}\}$, simply invokes the step operator $\textsc{step}_{\mathrm{F}}$ for natural-valued leaves on the constant term of the polynomial; undefined leaf nodes are unaltered:

$$\begin{aligned} \textsc{step}_{\mathrm{W}}(\bot_{\mathrm{W}}) &\stackrel{\text{def}}{=} \bot_{\mathrm{F}} \\ \textsc{step}_{\mathrm{W}}(\sum_i \omega^i \cdot f_i) &\stackrel{\text{def}}{=} \textsc{step}_{\mathrm{F}}(f_0) + \sum_{i>0} \omega^i \cdot f_i = (\sum_i \omega^i \cdot f_i) + 1 \\ \textsc{step}_{\mathrm{W}}(\top_{\mathrm{W}}) &\stackrel{\text{def}}{=} \top_{\mathrm{F}} \end{aligned} \tag{6.3.8}$$

Assignments. We now define the operator $\textsc{b-assign}_{\mathrm{W}}$ to handle backward assignments within ordinal-valued leaf nodes, in order to lift the operator $\textsc{b-assign}_{\mathrm{T}}$ (cf. Algorithm 11) to ordinal-valued functions.

The operator $\textsc{b-assign}_{\mathrm{W}}[\![X := \mathit{aexp}]\!] \colon \mathcal{D} \to \mathcal{D} \to \mathcal{W} \to \mathcal{W}$ is implemented by Algorithm 21. The assignment, given a numerical abstraction $d \in \mathcal{D}$ of

the reachable environments at the initial control point of the instruction, a numerical abstraction $D \in \mathcal{D}$ of the linear constraints accumulated along the path to the leaf node, is carried out in ascending powers of ω, invoking the B-ASSIGN$_{\mathrm{F}}$ operators on the coefficients (cf. Line 7), up to the maximum degree M (cf. Line 6). In case the assignment of natural-valued leaf nodes yields $\top_{\mathrm{F}}$ (cf. Line 8), the function W-ASSIGN lets the coefficient to equal the zero function (cf. Line 3) and carries one execution step (cf. Line 9 and Line 12) to the term with next higher degree; unless the maximum degree M has been reached, in which case W-ASSIGN returns $\top_{\mathrm{W}}$ (cf. Line 20). The operator STEP$_{\mathrm{W}}$ is invoked before returning (cf. Line 18) to take into account that one more program execution step is needed before termination.

To clarify, let us consider the following example:

Example 6.3.3

Let $\mathcal{X} \stackrel{\text{def}}{=} \{x, y\}$, let $\mathbb{D}$ be the intervals abstract domain $\langle \mathcal{B}, \sqsubseteq_{\mathrm{B}} \rangle$, and let $\mathbb{F}$ be the affine functions auxiliary abstract domain $\langle \mathcal{F}_{\mathrm{A}}, \preccurlyeq_{\mathrm{F}} \rangle$. We consider the ordinal-valued functions $p \stackrel{\text{def}}{=} \omega \cdot x + y$ and the non-deterministic assignment $x := ?$. We assume that the domain of definition $D \in \mathcal{D}$ and $d \in \mathcal{B}$ are defined as $D \stackrel{\text{def}}{=} \top_{\mathrm{B}}$ and $d \stackrel{\text{def}}{=} \top_{\mathrm{B}}$. The assignment is carried out in ascending powers of ω starting from the coefficient $f_0 \stackrel{\text{def}}{=} y$, which remains unchanged since the assignment only involves the variable x, whereas the coefficient $f_1 \stackrel{\text{def}}{=} x$ is reset to zero and carries one to the term with next higher degree ω^2. In fact, the non-deterministic assignment $x := ?$ allows x (and consequently f_1) to take *any* value, but there does not exists a *natural-valued* affine function that properly abstracts all possible outcomes of the assignment. The resulting ordinal-valued function, after the invocation of STEP$_{\mathrm{W}}$, is $r \stackrel{\text{def}}{=} \omega^2 \cdot 1 + (y + 1)$.

6.3.4 Widening

The widening operator, unlike the join and assignment operators, is not allowed to introduce ordinals of higher degree to avoid missing cases like case B and case C in Figure 5.10b, where the value of the abstract ranking function increases between iterates. We lift Algorithm 13 to ordinal-valued functions defining an extrapolation operator $\blacktriangledown_{\mathrm{W}}$ between ordinal-valued leaf nodes.

The extrapolation operator $\blacktriangledown_{\mathrm{W}}[D_1, D_2]$ is implemented by Algorithm 22. The extrapolation, given the numerical abstractions $D_1 \in \mathcal{D}$ and $D_2 \in \mathcal{D}$ representing the path to the leaf nodes from the root of the decision tree, is carried out in ascending powers of ω invoking the extrapolation operator $\blacktriangledown_{\mathrm{F}}[D_1, D_2]$ on the coefficients (cf. Line 6), up to the maximum degree M (cf.

Algorithm 22 : Ordinal-Valued Function Extrapolation

1: **function** W-WIDEN(D_1, D_2, p_1, p_2)
2: $\triangleright$ $D_1, D_2 \in \mathcal{D}$, $p_1, p_2 \in \mathcal{W} \setminus \{\bot_{\mathrm{W}}, \top_{\mathrm{W}}\}$
3: $r \leftarrow \sum_{i=1}^{M} \omega^i \cdot (\lambda X_1, \ldots, X_k.\ 0)$
4: $i \leftarrow 0$
5: **while** $i < M$ **do** $\triangleright$ widening carried out in ascending powers of ω
6: $f \leftarrow p_1[i] \blacktriangledown_{\mathrm{F}} [D_1, D_2]\ p_2[i]$
7: **if** $f = \top_{\mathrm{F}}$ **then**
8: **return** $\top_{\mathrm{W}}$
9: **else**
10: $r[i] \leftarrow f$
11: $i \leftarrow i + 1$
12: **return** r

Line 5). In case the extrapolation of natural leaf nodes yields $\top_{\mathrm{F}}$ (cf. Line 7), the function W-WIDEN returns $\top_{\mathrm{W}}$ (cf. Line 8).

6.4 Abstract Definite Termination Semantics

The lifted operators of the decision trees abstract domain can now lift to ordinal-valued functions the abstract termination semantics of a program instruction *stmt*, defined in Figure 5.12, and the abstract termination semantics of a program *prog*, defined in Definition 5.3.1.

The following result proves, for program instruction *stmt*, the soundness of the abstract termination semantics $\tau^{\natural}_{\mathrm{Mt}}[\![\, stmt \,]\!]$ with respect to the termination semantics $\tau_{\mathrm{Mt}}[\![\, stmt \,]\!]$ defined in Section 4.3, given sound over-approximations $R \in \mathcal{D}$ of $\tau_{\mathrm{I}}(i[\![\, stmt \,]\!])$ and $D \in \mathcal{D}$ of $\tau_{\mathrm{I}}(f[\![\, stmt \,]\!])$:

Lemma 6.4.1 $\tau_{Mt}[\![\, stmt \,]\!](\gamma_T[D]t) \preccurlyeq \gamma_T[R](\tau^{\natural}_{Mt}[\![\, stmt \,]\!]t)$.

Proof.
See Appendix A.4. ■

Similarly, the following result proves the soundness of the abstract termination semantics $\tau^{\natural}_{\mathrm{Mt}}[\![\, prog \,]\!] \in \mathcal{T}$ with respect to the termination semantics $\tau_{\mathrm{Mt}}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$, given a sound numerical over-approximation $R \in \mathcal{D}$ of the reachable environments $\tau_{\mathrm{I}}(i[\![\, prog \,]\!])$ at the initial program control point:

Theorem 6.4.2 $\tau_{Mt}[\![prog]\!] \preccurlyeq \gamma_T[R]\tau^{\natural}_{Mt}[\![prog]\!]$

Proof (Sketch).
The proof follows from the soundness of the operators of the decision trees abstract domain (cf. Lemma 6.4.1) used for the definition of $\tau^{\natural}_{\mathrm{Mt}}[\![prog]\!] \in \mathcal{T}$.■

In particular, the abstract termination semantics provides *sufficient preconditions* for ensuring definite termination of a program for a given over-approximation $R \in \mathcal{D}$ of the set of initial states $\mathcal{I} \subseteq \Sigma$:

Corollary 6.4.3 *A program must terminate for execution traces starting from a given set of initial states* $\gamma_D(R)$ *if* $\gamma_D(R) \subseteq \mathrm{dom}(\gamma_T[R]\tau^{\natural}_{Mt}[\![prog]\!])$.

Examples. In the following, we recall the example introduced at the beginning of the chapter and we describe in some detail the analysis of the program using the abstract domain of decision trees. Then, we propose further examples to illustrate the expressiveness of the abstract domain.

Example 6.4.4
Let us consider again the program from [CPR11]:

```
while 1(0 < x ∧ 0 < y) do
  if 2( ? ) then
    3x := x − 1
    4y := ?
  else
    5y := y − 1
  fi
od6
```

We present the analysis of the program using interval constraints based on the *intervals abstract domain* (cf. Section 3.4.1) for the decision nodes, and *ordinal-valued functions* for the leaf nodes (cf. Equation 6.3.1).

The starting point is the zero function at the program final control point:

6 : LEAF : $\lambda x.\lambda y.\ 0$

The ranking function is then propagated backwards towards the program initial control iterating through the `while` loop. We use a widening delay of three iterations. At the fourth iteration, the decision tree at program control point **1** represents the following piecewise-defined ranking function:

$$\lambda\rho. \begin{cases} 1 & \rho(x) \leq 0 \vee \rho(y) \leq 0 \\ 3y+2 & \rho(x) = 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $3y + 2$ is the result of the widening between adjacent leaf nodes with consecutive values for y (i.e., between LEAF : $\lambda x.\lambda y.\ 5$ within $\rho(x) = 1 \wedge \rho(y) = 1$ and LEAF : $\lambda x.\lambda y.\ 8$ within $\rho(x) = 1 \wedge \rho(y) = 2$).

Ordinals appear for the first time at program control point 4 due to the non-deterministic assignment to y:

$$\lambda\rho. \begin{cases} 2 & \rho(x) \leq 0 \\ \omega + 9 & \rho(x) = 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

In the first case the value of the function is simply increased to count one more program step before termination, but (since y can now have any value) its domain is modified forgetting all constraints on y (i.e. $y \leq 0$). In the second case, $3y + 2$ becomes $\omega + 9$ due to the non-deterministic assignment.

At the seventh iteration, the decision tree associated with program control point 1, represents the following ranking function:

$$\lambda\rho. \begin{cases} 1 & \rho(x) \leq 0 \vee \rho(y) \leq 0 \\ 3y+2 & \rho(x) = 1 \\ \omega + (3y+9) & \rho(x) = 2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $\omega + (3y + 9)$ is the result of the widening between LEAF : $\lambda x.\lambda y.\ \omega + 12$ within $\rho(x) = 2 \wedge \rho(y) = 1$ and LEAF : $\lambda x.\lambda y.\ \omega + 15$ within $\rho(x) = 2 \wedge \rho(y) = 2$. The widening is carried out in ascending powers of ω: from the constants 12 and 15, the widening infers the value $3y + 9$; then, since the coefficients of ω are equal to one, the inferred coefficient is again one. Thus, the result of the widening is $\omega + (3x + 9)$ within $\rho(x) = 2$.

Finally, at the eleventh iteration, the analysis reaches a fixpoint:

$$\lambda\rho. \begin{cases} 1 & \rho(x) \leq 0 \vee \rho(y) \leq 0 \\ 3y+2 & \rho(x) = 1 \\ \omega + (3y+9) & \rho(x) = 2 \\ \omega \cdot (x-1) + (7x+3y-5) & \text{otherwise} \end{cases}$$

where $3y + 2$ and $\omega + (3y + 9)$ are particular cases (within $\rho(x) = 1$ and $\rho(x) = 2$, respectively) of $\omega \cdot (x - 1) + (7x + 3y - 5)$ and are explicitly listed only due to the amount of widening delay we used. The ranking function proves that the program is always terminating, whatever the initial values for x and y, and whatever the non-deterministic choices during execution. The reason why we obtain a different and more complex ranking function with respect to Figure 6.1 is because we count the number of program execution steps whereas, for convenience of presentation, in Figure 6.1 we simply count the number of loop iterations.

Example 6.4.5

Let us consider the following program:

```
while 1(x ≠ 0 ∧ 0 < y) do
  if 2(0 < x) then
    if 3( ? ) then
      4x := x − 1
      5y := ?
    else
      6y := y − 1
    fi
  else
    if 7( ? ) then
      8x := x + 1
    else
      9y := y − 1
      10x := ?
    fi
  fi
od11
```

which is an involved variation of Example 6.1.1. Each loop iteration, when x is positive, either decrements the value of x, or decrements the value of x and resets the value of y; when x is negative, either increments the value of x, or decrements the value of y and resets the value of x to any value (possibly positive). The loop exits when x is equal to zero or y is less than zero.

Note that, there does not exist a lexicographic ranking function for the loop. In fact, the variables x and y can be alternatively reset to any value at each loop iteration: the value of y is reset at the program control point **5**, while the value of x is reset at the control point **10**.

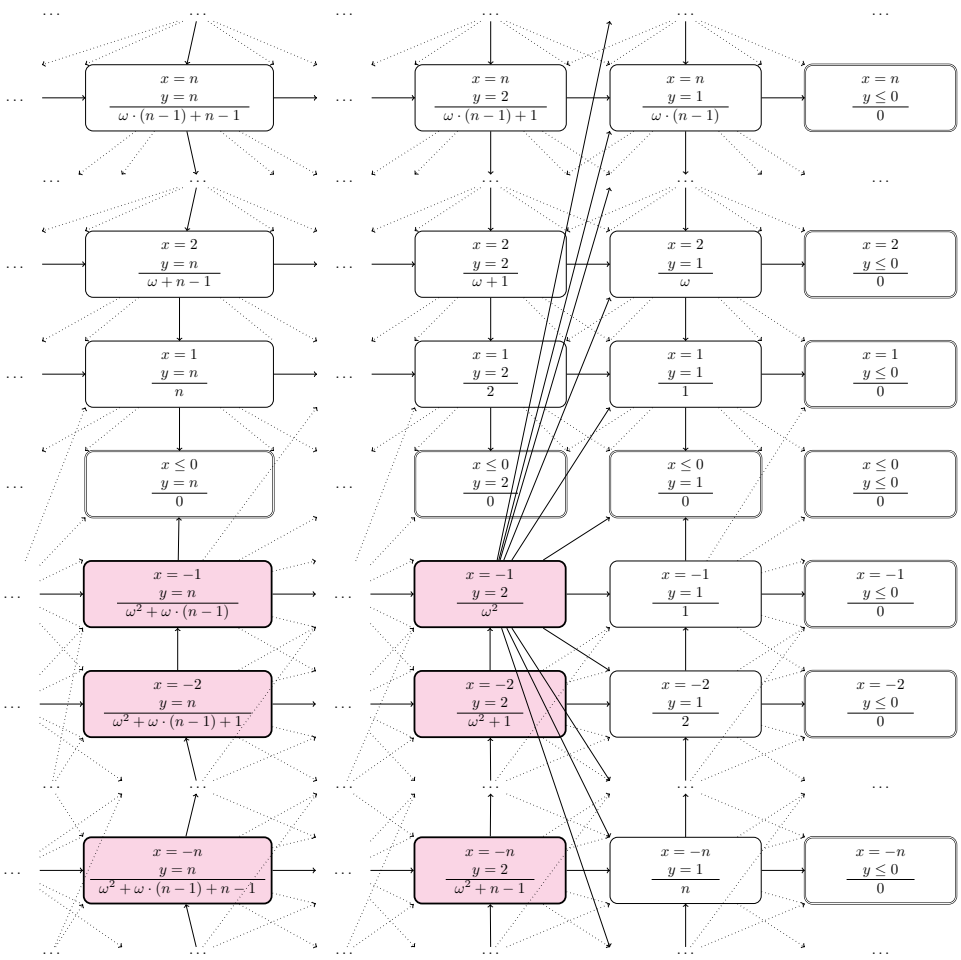


Figure 6.3: Transitions between states at control point **1** for the program in Figure 6.4.5. There is an edge from any node where x has value $k > 0$ (and $y > 0$) to all nodes where x has value $k - 1$ (and y has any value); there is also an edge from any node where y has value $h > 0$ (and $x < 0$) to all nodes where y has value $h - 1$ (and x has any value). In every node we indicate the maximum number of loop iterations needed to reach a blocking state: the highlighted nodes require ordinals greater than ω^2.

Nonetheless, the program always terminates, regardless of the initial values for x and y, and regardless of the non-deterministic choices taken during execution. Let us consider the graph in Figure 6.3. Whenever y is reset to any value, we move towards the blocking states decreasing the value of x, and whenever x is reset to any value, we move towards the blocking states decreasing the value of y. Moreover, whenever x is reset to a positive value, its value will only decrease until it reaches zero (or y is reset to a value less than zero).

The analysis of the program using interval constraints based on the *intervals abstract domain* (cf. Section 3.4.1) for the decision nodes, *ordinal-valued functions* for the leaf nodes (cf. Equation 6.3.1), yields the following piecewise-defined ranking function at program control point **1**:

$$\lambda\rho. \begin{cases} \omega^2 + \omega \cdot (y-1) + (-4x + 9y - 2) & \rho(x) \leq 0 \vee 0 < \rho(y) \\ 1 & \rho(x) = 0 \vee y \leq 0 \\ \omega \cdot (x-1) + (9x + 4y - 7) & 0 < \rho(x) \vee 0 \leq \rho(y) \end{cases}$$

In Figure 6.3, we justify the need for ω^2. Indeed, from any state where $x < 0$ and $y = h > 0$, whenever x is reset at program control point **10**, it is possible to jump to any state where $y = h - 1$. In particular, for example from the state where $x = -1$ and $y = 2$, it is possible to jump through unbounded non-determinism to states with value of the most precise ranking function equal to an arbitrary ordinal number between ω and ω^2, which requires ω^2 as upper bound of the maximum number of loop iterations needed to reach a final state.

Finally, note the expressions identified as coefficients of ω: where $x < 0$, the coefficient of ω is an expression in y (since y guides the progress towards the blocking states), and where $0 < x$, the coefficient of ω is an expression in x (because x rules the progress towards termination). The expressions are automatically inferred by the analysis without any assistance from the user.

6.5 Related Work

Interestingly, ordinal-valued ranking functions already appeared in the work of Alan Turing in the late 1940s [Tur49, MJ84]. To the best of our knowledge, the automatic *inference* of ordinal-valued ranking functions for proving termination of *imperative* programs is unique to our work.

The approach presented in this chapter is mostly related to [ADFG10]: both techniques handle programs with arbitrary structure and infer ranking functions (that also provide information on the program computational complexity in terms of executions steps) attached to program control points. In

[ADFG10], lexicographic ranking functions are obtained by a greedy algorithm based on Farkas lemma that, analogously to the operators of our abstract domain, constructs the ranking functions by adding one dimension at a time. However, the method proposed in [ADFG10], although complete for programs with rational-valued variables, is incomplete for integer-valued variables. In contrast, there is no completeness limitation to our method, only a choice of relevant abstract domains. In [GMR15], the authors improve over [ADFG10] and present an SMT-based method for the inference of lexicographic ranking functions. However, both [ADFG10] and [GMR15] remain focused only on proving termination for all program input, while our work naturally deals with proving conditional termination as well.

In a different context, a large amount of research followed the introduction of size-change termination [LJBA01]. The size-change termination approach consists in collecting a set of size-change graphs (representing function calls) and combining them into multipaths (representing program executions) in such a way that at least one variable is guaranteed to decrease. Compared to size-change termination, our approach avoids the exploration of the combinatorial space of multipaths with the explicit manipulation of ordinals. In [Lee09, BAL09], algorithms are provided to derive explicit ranking functions from size-change graphs, but these ranking functions have a shape quite different from ours which makes it difficult for us to compare their expressiveness. For example, the derived ranking functions use lexicographic orders on variables while our polynomial coefficients are arbitrary linear combinations of variables. In general, an in-depth comparison between such fairly different methods is an open research topic (e.g., see [HJP10] for the comparison of the transition invariants and the size-change termination methods).

Finally, we have seen that there exist programs for which there does not exist a lexicographic ranking function (cf. Example 6.4.5). In [CSZ13] the authors discuss the problem and propose some heuristics to circumvent it. Interestingly these heuristics rediscover exactly the need for piecewise-defined ranking functions, even if implicitly and in a roundabout way.

7

Recursive Programs

In the small programming language introduced in Chapter 3 only loops present a challenge when proving termination. In this chapter, we illustrate a simple extension of the language with recursive procedures. We revisit the definition of its maximal trace semantics (cf. Section 3.2) and its definite termination semantics (cf. Section 4.3). Moreover, we propose a sound decidable abstraction for proving termination of recursive programs based the piecewise-defined ranking functions introduced in Chapter 5 and Chapter 6.

Dans le petit langage de programmation introduit dans le Chapitre 3 seules les boucles présentent un défi pour prouver la terminaison. Dans ce chapitre, nous montrons une simple extension du language avec des procédures récursives. Nous reviendrons sur la définition de sa sémantique de traces maximales (cf. Section 3.2) et sa sémantique de terminaison (cf. Section 4.3). De plus, nous proposons une abstraction décidable correcte pour prouver la terminaison des programmes récursifs basée sur les fonctions de rang définies par morceaux introduites au Chapitre 5 et au Chapitre 6.

7.1 A Small Procedural Language

In Figure 7.1, the syntax of our programming language proposed in Figure 3.1 is extended with possibly recursive procedures.

A program *prog* consists of a procedure declaration followed by a unique label $l \in \mathcal{L}$. A procedure declaration *mthd* is either a *main* procedure declaration or a sequential composition of declarations. The *main* procedure is always declared last. A procedure consists in an instruction statement labelled by a unique procedure name $M \in \mathcal{M} \cup \{main\}$.

The set of language instructions is extended with `call` and `return` statements: a `call` statement branches to the initial control point of the called

$$
\begin{array}{lcll}
\mathit{stmt} & ::= & \ldots & \\
 & | & {}^{l}\texttt{call}\ M & M \in \mathcal{M} \\
\\
\mathit{ret} & ::= & {}^{l}\texttt{return} & \\
\\
\mathit{mthd} & ::= & \mathit{main}\colon\ \mathit{stmt} & \\
 & | & M\colon\ \mathit{stmt}\ \mathit{ret}\ \mathit{mthd} & M \in \mathcal{M} \\
\\
\mathit{prog} & ::= & \mathit{mthd}\ {}^{l} & l \in \mathcal{L}
\end{array}
$$

Figure 7.1: Syntax of our programming language extended with procedures.

procedure; the `return` statement branches back to the control point after the call statement. With the exception of the main procedure, the last instruction of all procedures is a `return` statement. For simplicity, we assume that any mutual recursion is reduced into a single recursive procedure [KRP93].

Note that, this extension of the language allows us to also encode programs containing functions with arguments and return values, thanks to variables. Therefore, we do not explicitly introduce such features in the language.

7.2 Maximal Trace Semantics

We now define the transition semantics of our extended language. In particular, since a procedure might be called during the execution of another procedure, we introduce a *stack* to recover the right calling control point: the procedure `call` statement pushes the calling control point onto the stack, whereas the procedure `return` statement pops the last calling control point from the top of the stack and branches to this control point. Then, we define the maximal trace semantics of the language by induction on its extended syntax.

Transition Systems. A *stack* $k \in \mathcal{L}^*$ is a possibly empty sequence of program control points. Let $\mathcal{K}$ denote the set of all stacks.

The set of all program states $\Sigma \stackrel{\text{def}}{=} \mathcal{L} \times \mathcal{E}$ is extended to pairs $\mathcal{K} \times \Sigma$ consisting of a stack $k \in \mathcal{K}$ and a program state $s \in \Sigma$, which in turn consists of a program control point $l \in \mathcal{L}$ paired with an environment $\rho \in \mathcal{E}$.

In Figure 7.2 we define the *initial* control point of a program *prog*, a `call` instruction *stmt* and a `return` instruction *ret*. The initial control point of a program *prog* is the initial control point of the *main* procedure. The initial

$$
\begin{array}{lcll}
\mathit{stmt} & ::= & \ldots & \\
 & | & {}^{l}\texttt{call}\ M & i[\![\, {}^{l}\texttt{call}\ M \,]\!] \stackrel{\text{def}}{=} l \\
\mathit{ret} & ::= & {}^{l}\texttt{return} & i[\![\, {}^{l}\texttt{return} \,]\!] \stackrel{\text{def}}{=} l \\
\mathit{mthd} & ::= & \mathit{main}\colon\ \mathit{stmt} & i[\![\, \mathit{main} : \mathit{stmt} \,]\!] \stackrel{\text{def}}{=} i[\![\, \mathit{stmt} \,]\!] \\
 & | & M\colon\ \mathit{stmt}\ \mathit{ret}\ \mathit{mthd}_1 & i[\![\, M : \mathit{stmt}\ \mathit{ret}\ \mathit{mthd}_1 \,]\!] \stackrel{\text{def}}{=} i[\![\, \mathit{mthd}_1 \,]\!] \\
\mathit{prog} & ::= & \mathit{mthd}\ {}^{l} & i[\![\, \mathit{mthd}\ {}^{l} \,]\!] \stackrel{\text{def}}{=} i[\![\, \mathit{mthd} \,]\!]
\end{array}
$$

Figure 7.2: Initial control point of *stmt*, *ret*, and *prog*.

$$
\begin{array}{lcll}
\mathit{stmt} & ::= & \ldots & \\
 & | & {}^{l}\texttt{call}\ M & f[\![\, {}^{l}\texttt{call}\ M \,]\!] \stackrel{\text{def}}{=} f[\![\, \mathit{stmt} \,]\!] \\
\mathit{prog} & ::= & \mathit{mthd}\ {}^{l} & f[\![\, \mathit{mthd}\ {}^{l} \,]\!] \stackrel{\text{def}}{=} l
\end{array}
$$

Figure 7.3: Final control point of *stmt* and *prog*.

control point of other instructions *stmt* is unchanged from Figure 3.4. Similarly, in Figure 7.3 we define the *final* control point of a program *prog* and a `call` instruction *stmt*. The final control point $f[\![\, {}^{l}\texttt{return} \,]\!]$ of a `return` instruction *ret* cannot be defined statically; it is a dynamic information depending on the calling control point on the stack during execution. The final control point of other instructions *stmt* is unchanged from Figure 3.5.

In the following, the function $i\colon \mathcal{M} \to \mathcal{L}$ maps procedure names to their initial control point: given a declaration $M : \mathit{stmt}\ \mathit{ret}$, $i(M) \stackrel{\text{def}}{=} i[\![\, \mathit{stmt} \,]\!]$.

The set of initial states is extended to $\{\varepsilon\} \times \mathcal{I} \stackrel{\text{def}}{=} \{\langle \varepsilon, \langle i[\![\, \mathit{prog} \,]\!], \rho \rangle\rangle \mid \rho \in \mathcal{E}\}$ and the set of final states is extended to $\{\varepsilon\} \times \mathcal{Q} \stackrel{\text{def}}{=} \{\langle \varepsilon, \langle f[\![\, \mathit{prog} \,]\!], \rho \rangle\rangle \mid \rho \in \mathcal{E}\}$.

We now redefine the transition relation $\tau \in (\mathcal{K} \times \Sigma) \times (\mathcal{K} \times \Sigma)$. In particular, in Figure 7.4, we define the transition semantics $\tau[\![\, \mathit{stmt} \,]\!] \in (\mathcal{K} \times \Sigma) \times (\mathcal{K} \times \Sigma)$ and $\tau[\![\, \mathit{ret} \,]\!] \in (\mathcal{K} \times \Sigma) \times (\mathcal{K} \times \Sigma)$ of a `call` instruction and a `return` instruction, respectively. The semantics of other instructions *stmt* is defined analogously to Figure 3.6, extending the states with a stack, which is left unchanged by the instruction. The execution of a `call` instruction pushes the final control point of the instruction onto the stack and branches to the initial control point of the called procedure. The function $\tau\colon \mathcal{M} \to \mathcal{L}$ maps procedure names to their transition semantics: given a declaration $M : \mathit{stmt}\ \mathit{ret}$, $\tau(M) \stackrel{\text{def}}{=} \tau[\![\, \mathit{stmt} \,]\!] \cup \tau[\![\, \mathit{ret} \,]\!]$. The execution of a `return` instruction branches to the control point popped from the top of the stack.

$$\tau[\![\,^l call\ M\,]\!] \stackrel{\text{def}}{=} \{\langle k, \langle l, \rho\rangle\rangle \to \langle f[\![\,^l call\ M\,]\!] \cdot k, \langle i(M), \rho\rangle\rangle \mid k \in \mathcal{K}, \rho \in \mathcal{E}\} \cup \tau(M)$$
$$\tau[\![\,^l return]\!] \stackrel{\text{def}}{=} \{\langle r \cdot k, \langle l, \rho\rangle\rangle \to \langle k, \langle r, \rho\rangle\rangle \mid r \in \mathcal{L},\ k \in \mathcal{K},\ \rho \in \mathcal{E}\}$$

Figure 7.4: Transition semantics of *stmt* and *ret*.

The transition relation $\tau \in (\mathcal{K} \times \Sigma) \times (\mathcal{K} \times \Sigma)$ of a program *prog* is defined by the semantics $\tau[\![\,prog]\!] \in (\mathcal{K} \times \Sigma) \times (\mathcal{K} \times \Sigma)$ of the program as follows:

$$\tau[\![\,prog]\!] = \tau[\![\,mthd\,^l\,]\!] \stackrel{\text{def}}{=} \tau[\![\,mthd]\!]$$

where $\tau[\![\,main : stmt]\!] \stackrel{\text{def}}{=} \tau[\![\,stmt]\!]$ and $\tau[\![\,M : stmt\ ret\ mthd]\!] \stackrel{\text{def}}{=} \tau[\![\,mthd]\!]$. In other words, the semantics $\tau[\![\,prog]\!]$ of a program *prog* is defined by the semantics $\tau[\![\,main : stmt]\!] \in (\mathcal{K} \times \Sigma) \times (\mathcal{K} \times \Sigma)$ of the *main* procedure.

Example 7.2.1

Let us consider the recursive version of the program of Example 4.1.2:

```
f :
    if 1(1 < x) then
      2x := x − 1
      3call f
    else
      4skip
    fi
    5return

main :
    6x := ?
    7call f
    8
```

The set of program environments $\mathcal{E}$ contains the functions $\rho\colon \{x\} \to \mathbb{Z}$ mapping the program variable x to any possible value $\rho(x) \in \mathbb{Z}$. The set of program states $\Sigma \stackrel{\text{def}}{=} \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8}\} \times \mathcal{E}$ consists of all pairs of numerical labels and environments; the extended initial states are $\{\varepsilon\} \times \mathcal{I} \stackrel{\text{def}}{=} \{\langle \varepsilon, \langle \mathbf{6}, \rho\rangle\rangle \mid \rho \in \mathcal{E}\}$. The program transition relation $\tau \in (\mathcal{K} \times \Sigma) \times (\mathcal{K} \times \Sigma)$ is defined as follows:

$$\tau \stackrel{\text{def}}{=} \{\langle \varepsilon, \langle \mathbf{6}, \rho\rangle\rangle \to \langle \varepsilon, \langle \mathbf{7}, \rho[x \leftarrow v]\rangle\rangle \mid \rho \in \mathcal{E} \wedge v \in \mathbb{Z}\}$$
$$\cup \{\langle \varepsilon, \langle \mathbf{7}, \rho\rangle\rangle \to \langle \mathbf{8} \cdot \varepsilon, \langle \mathbf{1}, \rho\rangle\rangle \mid \rho \in \mathcal{E}\}$$

$$
\begin{aligned}
&\cup \{\langle k, \langle \mathbf{1}, \rho\rangle\rangle \to \langle k, \langle \mathbf{2}, \rho\rangle\rangle \mid k \in \mathcal{K},\ \rho \in \mathcal{E} \wedge \text{true} \in [\![1 < x]\!]\rho\}\\
&\cup \{\langle k, \langle \mathbf{2}, \rho\rangle\rangle \to \langle k, \langle \mathbf{3}, \rho[x \leftarrow \rho(x) - 1]\rangle\rangle \mid k \in \mathcal{K},\ \rho \in \mathcal{E}\}\\
&\cup \{\langle k, \langle \mathbf{3}, \rho\rangle\rangle \to \langle \mathbf{5} \cdot k, \langle \mathbf{1}, \rho\rangle\rangle \mid k \in \mathcal{K},\ \rho \in \mathcal{E}\}\\
&\cup \{\langle k, \langle \mathbf{1}, \rho\rangle\rangle \to \langle k, \langle \mathbf{4}, \rho\rangle\rangle \mid k \in \mathcal{K},\ \rho \in \mathcal{E} \wedge \text{false} \in [\![1 < x]\!]\rho\}\\
&\cup \{\langle k, \langle \mathbf{4}, \rho\rangle\rangle \to \langle k, \langle \mathbf{5}, \rho\rangle\rangle \mid k \in \mathcal{K},\ \rho \in \mathcal{E}\}\\
&\cup \{\langle \mathbf{5} \cdot k, \langle \mathbf{5}, \rho\rangle\rangle \to \langle k, \langle \mathbf{5}, \rho\rangle\rangle \mid k \in \mathcal{K},\ \rho \in \mathcal{E}\}\\
&\cup \{\langle \mathbf{8} \cdot \varepsilon, \langle \mathbf{5}, \rho\rangle\rangle \to \langle \varepsilon, \langle \mathbf{8}, \rho\rangle\rangle \mid \rho \in \mathcal{E}\}
\end{aligned}
$$

The extended set of final states is $\{\varepsilon\} \times \mathcal{Q} \stackrel{\text{def}}{=} \{\langle \varepsilon, \langle \mathbf{8}, \rho\rangle\rangle \mid \rho \in \mathcal{E}\}$.

Maximal Trace Semantics. The maximal trace semantics $\tau^{+\infty} \in \mathcal{P}((\mathcal{K} \times \Sigma)^{+\infty})$ of our extended language is generated by the extended transition system $\langle \mathcal{K} \times \Sigma, \tau\rangle$ as in Section 3.2. In particular, the following result restates Theorem 2.2.12 and defines the maximal trace semantics in fixpoint form:

Theorem 7.2.2 (Maximal Trace Semantics) *The* maximal trace semantics $\tau^{+\infty} \in \mathcal{P}((\mathcal{K} \times \Sigma)^{+\infty})$ *can be expressed as a least fixpoint in the complete lattice* $\langle \mathcal{P}((\mathcal{K} \times \Sigma)^{+\infty}), \sqsubseteq, \sqcup, \sqcap, (\mathcal{K} \times \Sigma)^{\omega}, (\mathcal{K} \times \Sigma)^{+}\rangle$ *as follows:*

$$
\begin{aligned}
\tau^{+\infty} &= \text{lfp}^{\sqsubseteq}\ \phi^{+\infty}\\
\phi^{+\infty}(T) &\stackrel{def}{=} (\{\varepsilon\} \times \mathcal{Q}) \cup (\tau\ ;\ T)
\end{aligned}
\tag{7.2.1}
$$

In the following, we provide a structural definition of the maximal trace semantics by induction on the extended syntax of programs.

Remark 7.2.3 From now on, we assume that procedures have a single recursive call. We plan to generalize the framework as part of our future work. ■

We do not explicitly represent the stack. Instead, we use least fixpoints as denotations of recursive procedures. Thus, we have $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$.

In Figure 7.5, for any procedure P, we define the semantics $\tau^{+\infty}[\![\, stmt \,]\!]_P S \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+\infty})$ and $\tau^{+\infty}[\![\, ret \,]\!]_P S \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+\infty})$ of a `call` instruction and a `return` instruction, respectively. The traces are built backwards: each function $\tau^{+\infty}[\![\, stmt \,]\!]_P S$ (resp. $\tau^{+\infty}[\![\, ret \,]\!]_P S$) takes as input a set of traces starting with the final label of the instruction *stmt* (resp. *ret*) and outputs a set of traces starting with the initial label of *stmt* (resp. *ret*). The parameter set of traces $S \in \mathcal{P}(\Sigma^{+\infty})$ is used to handle recursive calls.

The trace semantics of a *recursive* `call` instruction, when the called procedure coincides with the caller, is parameterized by a set of traces S representing

$$\tau^{+\infty}[\![\, {}^{l}\texttt{call}\ M \,]\!]_M S(T) \stackrel{\text{def}}{=} \left\{ \langle l, \rho\rangle\langle i(M), \rho\rangle\sigma \;\middle|\; \begin{array}{c} \rho \in \mathcal{E},\ \sigma \in \Sigma^{*\infty} \\ \langle i(M), \rho\rangle\sigma \in S\ ;\ T \end{array} \right\}$$

$$\tau^{+\infty}[\![\, {}^{l}\texttt{call}\ M \,]\!]_P S(T) \stackrel{\text{def}}{=} \left\{ \langle l, \rho\rangle\langle i(M), \rho\rangle\sigma \;\middle|\; \begin{array}{c} \rho \in \mathcal{E},\ \sigma \in \Sigma^{*\infty} \\ \langle i(M), \rho\rangle\sigma \in C \end{array} \right\} \qquad P \neq M$$

$$C \stackrel{\text{def}}{=} \operatorname{lfp}^{\sqsubseteq}_{\Sigma^\omega}(\lambda S.\ \tau^{+\infty}(M)S(T))$$

$$\tau^{+\infty}[\![\, {}^{l}\texttt{return} \,]\!]_P S(T) \stackrel{\text{def}}{=} \{\langle l, \rho\rangle\langle r, \rho\rangle\sigma \mid r \in \mathcal{L},\ \rho \in \mathcal{E},\ \sigma \in \Sigma^{*\infty},\ \langle r, \rho\rangle\sigma \in T\}$$

Figure 7.5: Maximal trace semantics of instructions *stmt* and *ret*.

the trace semantics of the *following* recursive calls, followed by a set of traces T starting with the final label of the instruction. The traces belonging to $S\ ;\ T$ start with environments paired with the initial control point of the procedure. Thus, the trace semantics of the instruction simply prepends to them the same environments paired with the initial label of the instruction.

The trace semantics of a non-recursive `call` instruction, when the called procedure M is different from the caller P, takes as input a set of traces T starting with the final label of the instruction, determines from T the trace semantics C of the callee, and prepends to the initial states of the traces belonging to C the same environments paired with the initial label of the instruction. The called procedure M can be a recursive procedure. Thus, its trace semantics is defined as the least fixpoint of $\lambda S.\ \tau^{+\infty}(M)S(T)$ within the complete lattice $\langle \mathcal{P}(\Sigma^{+\infty}), \sqsubseteq, \sqcup, \sqcap, \Sigma^\omega, \Sigma^+\rangle$, analogously to Equation 2.2.5. Note that, we assumed that mutual recursion is reduced into a single recursive procedure. The function $\tau^{+\infty} \colon \mathcal{M} \to (\mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+\infty}))$ maps procedure names to their maximal trace semantics: given a declaration M : *stmt ret*, $\tau^{+\infty}(M)S(T) \stackrel{\text{def}}{=} \tau^{+\infty}[\![\, \mathit{stmt} \,]\!]_M S(\tau^{+\infty}[\![\, \mathit{ret} \,]\!]_M S(T))$. The iteration sequence, starting from all infinite *sequences* Σ^ω, builds the set of program *traces* that consist of an infinite number of recursive calls, and it prepends a finite number of recursive calls to an input set $T \in \mathcal{P}(\Sigma^{+\infty})$ of traces starting with the final label of the `call` instruction.

Finally, the semantics of a `return` instruction takes as input a set of traces starting from the calling control point, and prepends to them the same initial environments paired with the initial control point of the instruction.

The trace semantics $\tau^{+\infty}[\![\, \mathit{stmt} \,]\!]_P S \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+\infty})$ of other instructions *stmt* is defined as follows:

$$\tau^{+\infty}[\![\, \mathit{stmt} \,]\!]_P S \stackrel{\text{def}}{=} \tau^{+\infty}[\![\, \mathit{stmt} \,]\!] \tag{7.2.2}$$

where $\tau^{+\infty}[\![\, \mathit{stmt} \,]\!] \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+\infty})$ is defined in Figure 3.7.

The maximal trace semantics $\tau^{+\infty}[\![\, prog \,]\!] \in \mathcal{P}(\Sigma^{+\infty})$ of a program *prog* is defined by the maximal trace semantics of the *main* procedure taking as input the infinite sequences Σ^{ω} and the final states $\mathcal{Q}$:

Definition 7.2.4 (Maximal Trace Semantics) *The* maximal trace semantics $\tau^{+\infty}[\![\, prog \,]\!] \in \mathcal{P}(\Sigma^{+\infty})$ *of a program prog is:*

$$\tau^{+\infty}[\![\, prog \,]\!] = \tau^{+\infty}[\![\, mthd^{\,l} \,]\!] \stackrel{def}{=} \tau^{+\infty}[\![\, mthd \,]\!]\Sigma^{\omega}(\mathcal{Q}) \tag{7.2.3}$$

where $\tau^{+\infty}[\![\, main : stmt \,]\!] \stackrel{def}{=} \tau^{+\infty}[\![\, stmt \,]\!]_{main}$ *and* $\tau^{+\infty}[\![\, M : stmt\ ret\ mthd \,]\!] \stackrel{def}{=} \tau^{+\infty}[\![\, mthd \,]\!]$*, and where the semantics* $\tau^{+\infty}[\![\, stmt \,]\!]_P S \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+\infty})$ *of each program instruction stmt is defined in Figure 7.5.*

Example 7.2.5

Let us consider the recursive version of the program of Example 4.2.3:

```
f :
   if 1( ? ) then
     2skip
   else
     3call f
   fi
   4return

main :
   5call f
   6
```

The set of program environments $\mathcal{E}$ only contains the totally undefined function $\rho \colon \emptyset \to \mathbb{Z}$, since the program has no variables. The set of all program states is $\Sigma \stackrel{\text{def}}{=} \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}\} \times \mathcal{E}$, while the set of initial states is $\mathcal{I} \stackrel{\text{def}}{=} \{\mathbf{5}\} \times \mathcal{E}$, and the set of final states is $\mathcal{Q} \stackrel{\text{def}}{=} \{\mathbf{6}\} \times \mathcal{E}$.

The maximal trace semantics of the program is defined by the maximal trace semantics of the *main* procedure: $\tau^{+\infty}_{main}[\![\, {}^{\mathbf{5}}\texttt{call}\ f \,]\!]\Sigma^{\omega}(\mathcal{Q})$ (cf. Definition 3.2.5). The semantics of the `call` statement determines the maximal trace semantics of the procedure f (cf. Figure 7.5). The fixpoint iterates are depicted in Figure 7.6. Thus, the maximal trace semantics of the program contains the traces that starting from the initial states $\mathcal{I}$ call the procedure f a non-determinist number of times: $\{\langle \mathbf{5}, \rho\rangle(\langle \mathbf{1}, \rho\rangle\langle \mathbf{3}, \rho\rangle)^{*}\langle \mathbf{1}, \rho\rangle\langle \mathbf{2}, \rho\rangle\langle \mathbf{4}, \rho\rangle\langle \mathbf{8}, \rho\rangle \mid \rho \in \mathcal{E}\} \cup \{\langle \mathbf{5}, \rho\rangle(\langle \mathbf{1}, \rho\rangle\langle \mathbf{3}, \rho\rangle)^{\omega} \mid \rho \in \mathcal{E}\}$

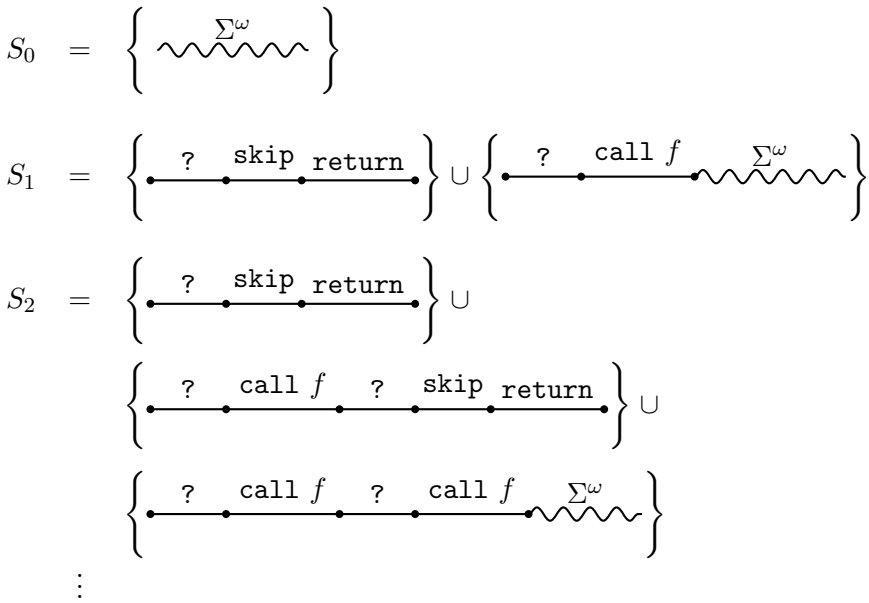


Figure 7.6: Fixpoint iterates of the maximal trace semantics for Example 7.2.5.

7.3 Definite Termination Semantics

We now extend the structural definition given in Section 4.3 of the fixpoint definite termination semantics $\tau_{\mathrm{Mt}} \in \Sigma \rightharpoonup \mathbb{O}$ (cf. Equation 4.2.16). Then, we propose a sound decidable abstraction of τ_{Mt} based on piecewise-defined ranking functions (cf. Chapter 5 and Chapter 6).

7.3.1 Definite Termination Semantics

We partition the definite termination semantics τ_{Mt} with respect to the program control points: $\tau_{\mathrm{Mt}} \in \mathcal{L} \to (\mathcal{E} \rightharpoonup \mathbb{O})$. In this way, to each program control point $l \in \mathcal{L}$ corresponds a partial function $f \colon \mathcal{E} \rightharpoonup \mathbb{O}$, and, for any procedure P, to each program instruction *stmt* and *ret* corresponds a termination semantics $\tau_{\mathrm{Mt}}[\![\, \mathit{stmt} \,]\!]_P \colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ and $\tau_{\mathrm{Mt}}[\![\, \mathit{ret} \,]\!]_P \colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$.

Each function $\tau_{\mathrm{Mt}}[\![\, \mathit{stmt} \,]\!]_P F$ and $\tau_{\mathrm{Mt}}[\![\, \mathit{ret} \,]\!]_P F$ takes as input a ranking function whose domain represents the terminating environments at the *final* label of the instruction and outputs a ranking function whose domain represents the terminating environments at the *initial* label of the instruction, and whose value is an upper bound on the number of program execution steps remaining to termination. The parameter $F \in \mathcal{E} \rightharpoonup \mathbb{O}$ is used to handle recursive calls.

The termination semantics of a *recursive* `call`, when the called procedure coincides with the caller, is parameterized by a ranking function $F\colon \mathcal{E} \rightharpoonup \mathbb{O}$ representing the termination semantics of the *following* recursive calls, and takes as input a ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain represents the terminating environments at the *final* label of the instruction. The domain of termination semantics of the instruction is the intersection of the domains of f and F, and its value is the sum of the value of f and F plus one, to take into account that from the environments at the initial label of the instruction another program execution step is necessary before termination:

$$(\tau_{\mathrm{Mt}}[\![\, {}^{l}\texttt{call}\ M \,]\!]_M F)(f) \stackrel{\text{def}}{=} \lambda\rho \in \mathrm{dom}(f) \cap \mathrm{dom}(F).\ f(\rho) + F(\rho) + 1 \tag{7.3.1}$$

The termination semantics of a non-recursive `call`, when the called procedure M is different from the caller P, takes as input a ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$, determines from f the termination semantics $I\colon \mathcal{E} \rightharpoonup \mathbb{O}$ of the callee, and increases its value to account of another program execution step:

$$\begin{aligned} (\tau_{\mathrm{Mt}}[\![\, {}^{l}\texttt{call}\ M \,]\!]_P F)(f) &\stackrel{\text{def}}{=} \lambda\rho \in \mathrm{dom}(I).\ I(\rho) + 1 \qquad P \neq M \\ I &\stackrel{\text{def}}{=} \mathrm{lfp}^{\sqsubseteq}_{\dot{\emptyset}}(\lambda F.\ (\tau_{\mathrm{Mt}}(M)F)(f)) \end{aligned} \tag{7.3.2}$$

The termination semantics of the possibly recursive called procedure M is defined as the least fixpoint of $\lambda F.\ (\tau_{\mathrm{Mt}}(M)F)(f)$ within the partially ordered set $\langle \mathcal{E} \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$, analogously to Equation 4.2.16. The rationale being that the call from P to M may be a call to a sequence of recursive calls to M, which are collected by the fixpoint. The function $\tau_{\mathrm{Mt}}\colon \mathcal{M} \to ((\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O}))$ maps procedure names to their termination semantics: given a procedure declaration $M : \mathit{stmt}\ \mathit{ret}$, we have $(\tau_{\mathrm{Mt}}(M)F)(f) \stackrel{\text{def}}{=} (\tau_{\mathrm{Mt}}[\![\, \mathit{stmt} \,]\!]_M F)((\tau_{\mathrm{Mt}}[\![\, \mathit{ret} \,]\!]_M F)(f))$. The iteration sequence, starting from the totally undefined function $\dot{\emptyset}$, builds the ranking function whose domain is the set of environments from which a finite number of recursive calls is made.

Finally, the termination semantics of a `return` instruction takes as input a ranking function whose domain represents the terminating environments at the calling control point and increases its value to take into account another program execution step before termination:

$$(\tau_{\mathrm{Mt}}[\![\, {}^{l}\texttt{return} \,]\!]_P F)(f) \stackrel{\text{def}}{=} \lambda\rho \in \mathrm{dom}(f).\ f(\rho) + 1 \tag{7.3.3}$$

The termination semantics $\tau_{\mathrm{Mt}}[\![\, \mathit{stmt} \,]\!]_P F\colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ of other instructions *stmt* is defined as follows:

$$\tau_{\mathrm{Mt}}[\![\, \mathit{stmt} \,]\!]_P F \stackrel{\text{def}}{=} \tau_{\mathrm{Mt}}[\![\, \mathit{stmt} \,]\!] \tag{7.3.4}$$

where $\tau_{\mathrm{Mt}}[\![\, stmt \,]\!] \colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ is defined in Section 4.3. For a conditional statement `if` ^{l}*bexp* `then` $stmt_1$ `else` $stmt_2$ `fi`, F has to be passed to the termination semantics of $stmt_1$ and $stmt_2$. For a loop `while` ^{l}*bexp* `do` *stmt* `od`, F has to be passed to the termination semantics of the loop body *stmt*. For a sequential composition of instructions $stmt_1$ $stmt_2$, F has to be passed to the termination semantics of the components $stmt_1$ and $stmt_2$.

The termination semantics $\tau_{\mathrm{Mt}}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$ of a program *prog* is a ranking function whose domain represents the terminating environments, which is determined by the termination semantics of the *main* procedure taking as input the totally undefined function and the zero function:

Definition 7.3.1 (Termination Semantics) *The* termination semantics $\tau_{Mt}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$ *of a program prog is:*

$$\tau_{Mt}[\![\, prog \,]\!] = \tau_{Mt}[\![\, mthd\ ^{l} \,]\!] \stackrel{def}{=} (\tau_{Mt}[\![\, mthd \,]\!]\dot{\emptyset})(\lambda\rho.\ 0) \tag{7.3.5}$$

where $\tau_{Mt}[\![\, main : stmt \,]\!] \stackrel{def}{=} \tau_{Mt}[\![\, stmt \,]\!]_{main}$ *and* $\tau_{Mt}[\![\, M : stmt\ ret\ mthd \,]\!] \stackrel{def}{=} \tau_{Mt}[\![\, mthd \,]\!]$*, and where the function* $\tau_{Mt}[\![\, stmt \,]\!]_P F \colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ *is the termination semantics of each program instruction stmt.*

7.3.2 Abstract Definite Termination Semantics

We propose a sound decidable abstraction of the definite termination semantics $\tau_{\mathrm{Mt}}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$ based on piecewise-defined ranking functions.

In particular, we supplement the set of operators of the decision trees abstract domain (cf. Chapter 5 and Chapter 6) with a sum operator $+_{\mathrm{T}}$.

Sum. The *sum* operator $+_{\mathrm{T}}$ is implemented by Algorithm 23: the function SUM, given a sound over-approximation $D \in \mathcal{D}$ of the reachable environments and two decision trees $t_1, t_2 \in \mathcal{T}$, first calls UNIFICATION (cf. Line 10) for tree unification and then calls the auxiliary function SUM-AUX (cf. Line 11). The latter descends along the paths of the decision trees (cf. Lines 5-6), up to the leaf nodes where the leaves sum operator $+_{\mathrm{F}}$ is invoked.

The sum $+_{\mathrm{F}}$ between defined and undefined leaf nodes is analogous to the leaves approximation join $\curlyvee_{\mathrm{F}}$ (cf. Equation 5.2.13):

$$\begin{aligned}
\bot_{\mathrm{F}} +_{\mathrm{F}} f &\stackrel{\mathrm{def}}{=} \bot_{\mathrm{F}} && f \in \mathcal{F} \setminus \{\top_{\mathrm{F}}\} \\
f +_{\mathrm{F}} \bot_{\mathrm{F}} &\stackrel{\mathrm{def}}{=} \bot_{\mathrm{F}} && f \in \mathcal{F} \setminus \{\top_{\mathrm{F}}\} \\
\top_{\mathrm{F}} +_{\mathrm{F}} f &\stackrel{\mathrm{def}}{=} \top_{\mathrm{F}} && f \in \mathcal{F} \setminus \{\bot_{\mathrm{F}}\} \\
f +_{\mathrm{F}} \top_{\mathrm{F}} &\stackrel{\mathrm{def}}{=} \top_{\mathrm{F}} && f \in \mathcal{F} \setminus \{\bot_{\mathrm{F}}\}
\end{aligned} \tag{7.3.6}$$

Algorithm 23 : Tree Sum

```
1:  function SUM-AUX(t1, t2, C)                                  ▷ t1, t2 ∈ 𝒯
2:      if ISLEAF(t1) ∧ ISLEAF(t2) then
3:          return LEAF : t1.f +F t2.f
4:      else if ISNODE(t1) ∧ ISNODE(t2) then
5:          (l1, l2) ← SUM-AUX(t1.l, t2.l)
6:          (r1, r2) ← SUM-AUX(t1.r, t2.r)
7:          return (NODE{c} : l1; r1, NODE{c} : l2; r2)
8:
9:  function SUM(D, t1, t2)                                      ▷ D ∈ 𝒟, t1, t2 ∈ 𝒯
10:     (t1, t2) ← UNIFICATION(D, t1, t2)
11:     return SUM-AUX(t1, t2)
```

$$
\begin{aligned}
(\tau^{\natural}_{\mathrm{Mt}}[\![\, {}^{l}\texttt{call}\ M \,]\!]_M T)(t) &\stackrel{\text{def}}{=} \mathrm{STEP}_{\mathrm{T}}(t +_{\mathrm{T}} T) \\
(\tau^{\natural}_{\mathrm{Mt}}[\![\, {}^{l}\texttt{call}\ M \,]\!]_P T)(t) &\stackrel{\text{def}}{=} \mathrm{STEP}_{\mathrm{T}}(I) \qquad\qquad P \neq M \\
I &\stackrel{\text{def}}{=} \mathrm{lfp}^{\natural}\ (\lambda T.\ (\tau^{\natural}_{\mathrm{Mt}}(M)T)(t)) \\
(\tau^{\natural}_{\mathrm{Mt}}[\![\, {}^{l}\texttt{return} \,]\!]_P T)(t) &\stackrel{\text{def}}{=} \mathrm{STEP}_{\mathrm{T}}(t)
\end{aligned}
$$

Figure 7.7: Abstract termination semantics of instructions *stmt*.

In particular, the sum is undefined between $\bot_{\mathrm{F}}$-leaves and $\top_{\mathrm{F}}$-leaves and always favors the undefined leaf nodes.

Instead, given two *defined* leaf nodes $f_1, f_2 \in \mathcal{F} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}$, their sum $f_1 +_{\mathrm{F}} f_2$ is defined as expected:

$$f_1 +_{\mathrm{F}} f_2 \stackrel{\text{def}}{=} f_1 + f_2 \tag{7.3.7}$$

Example 7.3.2

Let $\mathcal{X} = \{x\}$, and let $f_1 \stackrel{\text{def}}{=} \lambda x.\ x$ and $f_2 \stackrel{\text{def}}{=} \lambda x.\ -x+5$ be two affine functions. Then, their sum is the affine function $f_1 +_{\mathrm{F}} f_2 \stackrel{\text{def}}{=} \lambda x.\ x - x + 5 = \lambda x.\ 5$.

Abstract Definite Termination Semantics. In the following, as in Section 5.3, we assume to have, for each program control point $l \in \mathcal{L}$, a sound numerical over-approximation $R \in \mathcal{D}$ of the reachable environments $\tau_{\mathrm{I}}(l) \in \mathcal{P}(\mathcal{E})$: $\tau_{\mathrm{I}}(l) \subseteq \gamma_{\mathrm{D}}(R)$ (cf. Section 3.4).

In Figure 7.7, for any procedure P, we define $\tau^{\natural}_{\mathrm{Mt}}[\![\, \mathit{stmt} \,]\!]_P T \colon \mathcal{T} \to \mathcal{T}$ and $\tau^{\natural}_{\mathrm{Mt}}[\![\, \mathit{ret} \,]\!]_P T \colon \mathcal{T} \to \mathcal{T}$ for a `call` instruction *stmt* and a `return` instruction *ret*, respectively. Each function $\tau^{\natural}_{\mathrm{Mt}}[\![\, \mathit{stmt} \,]\!]_P$ and $\tau^{\natural}_{\mathrm{Mt}}[\![\, \mathit{ret} \,]\!]_P$ takes as input a

decision tree over-approximating the ranking function corresponding to the final control point of the instruction, and outputs a decision tree defined over a subset of the reachable environments $R \in \mathcal{D}$, which over-approximates the ranking function corresponding to the initial control point of the instruction.

The parameter $T \in \mathcal{T}$ is used to handle *recursive* `call`s. In particular, the decision tree T abstracts the termination semantics $F \colon \mathcal{E} \rightharpoonup \mathbb{O}$ of the *following* recursive calls (cf. Equation 7.3.1). Then, the semantics of a recursive `call` instruction invokes $\textsc{step}_{\mathrm{T}}$ (cf. Algorithm 9) on the sum $+_{\mathrm{T}}$ (cf. Algorithm 23) between T and the input decision tree $t \in \mathcal{T}$.

The following result proves that for a recursive `call` instruction ${}^{l}\texttt{call}\ M$, given sound over-approximations $R \in \mathcal{D}$ of $\tau_{\mathrm{I}}(l)$, $S \in \mathcal{D}$ of $\tau_{\mathrm{I}}(i(M))$, and $D \in \mathcal{D}$ of $\tau_{\mathrm{I}}(f[\![\,{}^{l}\texttt{call}\ M\,]\!])$, the abstract semantics $\tau^{\natural}_{\mathrm{Mt}}[\![\,{}^{l}\texttt{call}\ M\,]\!]_M$ is a sound over-approximation of $\tau_{\mathrm{Mt}}[\![\,{}^{l}\texttt{call}\ M\,]\!]_M$ defined in Equation 7.3.1:

Lemma 7.3.3 $(\tau_{Mt}[\![\,{}^{l}\texttt{call}\ M\,]\!]_M \gamma_T[S]T)(\gamma_T[D]t) \preccurlyeq \gamma_T[R](\textsc{step}_T(t +_T T))$.

Proof.
See Appendix A.5. ∎

The semantics of a non-recursive `call`, when the called procedure M is different from the caller P, invokes the $\textsc{step}_{\mathrm{T}}$ operator on the semantics $I \in \mathcal{T}$ of the callee. The semantics of the called procedure M is defined as the limit of the iteration sequence with widening (cf. Equation 5.2.24) of $\lambda T.\ \tau^{\natural}_{\mathrm{Mt}}(M)T(t)$. The function $\tau^{\natural}_{\mathrm{Mt}} \colon \mathcal{M} \to (\mathcal{T} \to \mathcal{T} \to \mathcal{T})$ maps procedure names to their semantics: given a procedure declaration $M : \mathit{stmt}\ \mathit{ret}$, we have $\tau^{\natural}_{\mathrm{Mt}}(M)T(t) \stackrel{\mathrm{def}}{=} \tau^{\natural}_{\mathrm{Mt}}[\![\,\mathit{stmt}\,]\!]_M T(\tau^{\natural}_{\mathrm{Mt}}[\![\,\mathit{ret}\,]\!]_M T(t))$.

The abstract semantics $\tau^{\natural}_{\mathrm{Mt}}[\![\,{}^{l}\texttt{call}\ M\,]\!]_P$ of a non-recursive `call` instruction ${}^{l}\texttt{call}\ M$, given sound over-approximations $R \in \mathcal{D}$ of $\tau_{\mathrm{I}}(l)$, $S \in \mathcal{D}$ of $\tau_{\mathrm{I}}(i(M))$, and $D \in \mathcal{D}$ of $\tau_{\mathrm{I}}(f[\![\,{}^{l}\texttt{call}\ M\,]\!])$, is a sound over-approximation of the termination semantics $\tau_{\mathrm{Mt}}[\![\,{}^{l}\texttt{call}\ M\,]\!]_P$ defined in Equation 7.3.2:

Lemma 7.3.4 $(\tau_{Mt}[\![\,{}^{l}\texttt{call}\ M\,]\!]_P \gamma_T[S]T)(\gamma_T[D]t) \preccurlyeq \gamma_T[R](\textsc{step}_T(I))$, *where* $I \stackrel{def}{=} \mathrm{lfp}^{\natural}\ (\lambda T.\ (\tau^{\natural}_{Mt}(M)T)(t))$.

Proof.
See Appendix A.5. ∎

Finally, the semantics of a `return` instruction simply invokes the $\textsc{step}_{\mathrm{T}}$ operator on the given input decision tree.

The following result proves, given sound over-approximations $R \in \mathcal{D}$ of $\tau_{\mathrm{I}}(l)$ and $D \in \mathcal{D}$ of $\tau_{\mathrm{I}}(f[\![\, ^l\texttt{return}\,]\!])$, the soundness of the abstract semantics $\tau^{\natural}_{\mathrm{Mt}}[\![\, ^l\texttt{return}\,]\!]_P$ with respect to $\tau_{\mathrm{Mt}}[\![\, ^l\texttt{return}\,]\!]_P$ defined in Equation 7.3.3:

Lemma 7.3.5 $(\tau_{Mt}[\![\, ^l\texttt{return}\,]\!]_P\gamma_T[D]T)(\gamma_T[D]t) \preccurlyeq \gamma_T[R](\textsc{step}_T(t))$.

Proof.
See Appendix A.5. ■

The semantics $\tau^{\natural}_{\mathrm{Mt}}[\![\, stmt\,]\!]_P T \colon \mathcal{T} \to \mathcal{T}$ of other instructions *stmt* is:

$$\tau^{\natural}_{\mathrm{Mt}}[\![\, stmt\,]\!]_P T \stackrel{\text{def}}{=} \tau^{\natural}_{\mathrm{Mt}}[\![\, stmt\,]\!] \tag{7.3.8}$$

where $\tau^{\natural}_{\mathrm{Mt}}[\![\, stmt\,]\!] \colon \mathcal{T} \to \mathcal{T}$ is defined in Figure 5.12. For a conditional statement $\texttt{if}\ ^l bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi}$, T has to be passed to the termination semantics of $stmt_1$ and $stmt_2$. For a loop $\texttt{while}\ ^l bexp\ \texttt{do}\ stmt\ \texttt{od}$, T has to be passed to the termination semantics of the loop body *stmt*. For a sequential composition of instructions $stmt_1\ stmt_2$, T has to be passed to the termination semantics of the components $stmt_1$ and $stmt_2$. Given sound over-approximations $R \in \mathcal{D}$ of $\tau_{\mathrm{I}}(i[\![\, stmt\,]\!])$ and $D \in \mathcal{D}$ of $\tau_{\mathrm{I}}(f[\![\, stmt\,]\!])$, $\tau^{\natural}_{\mathrm{Mt}}[\![\, stmt\,]\!]_P$ is a sound over-approximation of $\tau_{\mathrm{Mt}}[\![\, stmt\,]\!]_P$ defined in Equation 7.3.4:

Lemma 7.3.6 $(\tau_{Mt}[\![\, stmt\,]\!]_P\gamma_T[D]T)(\gamma_T[D]t) \preccurlyeq \gamma_T[R]((\tau^{\natural}_{Mt}[\![\, stmt\,]\!]_P T)(t))$.

Proof (Sketch).
The proof follows from Equation 7.3.4 and Equation 7.3.8 and from the soundness of $\tau^{\natural}_{\mathrm{Mt}}[\![\, stmt\,]\!]$ defined in Figure 5.12 with respect to $\tau_{\mathrm{Mt}}[\![\, stmt\,]\!]$ defined in Section 4.3 (cf. Lemma 5.2.8, Lemma 5.2.14, Lemma 5.2.15, Lemma 5.2.16, Lemma 5.2.24, and Lemma 6.4.1). ■

The abstract termination semantics $\tau^{\natural}_{\mathrm{Mt}}[\![\, prog\,]\!] \in \mathcal{T}$ of a program *prog* outputs the decision tree over-approximating the ranking function corresponding to the initial program control point $i[\![\, prog\,]\!] \in \mathcal{L}$. It is defined by means of the abstract termination semantics of the *main* procedure taking as input $\bot_{\mathrm{T}}$ and the leaf node $\textsc{leaf} : \lambda X_1, \ldots, X_k.\ 0$:

Definition 7.3.7 (Abstract Termination Semantics) *The* abstract termination semantics $\tau^{\natural}_{Mt}[\![\, prog\,]\!] \in \mathcal{T}$ *of a program prog is:*

$$\tau^{\natural}_{Mt}[\![\, prog\,]\!] = \tau^{\natural}_{Mt}[\![\, mthd\ ^l\,]\!] \stackrel{def}{=} (\tau^{\natural}_{Mt}[\![\, mthd\,]\!]\bot_T)(\textsc{leaf} : \lambda X_1, \ldots, X_k.\ 0) \tag{7.3.9}$$

where $\tau^{\natural}_{Mt}[\![\, main : stmt \,]\!] \stackrel{def}{=} \tau^{\natural}_{Mt}[\![\, stmt \,]\!]_{main}$ *and* $\tau^{\natural}_{Mt}[\![\, M : stmt\ ret\ mthd \,]\!] \stackrel{def}{=} \tau^{\natural}_{Mt}[\![\, mthd \,]\!]$*, and where the abstract semantics* $\tau^{\natural}_{Mt}[\![\, stmt \,]\!]_P T \colon \mathcal{T} \to \mathcal{T}$ *of each program instruction stmt is defined in Figure 7.7 and Equation 7.3.8.*

The following result proves the soundness of the abstract termination semantics $\tau^{\natural}_{\mathrm{Mt}}[\![\, prog \,]\!] \in \mathcal{T}$ with respect to the termination semantics $\tau_{\mathrm{Mt}}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$, given a sound numerical over-approximation $R \in \mathcal{D}$ of the reachable environments $\tau_{\mathrm{I}}(i[\![\, prog \,]\!])$ at the initial program control point:

Theorem 7.3.8 $\tau_{Mt}[\![\, prog \,]\!] \preccurlyeq \gamma_T[R]\tau^{\natural}_{Mt}[\![\, prog \,]\!]$

Proof (Sketch).
The proof follows from the soundness of the operators of the decision trees abstract domain (Lemma 7.3.3, Lemma 7.3.4, Lemma 7.3.5, and Lemma 7.3.6) used for the definition of $\tau^{\natural}_{\mathrm{Mt}}[\![\, prog \,]\!] \in \mathcal{T}$. ■

In particular, the abstract termination semantics provides *sufficient preconditions* for ensuring definite termination of a program for a given over-approximation $R \in \mathcal{D}$ of the set of initial states $\mathcal{I} \subseteq \Sigma$:

Corollary 7.3.9 *A program must terminate for execution traces starting from a given set of initial states* $\gamma_D(R)$ *if* $\gamma_D(R) \subseteq \mathrm{dom}(\gamma_T[R]\tau^{\natural}_{Mt}[\![\, prog \,]\!])$.

Example. In the following, we recall the example introduced at the beginning of the chapter and we present the fully detailed analysis of the program using the abstract domain of decision trees.

Example 7.3.10
Let us consider again the recursive program of Example 7.2.1:

```
f :
    if [1](1 < x) then
      [2]x := x − 1
      [3]call f
    else
      [4]skip
    fi
    [5]return

main :
    [6]x := ?
    [7]call f
    [8]
```

We present the analysis of the program using interval constraints based on the *intervals abstract domain* (cf. Section 3.4.1) for the decision nodes, and *ordinal-valued* functions for the leaf nodes (cf. Chapter 6).

The starting point is the zero function at the program final control point:

8 : $\quad \text{LEAF} : \lambda x.\ 0$

The ranking function is then propagated backwards towards the program initial control point taking into account the `call` to the recursive procedure *f*.

At program control point **5** the semantics of the `return` instruction simply increases the value of the ranking function:

5 : $\quad \text{LEAF} : \lambda x.\ 1.$

The semantics of the `skip` instruction does the same at program point **4**:

4 : $\quad \text{LEAF} : \lambda x.\ 2.$

Instead, the *recursive* `call` at program control point **3** returns:

3 : $\quad \text{LEAF} : \bot_{\text{F}}$

which is propagated by the variable assignment at program control point **2**:

2 : $\quad \text{LEAF} : \bot_{\text{F}}.$

Thus, the first iterate of the `call` to f is able to conclude that the procedure terminates in at most three execution steps when $1 < x$ is not satisfied:

1 : $\quad \text{NODE}\{x - 2 \geq 0\} : (\text{LEAF} : \bot_{\text{F}}); (\text{LEAF} : \lambda x.\ 3)$

Then, during the second iterate, the *recursive* `call` at program control point **3** increases the value of the ranking function obtained after the first iterate (plus the ranking function obtained at program control point **5**):

3 : $\quad \text{NODE}\{x - 2 \geq 0\} : (\text{LEAF} : \bot_{\text{F}}); (\text{LEAF} : \lambda x.\ 5)$

and, at program control point **2**, the operator ASSIGN_{T} replaces the program variable x with the expression $x - 1$ within the decision tree:

2 : $\quad \text{NODE}\{x - 3 \geq 0\} : (\text{LEAF} : \bot_{\text{F}}); (\text{LEAF} : \lambda x.\ 6).$

Thus, the second iterate of the `call` to f is able to conclude that the procedure terminates in at most seven execution steps, when it calls itself recursively only once, and in at most three program steps when $1 < x$ is not satisfied:

$$
\begin{array}{ll}
\mathbf{1}: & \textsc{node}\{x-3 \geq 0\}: \\
& \quad \textsc{leaf}: \bot_{\mathrm{F}}; \\
& \quad \textsc{node}\{x-2 \geq 0\}: (\textsc{leaf}: \lambda x.\ 7); (\textsc{leaf}: \lambda x.\ 3)
\end{array}
$$

Similarly, the third iterate of the `call` to f concludes that the procedure terminates in at most eleven execution steps, when it calls itself recursively twice, in at most seven execution steps, when it calls itself recursively only once, and in at most three program steps when $1 < x$ is not satisfied:

$$
\begin{array}{ll}
\mathbf{1}: & \textsc{node}\{x-4 \geq 0\}: \\
& \quad \textsc{leaf}: \bot_{\mathrm{F}}; \\
& \quad \textsc{node}\{x-3 \geq 0\}: \\
& \qquad \textsc{leaf}: \textsc{leaf}: \lambda x.\ 11; \\
& \qquad \textsc{node}\{x-2 \geq 0\}: (\textsc{leaf}: \lambda x.\ 7); (\textsc{leaf}: \lambda x.\ 3)
\end{array}
$$

Then, the widening extrapolates the ranking function on the partitions over which it is not yet defined:

$$
\begin{array}{ll}
\mathbf{1}: & \textsc{node}\{x-3 \geq 0\}: \\
& \quad \textsc{leaf}: \lambda x.\ 4x-1; \\
& \quad \textsc{node}\{x-2 \geq 0\}: (\textsc{leaf}: \lambda x.\ 7); (\textsc{leaf}: \lambda x.\ 3)
\end{array}
$$

yielding a fixpoint for the `call` to f within the *main* procedure.

The ranking function associated with program control point **7** is:

$$
\begin{array}{ll}
\mathbf{7}: & \textsc{node}\{x-3 \geq 0\}: \\
& \quad \textsc{leaf}: \lambda x.\ 4x; \\
& \quad \textsc{node}\{x-2 \geq 0\}: (\textsc{leaf}: \lambda x.\ 8); (\textsc{leaf}: \lambda x.\ 4)
\end{array}
$$

Finally, the non-deterministic assignment at program control point **6** yields:

$$
\mathbf{6}: \quad \textsc{leaf}: \lambda x.\ \omega + 8.
$$

which proves that the program is always terminating, whatever the initial value of the program variable x.

The results presented in this chapter are only preliminary, more general results being necessary to cover all practical cases.

8

Implementation

This chapter presents our prototype static analyzer FuncTion which is based on piecewise-defined ranking functions. We also propose the most recent experimental evaluation.

Ce chapitre présente notre prototype d'analyseur statique FuncTion *qui est basé sur des fonctions de rang définies par morceaux. Nous présentons aussi l'évaluation expérimentale la plus récente.*

8.1 FuncTion

We have implemented a prototype static analyzer FuncTion based on the decision trees abstract domain presented in Chapter 5. It is available online: http://www.di.ens.fr/~urban/FuncTion.html.

The prototype accepts programs written in a (subset of) C, without `struct` and `union` types. It provides only a limited support for arrays and pointers. The only basic data type are *mathematical integers*, deviating from the standard semantics of C. The prototype is written in OCaml and, at the time of writing, the available numerical abstractions for handling linear constraints within the decision nodes are based on the intervals abstract domain [CC76] (cf. Section 3.4.1), the convex polyhedra abstract domain [CH78] (cf. Section 3.4.2), and the octagons abstract domain [Min06] (cf. Section 3.4.3), and the available abstraction for handling functions within the leaf nodes are based on affine functions. The numerical abstract domains are provided by the *APRON* library [JM09]. It is also possible to activate the extension to ordinal-valued ranking functions presented in Chapter 6, and tune the precision of the analysis by adjusting the widening delay.

To improve precision, we avoid trying to compute a ranking function for the non-reachable states: FuncTion runs a forward analysis to over-approximate

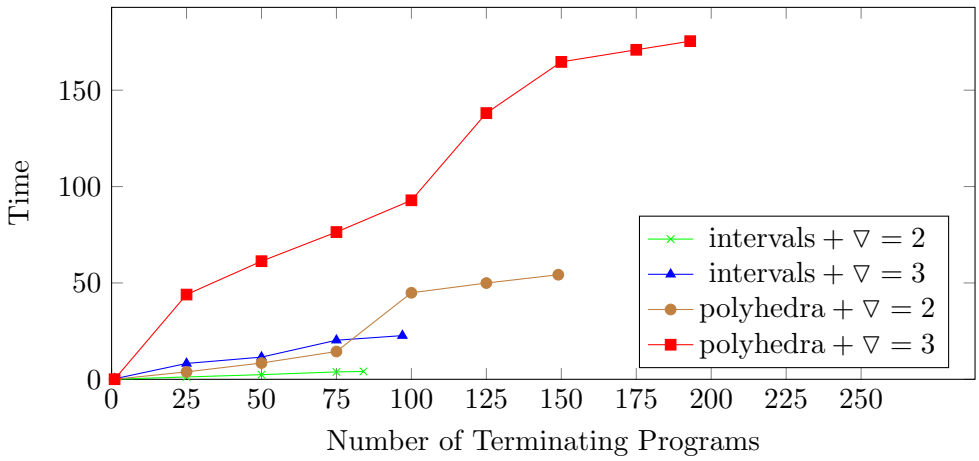


Figure 8.1: Comparison of different parameterizations of FuncTion.

the reachable states using a numerical abstract domain (cf. Section 3.4). Then, it runs the backward analysis to infer a ranking function, intersecting its domain at each step with the states identified by the previous analysis.

The analysis proceeds by structural induction on the program syntax, iterating loops and recursive procedures (cf. Chapter 7) until an abstract fixpoint is reached. In case of nested loops, a fixpoint on the inner loop is computed for each iteration of the outer loop, following [Bou93, MH92]. It is also possible to activate the extension of FuncTion with conflict-driven learning [DU15].

8.2 Experimental Evaluation

We evaluated our prototype implementation FuncTion against 288 terminating C programs collected from the termination category of the 4*th International Competition on Software Verification (SV-COMP 2015)*. FuncTion provides only a limited support for arrays and ponters. Therefore, we were not able to analyze 17% of the *SV-COMP 2015* benchmark test cases.

The experiments were performed on a system with a 1.30GHz 64-bit Dual-Core CPU (Intel i5-4250U) and 4GB of RAM, and running Ubuntu 14.04.

In Figure 8.1, we compared different parameterizations of FuncTion. The results match the expectations: FuncTion parameterized with interval constraints based on the intervals abstract domains (cf. Section 3.4.1) and a

	Tot	Time	Timeouts
FuncTion	200	1.5s	15
AProVE [SAF+15]	256	15.9s	24
FuncTion [Urb15]	175	0.7s	5
HIPTnT+ [LQC15]	246	1.2s	4
Ultimate [HDL+15]	226	15.3s	35

(a)

	FuncTion			
	■	▲	×	○
AProVE [SAF+15]	15	71	185	17
FuncTion [Urb15]	25	0	175	88
HIPTnT+ [LQC15]	22	68	178	20
Ultimate [HDL+15]	41	67	159	21

(b)

Figure 8.2: Overview of the experimental evaluation.

widening delay of two iterations is the fastest but least precise, and FuncTion parameterized with polyhedral constraints based on the polyhedra abstract domain (cf. Section 3.4.2) and a widening delay of three iterations is the slowest but most precise. We observed that delaying the widening further marginally improves precision but significantly increases running times.

We also compared FuncTion to the other tools that participated to the termination category of *SV-COMP 2015*: AProVE [SAF+15], the preliminary version of FuncTion [Urb15], HIPTnT+ [LQC15], and Ultimate [HDL+15]. FuncTion, with respect to its preliminary version, has been extended with conflict-driven learning [DU15]. In the comparison, we implemented FuncTion to use interval constraints and a two iteration widening delay, and respond to failure to prove a program terminating by using polyhedral constraints with three iterations widening delay. The reported execution times are for the entire run, which may involve trying both parameterizations. It was not possible to run all the tools on the same system because we did not have access to the competition versions of AProVE, HIPTnT+ and Ultimate. For these tools, we used the results of *SV-COMP 2015*[1] even though the competition was conducted on more powerful systems with a 3.40GHz 64-bit Quad-Core CPU (Intel i7-4770) and 33GB of RAM.

Figure 8.2 summarizes our experimental evaluation and Figure 8.3 shows a detailed comparison of FuncTion against each other tool. In Figure 8.2a, the first column reports the total number of programs that each tool could prove terminating, the second column reports the average running time in seconds for the programs where the tool proved termination, and the last column reports the number of time outs. We used a time limit of 180 seconds for each program test case. In Figure 8.2b, the first column (■) lists the total number of programs that the tool was not able to prove termination for and

[1] http://sv-comp.sosy-lab.org/2015/results/Termination.table.html

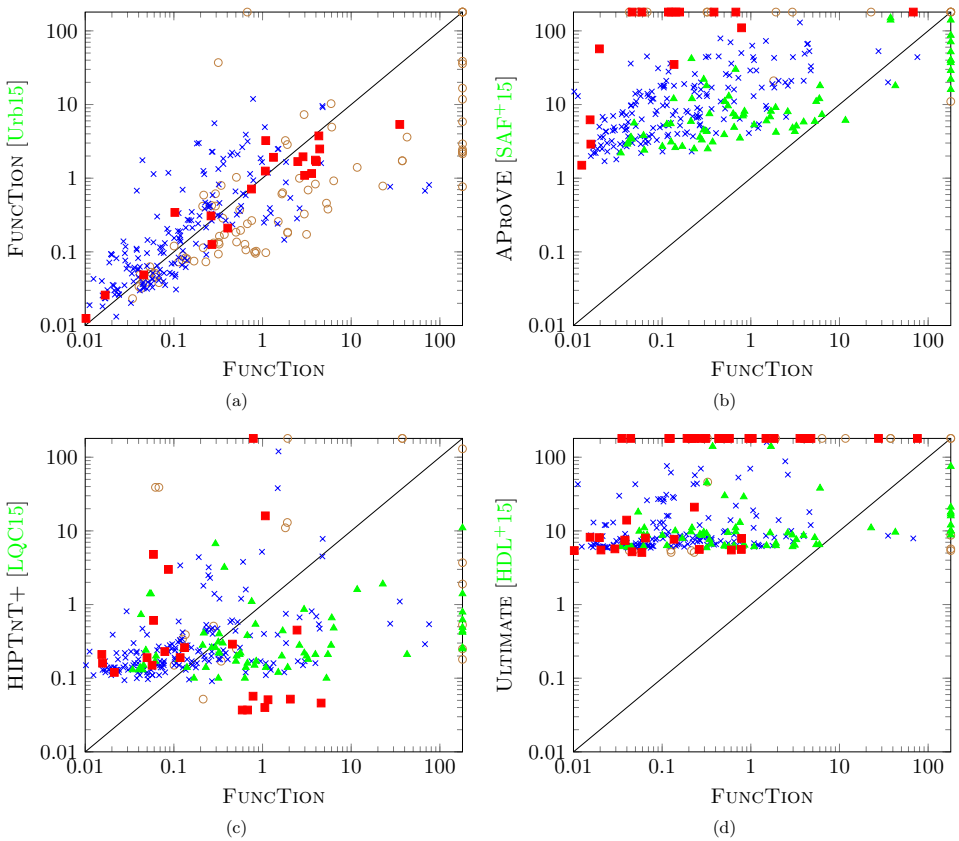


Figure 8.3: Detailed comparison of our prototype FuncTion against its previous version [Urb15] (a), AProVE [SAF+15] (b), HIPTnT+ [LQC15] (c), and Ultimate [HDL+15] (d).

that FuncTion could prove terminating, the second column (▲) reports the total number of programs that FuncTion was not able to prove termination for and that the tool could prove terminating, and the last two columns report the total number of programs that both the tool and FuncTion were able (×) or unable (○) to prove terminating. The same symbols are used in Figure 8.3.

Figure 8.2a shows that FucTion improves by around 9% the result of its preliminary version. The increase in the execution time is not evenly distributed, and about 2% of the program test cases require more than 20 seconds to be proved terminating by FuncTion (cf. also Figure 8.3a). The reason for this overhead is the heuristic that we have chosen to guide the conflict-driven analysis, which appears to be unfortunate in these cases [DU15]. From

	Tot	Time	Timeouts
FuncTion	20	0.1s	1
AProVE [SAF+15]	20	4.5s	3
FuncTion [Urb15]	18	0.2s	0
HIPTnT+ [LQC15]	19	0.3s	1
Ultimate [HDL+15]	22	7.4s	1

(a)

	FuncTion			
	■	▲	×	○
AProVE [SAF+15]	3	3	17	2
FuncTion [Urb15]	2	0	18	5
HIPTnT+ [LQC15]	5	4	15	1
Ultimate [HDL+15]	3	5	17	0

(b)

Figure 8.4: crafted: overview of the experimental evaluation.

Figure 8.3a it also emerges that, as expected, the previous version of FuncTion gives up earlier when unable to prove termination. Instead, FuncTion persists in the analysis and times out slightly more frequently.

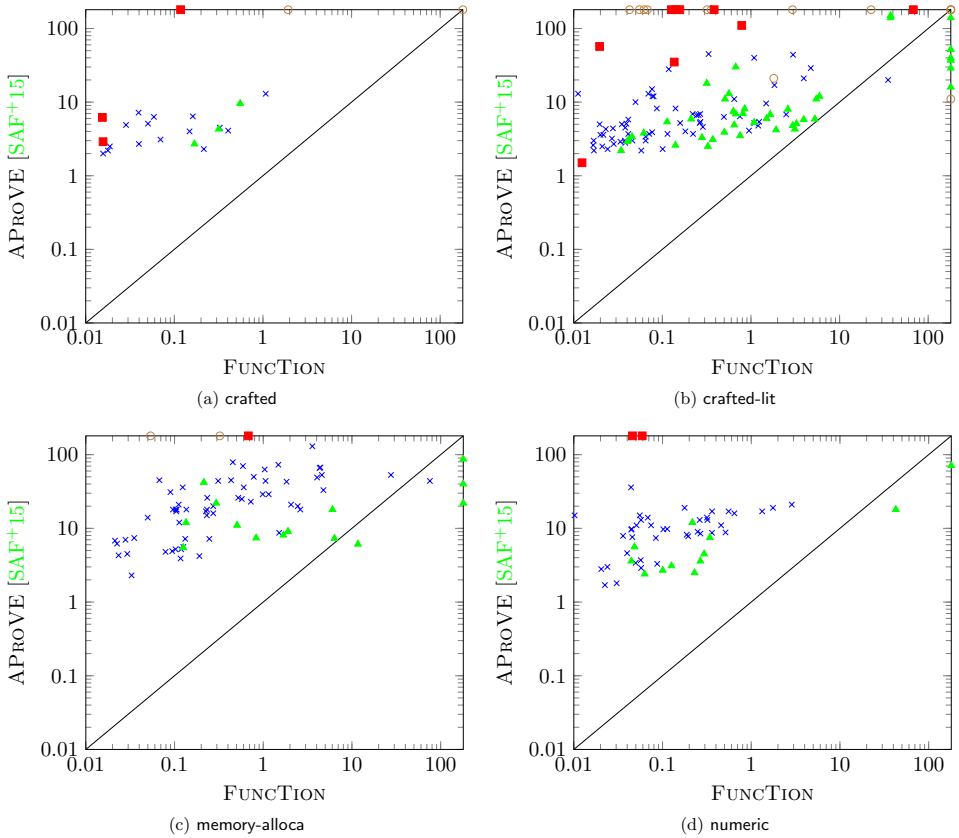
In Figure 8.3b and Figure 8.3d we clearly notice that, despite the fact that AProVE and Ultimate were run on the more powerful machines of *SV-COMP 2015*, FuncTion is generally faster but able to prove termination of respectively 19% and 9% fewer program test cases (cf. also Figure 8.2a).

HIPTnT+ is able to prove termination of 16% more programs than FuncTion (cf. Figure 8.2a), but FuncTion can prove termination of 52% of the programs that HIPTnT+ is not able to prove terminating (8% of the total program test cases, cf. Figure 8.2b). Note that, when performing the experiments with the previous version of FuncTion [Urb15] we observed that the *SV-COMP 2015* machines provided a 2x speedup. Thus, the comparison of the execution times of FuncTion and HIPTnT+ is currently inconclusive.

Finally, we noticed that 1% of the *SV-COMP 2015* program test cases could be proved terminating only by FuncTion (2.7% only by AProVE, 1% only by HIPTnT+, and 1.7% only by Ultimate). None of the tools was able to prove termination for 0.7% of the programs.

In the following, we further detail our experimental evaluation. In particular, we further discuss the comparison of our prototype implementation FuncTion with each of the other tools that participated to the termination category of *SV-COMP 2015*.

The *SV-COMP 2015* program test cases for termination are arranged into four verification tasks, according to the characteristics of the programs: crafted (programs manually crafted by the participants to the competition), crafted-lit (programs collected from the literature), memory-alloca (programs with dynamic memory allocation), and numeric (programs implementing numerical algorithms). For each verification task, a dedicated overview of the experimental evaluation is shown in Figure 8.4a and 8.4b (crafted), Figure 8.6a



(a) crafted (b) crafted-lit (c) memory-alloca (d) numeric

Figure 8.5: Detailed comparison of FUNCTION against APROVE [SAF+15].

and 8.6b (crafted-lit), Figure 8.8a and 8.8b (memory-alloca), and Figure 8.10a and 8.10b (numeric). As in Figure 8.2a, the first column of the tables reports the total number of programs that each tool was able to prove termination for, the second column reports the average running time in seconds for the programs where the tool proved termination, and the last column reports the number of time outs. We used a time limit of 180 seconds for each program test case. As in Figure 8.2b, the first column (■) of the tables reports the total number of programs that FUNCTION was able to prove terminating and that the tool was not, the second column (▲) reports the total number of programs that the tool was able to prove terminating and that FUNCTION was not, and the last two columns report the total number of programs that both the tool and FUNCTION were able (×) or unable (○) to prove terminating. The same

	Tot	Time	Timeouts
FuncTion	72	1.9s	10
AProVE [SAF+15]	105	13.2s	16
FuncTion [Urb15]	64	0.7s	2
HIPTnT+ [LQC15]	115	0.8s	3
Ultimate [HDL+15]	113	10s	5

(a)

	FuncTion			
	■	▲	×	○
AProVE [SAF+15]	9	42	63	13
FuncTion [Urb15]	8	0	64	55
HIPTnT+ [LQC15]	2	45	70	10
Ultimate [HDL+15]	7	48	65	7

(b)

Figure 8.6: crafted-lit: overview of the experimental evaluation.

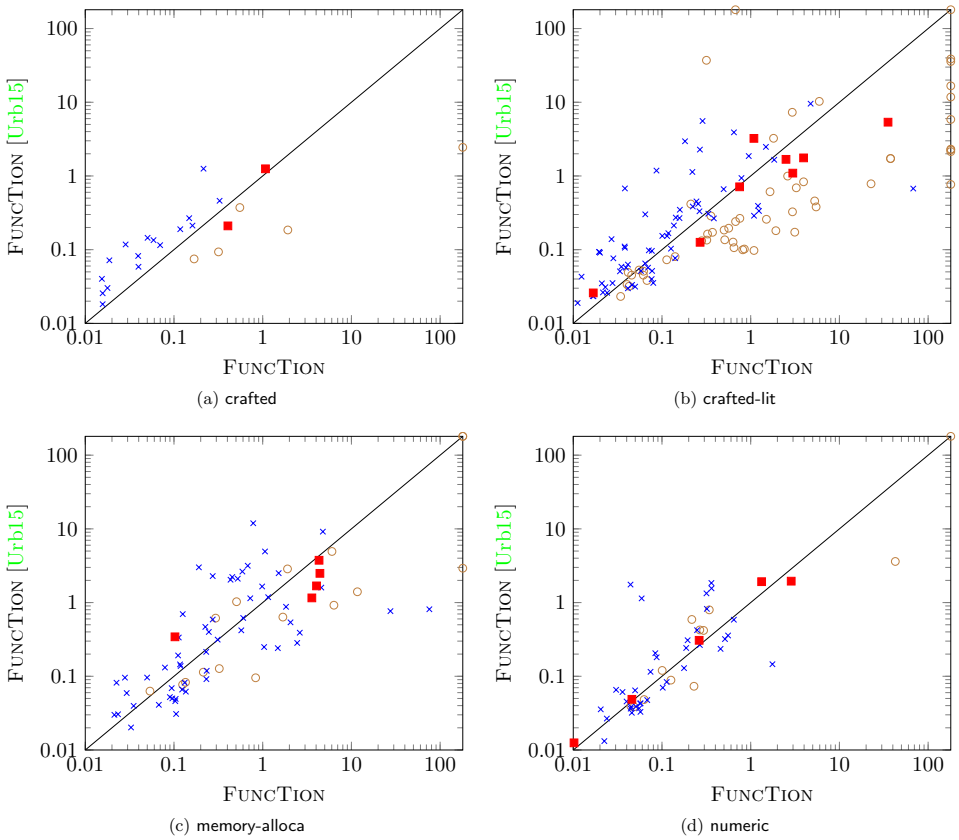
symbols are used for the detailed comparison of FuncTion against AProVE (Figure 8.5), its previous version (Figure 8.7), HIPTnT+ (Figure 8.9), and Ultimate (Figure 8.11).

Figure 8.5 shows, for each verification task, a detailed comparison of FuncTion against AProVE [SAF+15]. In the crafted verification task, the tool are able to prove the termination of the same number of programs (cf. also Figure 8.4a), while in the crafted-lit verification task, AProVE performs much better than FuncTion (cf. also Figure 8.6a). In all verification tasks, FuncTion is faster despite running on a less powerful machine than AProVE. In particular, we observe that AProVE times out on most of the programs that only FuncTion is able to prove terminating.

In Figure 8.7, we observe a comparison of FuncTion against its previous version [Urb15]. The numeric verification task is where FuncTion improves the most the result of its previous version (around 17%, cf. also Figure 8.10a). In the crafted-lit verification task, we clearly see that, as expected, FuncTion spends more execution time and often times out when unable to prove termination (cf. also Figure 8.6a).

The comparison of FuncTion against HIPTnT+ is shown in Figure 8.9. In the crafted-lit verification task HIPTnT+ performs better than FuncTion (cf. also Figure 8.6a), while FuncTion performs better than HIPTnT+ in the memory-alloca verification task (cf. also Figure 8.8a). Note that, as mentioned, when performing the experiments with the previous version of FuncTion [Urb15] we observed that the more powerful *SV-COMP 2015* machines provided a 2x speedup in the execution time. Nonetheless, in the crafted verification task FuncTion is faster than HIPTnT+ (cf. also Figure 8.4a), and in the numeric verification task the execution time of the tools is at least comparable (cf. also Figure 8.10a).

Figure 8.11 provides, for each verification task, a dedicated comparison of FuncTion against Ultimate [HDL+15]. Similarly to HIPTnT+, Ultimate performs much better than FuncTion in the crafted-lit verification task (cf.



(a) crafted (b) crafted-lit (c) memory-alloca (d) numeric

Figure 8.7: Detailed comparison of FuncTion against its previous version [Urb15].

also Figure 8.6), while FuncTion performs much better in the memory-alloca verification task (cf. also Figure 8.8). Moreover, despite running on a less powerful machine, FuncTion is faster while Ultimate often times out.

Finally, the programs that could be proved terminating only by FuncTion were 4% of the programs for the crafted verification task (4% only by Ultimate), 0.7% of the programs for the crafted-lit verification task (1.5% only by AProVE and HIPTnT+, and 3.1% only by Ultimate), and 1.7% of the programs for the numeric verification task (1.7% only by AProVE). For the memory-alloca, 6.5% and 1.3% of the programs could be proved terminating only by AProVE and HIPTnT+, respectively. All the programs that none of the tools could prove terminating belong to the crafted-lit verification task.

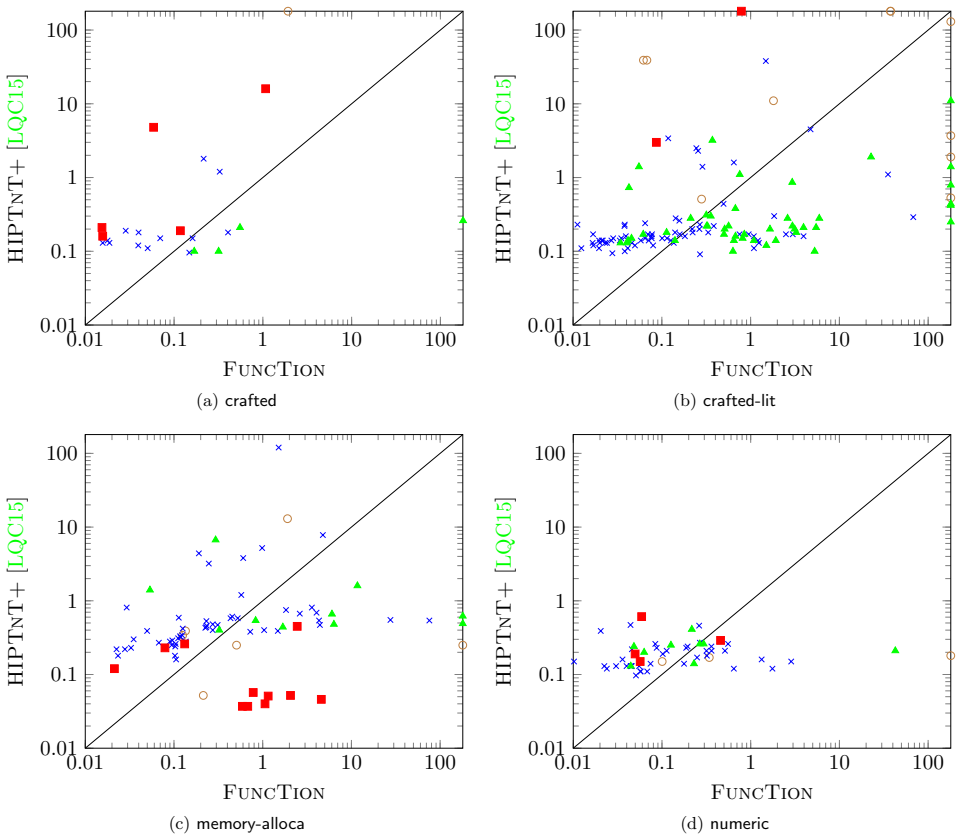
	Tot	Time	Timeouts
FuncTion	61	2.5s	3
AProVE [SAF+15]	74	26.8s	3
FuncTion [Urb15]	56	1.1s	2
HIPTnT+ [LQC15]	60	2.9s	0
Ultimate [HDL+15]	42	42.4s	28

(a)

	FuncTion			
	■	▲	×	○
AProVE [SAF+15]	1	14	60	2
FuncTion [Urb15]	5	0	56	16
HIPTnT+ [LQC15]	11	10	50	6
Ultimate [HDL+15]	25	6	36	10

(b)

Figure 8.8: memory-alloca: overview of the experimental evaluation.



(a) crafted

(b) crafted-lit

(c) memory-alloca

(d) numeric

Figure 8.9: Detailed comparison of FuncTion against HIPTnT+ [LQC15].

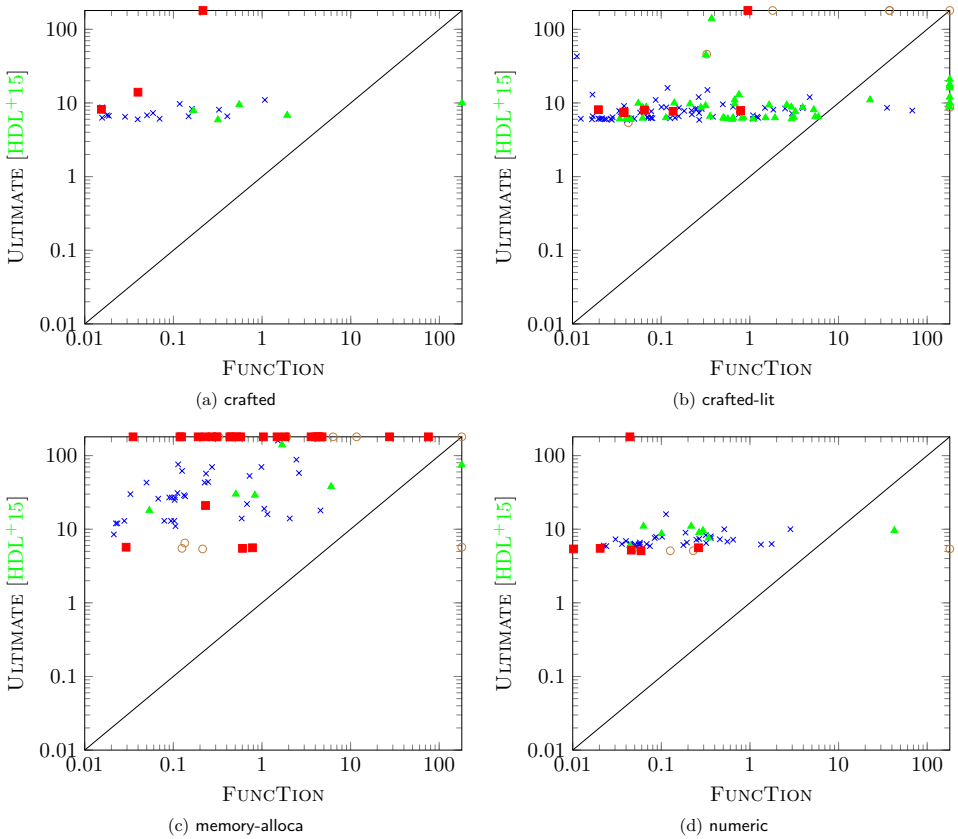
	Tot	Time	Timeouts
FuncTion	47	0.3s	1
AProVE [SAF+15]	57	10.7s	2
FuncTion [Urb15]	37	0.3s	1
HIPTnT+ [LQC15]	52	0.2s	0
Ultimate [HDL+15]	49	7.9s	1

(a)

	FuncTion			
	■	▲	×	○
AProVE [SAF+15]	2	12	45	0
FuncTion [Urb15]	10	0	37	12
HIPTnT+ [LQC15]	4	9	43	3
Ultimate [HDL+15]	6	8	41	4

(b)

Figure 8.10: numeric: overview of the experimental evaluation.



(a) crafted (b) crafted-lit (c) memory-alloca (d) numeric

Figure 8.11: Detailed comparison of FuncTion against Ultimate Automizer [HDL+15].

IV

Liveness

9

A Hierarchy of Temporal Properties

In this chapter, we generalize the abstract interpretation framework proposed for termination by Patrick Cousot and Radhia Cousot [CC12] and presented in Chapter 4, to other liveness properties. In particular, with reference to the hierarchy of temporal properties proposed by Zohar Manna and Amir Pnueli [MP90], we focus on guarantee ("something good occurs at least once") and recurrence ("something good occurs infinitely often") temporal properties.

Specifically, static analyses of guarantee and recurrence temporal properties are systematically derived by abstract interpretation of the program maximal trace semantics presented in Chapter 2 and Chapter 3. These methods automatically infer sufficient preconditions for the temporal properties by leveraging the abstract domains based on piecewise-defined ranking functions presented in Chapter 5 and Chapter 6. We augment these abstract domains with new operators including a dual widening.

Finally, we describe the implementation of the static analysis methods for guarantee and recurrence temporal properties into our prototype static analyzer FuncTion described in Chapter 8.

Dans ce chapitre, nous généralisons le cadre de l'interprétation abstraite proposé pour la terminaison par Patrick Cousot et Radhia Cousot [CC12] et présenté dans le Chapitre 4, à d'autres propriétés de vivacité. En particulier, en référence à la hiérarchie des propriétés temporelles proposée par Zohar Manna et Amir Pnueli [MP90], nous nous concentrons sur les propriétés temporelles de garantie ("quelque bon chose se produit au moins une fois") et de récurrence ("quelque bon chose se produit infiniment souvent").

Plus précisément, les analyses statiques pour les propriétés temporelles de garantie et de récurrence sont dérivées systématiquement par interprétation

abstraite de la sémantique de traces maximales des programmes présentée dans le Chapitre 2 et le Chapitre 3. Ces méthodes déduisent automatiquement des conditions suffisantes pour les propriétés temporelles en s'appuyant sur les domaines abstraits basés sur des fonctions de rang définies par morceaux présentées dans le Chapitre 5 et le Chapitre 6. Nous enrichissons ces domaines abstraits avec de nouveaux opérateurs, dont un élargissement dual.

Enfin, nous décrivons la mise en œuvre des méthodes d'analyse statique pour les propriétés temporelles de garantie et de récurrence dans notre prototype d'analyseur statique FuncTion *décrit dans le Chapitre 8.*

9.1 A Hierarchy of Temporal Properties

Leslie Lamport, in the late 1970s, suggested a classification of program properties into the classes of *safety* and *liveness* properties [Lam77]. The class of safety properties is informally characterized as the class of properties stating that "something *bad* never happens", that is, a program never reaches an unacceptable state. The class of liveness properties is informally characterized as the class of properties stating that "something *good* eventually happens", that is, a program *eventually* reaches a desirable state (cf. Section 1.2).

Zohar Manna and Amir Pnueli, in the late 1980s, suggested an alternative classification of program properties into a hierarchy [MP90], which distinguishes four basic classes making different claims about the frequency or occurrence of "something good" mentioned in the informal characterizations of the classes proposed by Leslie Lamport: *safety* properties, *guarantee* properties, *recurrence* properties, and *persistence* properties.

In the following, we only consider program properties expressible by *temporal logic*. To this end, we assume an underlying specification language, which is used to describe properties of program states.

For instance, for the small imperative language presented in Chapter 3, we define inductively the syntax of the state properties as follows:

$$\varphi \ ::= \ \mathit{bexp} \mid l : \mathit{bexp} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \qquad\qquad l \in \mathcal{L} \tag{9.1.1}$$

where *bexp* is a boolean expression (cf. Figure 3.1). The predicate $l : \mathit{bexp}$ allows specifying a program state property at a particular program control point $l \in \mathcal{L}$. When a program state $s \in \Sigma$ satisfies the property φ, we write $s \models \varphi$ and we say that s is a φ-state. We also slightly abuse notation and write φ to also denote the set $\{s \in \Sigma \mid s \models \varphi\}$ of states that satisfy the property φ.

Example 9.1.1

Let us consider the following program:

```
while 1(x ≥ 0) do
  2x := x + 1
od
while 3( true ) do
  if 4(x ≤ 10) then
    5x := x + 1
  then
    6x := −x
  fi
od7
```

The first loop is an infinite loop for the values of the variable x greater than or equal to zero: at each iteration the value of x is increased by one. The second loop is an infinite loop for any value of the variable x: at each iteration, the value of x is increased by one or negated when it becomes greater than ten.

The set of program environments $\mathcal{E}$ contains functions $\rho\colon \{x\} \to \mathbb{Z}$ mapping the program variable x to any possible value $\rho(x) \in \mathbb{Z}$. The set of program states $\Sigma \stackrel{\text{def}}{=} \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}\} \times \mathcal{E}$ consists of all pairs of numerical labels and environments; the initial states are $\mathcal{I} \stackrel{\text{def}}{=} \{\langle \mathbf{1}, \rho\rangle \mid \rho \in \mathcal{E}\}$.

An example of state property allowed by the specification language defined in Equation 9.1.1 is the property $x = 3$. The set of states that satisfy this property is $\{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}\} \times \{\langle x, 3\rangle\}$. Note however, that the states $\langle \mathbf{6}, \{\langle x, 3\rangle\}\rangle$ and $\langle \mathbf{7}, \{\langle x, 3\rangle\}\rangle$ are not reachable from the initial states.

Another examples of state property allowed by the specification language is $\mathbf{7} : x = 3$, which is only satisfied by the unreachable state $\langle \mathbf{7}, \{\langle x, 3\rangle\}\rangle$.

The program properties within the hierarchy are then defined by means of the temporal operators *always* $\Box$ and *eventually* $\Diamond$.

9.1.1 Safety Properties

The class of safety properties is informally characterized as the class of properties stating that "something good *alway* happens", that is, a program *never* reaches an unacceptable state. The safety properties that we consider are expressible by a temporal formula of the following form:

$$\Box\varphi$$

where φ is a state property. The temporal formula expresses that all program states in every program trace satisfy the state property φ.

In general, these safety properties are used to express *invariance* of some program state property over all computations.

A typical safety property is program *partial correctness*, which guarantees that all terminating computations produce correct results, and which is expressible by the following temporal formula:

$$\Box(l_e : \psi)$$

where $l_e \in \mathcal{L}$ denotes the program final control point and the formula ψ specifies the postcondition of the program.

Another typical safety property is *mutual exclusion*, which guarantees that no two concurrent processes can enter their critical section at the same time, and which is expressible by the following temporal formula:

$$\Box(l_1 : \texttt{false} \vee l_2 : \texttt{false})$$

where $l_1 \in \mathcal{L}$ and $l_2 \in \mathcal{L}$ denote the program control points representing the critical section of the first and second process, respectively.

Example 9.1.2

Let us consider again the program of Example 9.1.1:

```
while 1(x ≥ 0) do
  2x := x + 1
od
while 3( true ) do
  if 4(x ≤ 10) then
    5x := x + 1
  then
    6x := −x
  fi
od7
```

An example of safety property is the formula $\Box(x = 3)$, which is never satisfied by the program. Instead, since the first while loop increases the value of the variable x at each iteration, the safety property $\Box(3 \leq x)$ is satisfied when the program initial states are limited to the set $\{\langle \mathbf{1}, \rho\rangle \in \Sigma \mid 3 \leq \rho(x)\}$.

The class of safety properties that we consider are closed under *conjunction*. In fact, any temporal formula of the form $\Box\varphi_1 \wedge \Box\varphi_2$ is equivalent to the safety property formula $\Box(\varphi_1 \wedge \varphi_2)$.

9.1.2 Guarantee Properties

The class of guarantee properties is informally characterized as the class of properties stating that "something good happens *at least once*", that is, a program *eventually* reaches a desirable state. The guarantee properties that we consider are expressible by a temporal formula of the following form:

$$\Diamond\varphi$$

where φ is a state property. The temporal formula expresses that at least one program state in every program trace satisfies the property φ, but it does not promise any repetition.

In general, these guarantee properties are used to ensure that some event happens once during a program execution.

A typical guarantee property is program *termination*, which ensures that all computations are finite, expressible by the following temporal formula:

$$\Diamond(l_e : \texttt{true})$$

where $l_e \in \mathcal{L}$ denotes the program final control point.

Another typical guarantee property is program *total correctness*, which ensures that all computations starting in a φ-state terminate in a ψ-state, expressible by the following temporal formula:

$$\Diamond(l_i : \neg\varphi \vee l_e : \psi)$$

where $l_i, l_e \in \mathcal{L}$ respectively denote the initial and final program control point.

Example 9.1.3

Let us consider again the program of Example 9.1.1:

```
while 1(x ≥ 0) do
  2x := x + 1
od
while 3( true ) do
  if 4(x ≤ 10) then
    5x := x + 1
  then
    6x := −x
  fi
od7
```

An example of guarantee property is the formula $\Diamond(x = 3)$, which is satisfied when the program initial states are limited to the set $\{\langle \mathbf{1}, \rho\rangle \in \Sigma \mid \rho(x) \leq 3\}$. In particular, note that when the program initial state are limited to $\{\langle \mathbf{1}, \rho\rangle \in \Sigma \mid 0 \leq \rho(x) \leq 3\}$, the guarantee property is satisfied within the first while loop. Instead, when the program initial states are limited to $\{\langle \mathbf{1}, \rho\rangle \in \Sigma \mid \rho(x) < 0\}$, the guarantee property is satisfied within the second while loop.

Another example of guarantee property is the formula $\Diamond(3 \leq x)$, which is always satisfied by the program whatever its initial states.

The class of guarantee properties that we considered are closed under *disjunction*. In fact, any temporal formula of the form $\Diamond\varphi_1 \vee \Diamond\varphi_2$ is equivalent to the guarantee property formula $\Diamond(\varphi_1 \vee \varphi_2)$.

The classes of safety and guarantee properties that we consider are not closed under *negation*. On the other hand, the negation of a safety property formula $\Box\varphi$ is a guarantee property formula $\Diamond\neg\varphi$. Similarly, the negation of a guarantee property formula $\Diamond\varphi$ is a safety property formula $\Box\neg\varphi$.

9.1.3 Obligation Properties

The program properties that cannot be expressed by either a safety or a guarantee property formula belong to the *compound* class of obligation properties, which contains program properties expressible as a boolean combination of a safety property formula and guarantee property formula. The obligation properties that we consider are represented by a temporal formula of the form:

$$\Box\varphi \vee \Diamond\psi$$

where φ and ψ are state properties.

9.1.4 Recurrence Properties

The class of recurrence properties is informally characterized as the class of properties stating that "something good happens *infinitely often*", that is, a program reaches a desirable state *infinitely often*. The recurrence properties that we consider are expressible by a temporal formula of the following form:

$$\Box\Diamond\varphi$$

where φ is a state property. The temporal formula expresses that infinitely many program states in every program trace satisfy the property φ.

In general, these recurrence properties are used to ensure that some event happens infinitely many times during a program execution.

A typical recurrence property is program *starvation freedom*, which ensures that a process will eventually enter its critical section, and which is expressible by the following temporal formula:

$$\Box\Diamond(l_c : \texttt{true})$$

where $l_c \in \mathcal{L}$ is the program control point representing the critical section.

Example 9.1.4

Let us consider again the program of Example 9.1.1:

```
while 1(x ≥ 0) do
  2x := x + 1
od
while 3( true ) do
  if 4(x ≤ 10) then
    5x := x + 1
  then
    6x := −x
  fi
od7
```

The recurrence property represented by the formula $\Box\Diamond x = 3$ is satisfied when the program initial states are limited to the set $\{\langle \mathbf{1}, \rho\rangle \in \Sigma \mid \rho(x) < 0\}$. In particular, note that the recurrence property is satisfied only within the second while loop. Instead, the recurrence property $\Box\Diamond 3 \leq x$ is always satisfied by the program whatever its initial states.

The class of recurrence properties that we consider are closed under disjunction. In fact, any temporal formula of the form $\Box\Diamond\varphi_1 \vee \Box\Diamond\varphi_2$ is equivalent to the recurrence property formula $\Box\Diamond(\varphi_1 \vee \varphi_2)$.

9.1.5 Persistence Properties

The class of persistence properties is informally characterized as the class of properties stating that "something good *eventually* happens *continuously*", that is, a program *eventually* reaches a desirable state and *continues* to stay in a desirable state. The persistence properties that we consider are expressible by a temporal formula of the following form:

$$\Diamond\Box\varphi$$

where φ is a state property. The temporal formula expresses that all but finitely many program states (and, in particular, all program states from a certain point on) in every program trace satisfy the property φ.

In general, these persistence properties are used to ensure the eventual stabilization of the program into a state. They allow an arbitrary delay until the stabilization, but require that once it occurs it is continuously maintained.

Example 9.1.5

Let us consider again the program of Example 9.1.1:

```
while 1(x ≥ 0) do
  2x := x + 1
od
while 3( true ) do
  if 4(x ≤ 10) then
    5x := x + 1
  then
    6x := −x
  fi
od7
```

An example of persistence property is the formula $\Diamond\Box x = 3$, which is never satisfied by the program. Instead, the persistence property represented by the formula $\Diamond\Box 3 \leq x$ is satisfied when the program initial states are limited to the set $\{\langle \mathbf{1}, \rho\rangle \in \Sigma \mid 0 \leq \rho(x)\}$. In particular, note that the persistence property is satisfied only within the first while loop.

The class of persistence properties that we consider are closed under *conjunction*. In fact, any temporal formula of the form $\Diamond\Box\varphi_1 \wedge \Diamond\Box\varphi_2$ is equivalent to the persistence property formula $\Diamond\Box(\varphi_1 \wedge \varphi_2)$.

The classes of recurrence and persistence properties that we consider are not closed under *negation* but, analogously to the classes of safety and guarantee properties, the negation of a formula in one of the classes belongs to the other. The negation of a recurrence property formula $\Box\Diamond\varphi$ is a persistence property formula $\Diamond\Box\neg\varphi$. Similarly, the negation of a persistence property formula $\Diamond\Box\varphi$ is a recurrence property formula $\Box\Diamond\neg\varphi$.

9.1.6 Reactivity Properties

The program properties that cannot be expressed by either a recurrence or a persistence property formula belong to the *compound* class of reactivity prop-

erties, which contains program properties expressible as a boolean combination of a recurrence property formula and a persistence property formula. The reactivity properties that we consider are represented by a temporal formula of the following form:

$$\Box\Diamond\varphi \vee \Diamond\Box\psi$$

where φ and ψ are state properties.

9.2 Guarantee Semantics

In the following, we generalize Section 4.2 and Section 4.3 from *definite* termination to guarantee properties. We define a sound and complete semantics for proving guarantee temporal properties by abstract interpretation of the program maximal trace semantics (cf. Equation 2.2.5). The generalization is straightforward but provides a building block for the next Section 9.3. Then, we propose a sound decidable abstraction based on piecewise-defined ranking functions (cf. Chapter 5 and Chapter 6).

9.2.1 Guarantee Semantics

The *guarantee semantics*, given a set of desirable states $S \subseteq \Sigma$, is a ranking function $\tau_{\mathrm{g}}[S] \in \Sigma \rightharpoonup \mathbb{O}$ defined starting from the states in S, where the function has value zero, and retracing the program *backwards* while mapping every state in Σ definitely leading to a state in S (i.e., a state such that all the traces to which it belongs eventually reach a state in S) to an ordinal in $\mathbb{O}$ representing an upper bound on the number of program execution steps remaining to S. The domain $\mathrm{dom}(\tau_{\mathrm{g}}[S])$ of $\tau_{\mathrm{g}}[S]$ is the set of states definitely leading to a desirable state in S: all traces branching from a state $s \in \mathrm{dom}(\tau_{\mathrm{g}}[S])$ reach a state in S in at most $\tau_{\mathrm{g}}[S]s$ execution steps, while at least one trace branching from a state $s \notin \mathrm{dom}(\tau_{\mathrm{g}}[S])$ never reaches S.

Note that the program traces that satisfy a guarantee property can also be *infinite* traces. In particular, guarantee properties are satisfied by finite subsequences of possibly infinite traces. Thus, in order to reason about subsequences, we define the function $\mathrm{sq}\colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+})$, which extracts the finite subsequences of a set of sequences $T \subseteq \Sigma^{+\infty}$:

$$\mathrm{sq}(T) \stackrel{\mathrm{def}}{=} \{\sigma \in \Sigma^{+} \mid \exists \sigma' \in \Sigma^{*}, \sigma'' \in \Sigma^{*\infty} : \sigma'\sigma\sigma'' \in T\} \tag{9.2.1}$$

We recall that the neighborhood of a sequence $\sigma \in \Sigma^{+\infty}$ in a set of sequences $T \subseteq \Sigma^{+\infty}$ is the set of sequences $\sigma' \in T$ with a common prefix with σ (cf. Equation 4.2.3). A finite subsequence of a program trace satisfies a guarantee

property if and only if it terminates in the desirable set of states (and never encounters a desirable state before), and its neighborhood in the subsequences of the program semantics consists only of sequences that are terminating in the desirable set of states (and never encounter a desirable state before). The corresponding *guarantee abstraction* $\alpha^{\mathrm{g}}[S]\colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+})$ is parameterized by a set of desirable states $S \subseteq \Sigma$ and it is defined as follows:

$$\alpha^{\mathrm{g}}[S]T \stackrel{\mathrm{def}}{=} \left\{\sigma s \in \mathrm{sq}(T) \;\middle|\; \sigma \in \bar{S}^{*},\ s \in S,\ \mathrm{nhbd}(\sigma, \mathrm{sf}(T) \cap \bar{S}^{+\infty}) = \emptyset\right\} \tag{9.2.2}$$

where $\bar{S} \stackrel{\mathrm{def}}{=} \Sigma \setminus S$ and the function $\mathrm{sf}\colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+\infty})$ yields the set of suffixes of a set of sequences $T \subseteq \Sigma^{+\infty}$:

$$\mathrm{sf}(T) \stackrel{\mathrm{def}}{=} \bigcup\{\sigma \in \Sigma^{+\infty} \mid \exists \sigma' \in \Sigma^{*} : \sigma'\sigma \in T\}. \tag{9.2.3}$$

Example 9.2.1

Let $T \stackrel{\mathrm{def}}{=} \{(abcd)^{\omega}, (cd)^{\omega}, a^{\omega}, cd^{\omega}\}$ and let $S \stackrel{\mathrm{def}}{=} \{c\}$. We have $\mathrm{sf}(T) \cap \bar{S}^{+\infty} = \{a^{\omega}, d^{\omega}\}$. Then, we have $\alpha^{\mathrm{g}}[S]T = \{c, bc\}$. In fact, let us consider the trace $(abcd)^{\omega}$: the subsequences of $(abcd)^{\omega}$ that are terminating with c and never encounter c before are $\{c, bc, abc, dabc\}$; for abc, we have $\mathrm{pf}(ab) \cap \mathrm{pf}(a^{\omega}) = \{a\} \neq \emptyset$ and, for $dabc$, we have $\mathrm{pf}(dab) \cap \mathrm{pf}(d^{\omega}) = \{d\} \neq \emptyset$. Similarly, let us consider $(cd)^{\omega}$: the subsequences of $(cd)^{\omega}$ that are terminating with c and never encounter c before are $\{c, dc\}$; for dc, we have $\mathrm{pf}(d) \cap \mathrm{pf}(d^{\omega}) = \{d\} \neq \emptyset$.

We can now define the guarantee semantics $\tau_{\mathrm{g}}[S] \in \Sigma \rightharpoonup \mathbb{O}$:

Definition 9.2.2 (Guarantee Semantics) *Given a desirable set of states $S \subseteq \Sigma$, the* guarantee semantics $\tau_g[S] \in \Sigma \rightharpoonup \mathbb{O}$ *is an abstract interpretation of the maximal trace semantics $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ (cf. Equation 2.2.5):*

$$\tau_g[S] \stackrel{def}{=} \alpha^{Mrk}(\alpha^{g}[S](\tau^{+\infty})) \tag{9.2.4}$$

where the abstraction $\alpha^{\mathrm{Mrk}}\colon \mathcal{P}(\Sigma^{+}) \to (\Sigma \rightharpoonup \mathbb{O})$ is the same ranking abstraction defined in Equation 4.2.14.

The following result provides a fixpoint definition of the guarantee semantics within the partially ordered set $\langle \Sigma \rightharpoonup \mathbb{O}, \sqsubseteq\rangle$, where the computational order is defined as in Equation 4.2.15 as:

$$f_1 \sqsubseteq f_2 \iff \mathrm{dom}(f_1) \subseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_1) : f_1(x) \leq f_2(x).$$

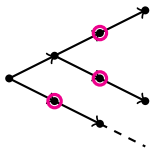
Theorem 9.2.3 (Guarantee Semantics) *Given a desirable set of states $S \subseteq \Sigma$, the guarantee semantics $\tau_g[S] \in \Sigma \rightharpoonup \mathbb{O}$ can be expressed as a least fixpoint in the partially ordered set $\langle \Sigma \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$:*

$$\tau_g[S] \;=\; \mathrm{lfp}_{\dot{\emptyset}}^{\sqsubseteq}\; \phi_g[S]$$

$$\phi_g[S]f \;\stackrel{def}{=}\; \lambda s. \begin{cases} 0 & s \in S \\ \sup\{f(s') + 1 \mid \langle s, s' \rangle \in \tau\} & s \notin S \wedge s \in \widetilde{\mathrm{pre}}(\mathrm{dom}(f)) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{9.2.5}$$

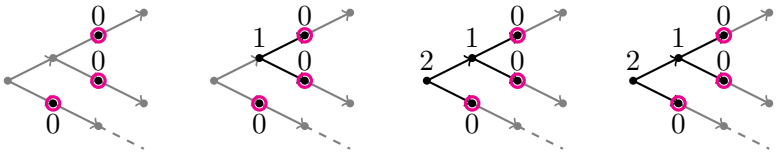
Example 9.2.4

Let us consider the following trace semantics:



where the highlighted states are the set S of desirable states.

The fixpoint iterates of the guarantee semantics $\tau_{\mathrm{g}}[S] \in \Sigma \rightharpoonup \mathbb{O}$ are:



where unlabelled states are outside the domain of the function.

Note that, when the set of desirable states S is the set of final states Ω, unsurprisingly we rediscover the definite termination semantics presented in Section 4.2, since $\phi_{\mathrm{g}}[\Omega] = \phi_{\mathrm{Mt}}$ (cf. Equation 4.2.16).

Let φ be a state property. The *φ-guarantee semantics* $\tau_{\mathrm{g}}^{\varphi} \in \Sigma \rightharpoonup \mathbb{O}$:

$$\tau_{\mathrm{g}}^{\varphi} \;\stackrel{\mathrm{def}}{=}\; \tau_{\mathrm{g}}[\varphi] \tag{9.2.6}$$

is sound and complete for proving a guarantee property $\Diamond\varphi$:

Theorem 9.2.5 *A program satisfies a guarantee property $\Diamond\varphi$ for execution traces starting from a given set of initial states $\mathcal{I}$ if and only if $\mathcal{I} \subseteq \mathrm{dom}(\tau_g^{\varphi})$.*

Proof.

See Appendix A.6. ■

9.2.2 Denotational Guarantee Semantics

In the following, we provide a structural definition of the φ-guarantee semantics $\tau_{\mathrm{g}}^{\varphi} \in \Sigma \rightharpoonup \mathbb{O}$ by induction on the syntax of programs written in the small language presented in Chapter 3.

We partition $\tau_{\mathrm{g}}^{\varphi}$ with respect to the program control points: $\tau_{\mathrm{g}}^{\varphi} \in \mathcal{L} \to (\mathcal{E} \rightharpoonup \mathbb{O})$. In this way, to each program control point $l \in \mathcal{L}$ corresponds a partial function $f \colon \mathcal{E} \rightharpoonup \mathbb{O}$, and to each program instruction *stmt* corresponds a guarantee semantics $\tau_{\mathrm{g}}^{\varphi}[\![\, \mathit{stmt} \,]\!] \colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$.

The guarantee semantics $\tau_{\mathrm{g}}^{\varphi}[\![\, \mathit{stmt} \,]\!] \colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ of each program instruction *stmt* takes as input a ranking function whose domain represents the environments always leading to φ from the final label of *stmt*, and determines the ranking function whose domain represents the environments always leading to φ from the initial label of *stmt*, and whose value represents an upper bound on the number of program execution steps remaining to φ.

The guarantee semantics of a `skip` instruction *resets* the value of the input ranking function $f \colon \mathcal{E} \rightharpoonup \mathbb{O}$ for the environments that satisfy φ, and otherwise it increases its value to take into account another program execution step to reach φ from the environments at the initial label of the instruction:

$$\tau_{\mathrm{g}}^{\varphi}[\![\, {}^{l}\mathtt{skip} \,]\!] f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 0 & \langle l, \rho \rangle \models \varphi \\ f(\rho) + 1 & \langle l, \rho \rangle \not\models \varphi \wedge \rho \in \mathrm{dom}(f) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{9.2.7}$$

Similarly, the guarantee semantics of a variable assignment ${}^{l}X := \mathit{aexp}$ *resets* the value of the input ranking function $f \colon \mathcal{E} \rightharpoonup \mathbb{O}$ for the environments that satisfy φ; otherwise, the resulting ranking function is defined over the environments that, when subject to the variable assignment, always belong to the domain of the input ranking function. The value of the input ranking function for these environments is increased by one, to take into account another execution step, and the value of the resulting ranking function is the least upper bound of these values:

$$\tau_{\mathrm{g}}^{\varphi}[\![\, {}^{l}X := \mathit{aexp} \,]\!] f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 0 & \langle l, \rho \rangle \models \varphi \\ \sup\{f(\rho[X \leftarrow v]) + 1 \mid v \in [\![\mathit{aexp}]\!]\rho\} & \\ & \langle l, \rho \rangle \not\models \varphi \wedge \exists v \in [\![\mathit{aexp}]\!]\rho \wedge \\ & \forall v \in [\![\mathit{aexp}]\!]\rho : \rho[X \leftarrow v] \in \mathrm{dom}(f) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{9.2.8}$$

Note that all environments yielding a run-time error due to a division by zero do not belong to the domain of the termination semantics of the assignment.

Example 9.2.6

Let $\mathcal{X} \stackrel{\text{def}}{=} \{x\}$. We consider the following ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$:

$$f(\rho) \stackrel{\text{def}}{=} \begin{cases} 2 & \rho(x) = 1 \\ 3 & \rho(x) = 2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

the backward assignment $x{:=}x + [1, 2]$ and the guarantee property $\Diamond(x = 3)$. The guarantee semantics of the assignment, given the ranking function, is:

$$\tau_{\mathrm{g}}^{x=3}[\![\, x := x + [1, 2] \,]\!] f(\rho) \stackrel{\text{def}}{=} \begin{cases} 4 & \rho(x) = 0 \\ 0 & \rho(x) = 3 \\ \text{undefined} & \text{otherwise} \end{cases}$$

In particular, note that unlike Example 4.3.1, the function is also defined when $\rho(x) = 3$, since the environment satisfies the property $x = 3$.

Given a conditional instruction `if` ^{l}*bexp* `then` *stmt*$_1$ `else` *stmt*$_2$ `fi`, its guarantee semantics takes as input a ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$ and derives the guarantee semantics $\tau_{\mathrm{g}}^{\varphi}[\![\, \mathit{stmt}_1 \,]\!] f$ of stmt_1, in the following denoted by S_1, and the guarantee semantics $\tau_{\mathrm{g}}^{\varphi}[\![\, \mathit{stmt}_2 \,]\!] f$ of stmt_2, in the following denoted by S_2. Then, the guarantee semantics of the conditional instruction is defined by means of the ranking function $F[f]\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments that belong to the domain of S_1 and to the domain of S_2, and that may both satisfy and not satisfy the boolean expression *bexp*:

$$F[f] \stackrel{\text{def}}{=} \lambda\rho \in \mathrm{dom}(S_1) \cap \mathrm{dom}(S_2).\ \begin{cases} \sup\{S_1(\rho) + 1, S_2(\rho) + 1\} & \\ & [\![\mathit{bexp}]\!]\rho = \{\text{true}, \text{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

and the ranking function $F_1[f]\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments $\rho \in \mathcal{E}$ that belong to the domain of S_1 and that must satisfy *bexp*:

$$F_1[f] \stackrel{\text{def}}{=} \lambda\rho \in \mathrm{dom}(S_1).\ \begin{cases} S_1(\rho) + 1 & [\![\mathit{bexp}]\!]\rho = \{\text{true}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

and the ranking function $F_2[f]\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments that belong to the domain of S_2 and that cannot satisfy *bexp*:

$$F_2[f] \stackrel{\text{def}}{=} \lambda\rho \in \text{dom}(S_2).\ \begin{cases} S_2(\rho) + 1 & [\![bexp]\!]\rho = \{\text{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The resulting ranking function is defined by joining $F[f]$, $F_1[f]$, and $F_2[f]$, and resetting the value of the function for the environments that satisfy φ:

$$\tau_{\text{g}}^{\varphi}[\![\,\texttt{if}\ {}^{l}bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi}\,]\!]f \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 0 & \langle l, \rho\rangle \models \varphi \\ G(\rho) & \langle l, \rho\rangle \not\models \varphi \wedge \\ & \rho \in \text{dom}(G) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{9.2.9}$$

where $G \stackrel{\text{def}}{=} F[f] \dot\cup F_1[f] \dot\cup F_2[f]$.

Example 9.2.7

Let $\mathcal{X} \stackrel{\text{def}}{=} \{x\}$. We consider the guarantee property $\Diamond(x = 3)$ and the guarantee semantics of the conditional statement `if` *bexp* `then` $stmt_1$ `else` $stmt_2$ `fi`. We assume, given a ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$, that the guarantee semantics of $stmt_1$ is defined as:

$$\tau_{\text{g}}^{x=3}[\![\,stmt_1\,]\!]f(\rho) \stackrel{\text{def}}{=} \begin{cases} 1 & \rho(x) \le 0 \\ 0 & \rho(x) = 3 \\ \text{undefined} & \text{otherwise} \end{cases}$$

and that the guarantee semantics of $stmt_2$ is defined as

$$\tau_{\text{g}}^{x=3}[\![\,stmt_2\,]\!]f(\rho) \stackrel{\text{def}}{=} \begin{cases} 3 & 0 \le \rho(x) < 3 \\ 0 & \rho(x) = 3 \\ 3 & 3 < \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Then, when the boolean expression *bexp* is for example $x \le 3$, the guarantee semantics of the conditional statement is:

$$\tau_{\text{g}}^{x=3}[\![\,\texttt{if}\ {}^{l}bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi}\,]\!]f(\rho) \stackrel{\text{def}}{=} \begin{cases} 2 & \rho(x) \le 0 \\ 0 & \rho(x) = 3 \\ 4 & 3 < \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Instead, when *bexp* is for example the non-deterministic choice `?`, we have:

$$\tau_{\mathrm{g}}^{x=3}[\![\,\texttt{if}\ {}^{l}\mathit{bexp}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \texttt{fi}\,]\!]f(\rho) \stackrel{\text{def}}{=} \begin{cases} 4 & \rho(x) = 0 \\ 0 & \rho(x) = 3 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that, unlike Example 4.3.2, both functions are also defined when $\rho(x) = 3$, since the environment satisfies the property $x = 3$.

The guarantee semantics of a loop instruction `while` ^{l}*bexp* `do` *stmt* `od` takes as input a ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain represents the environments leading to φ from the final label of the instruction, and outputs the ranking function which is defined as the least fixpoint of the function $\phi_{\mathrm{g}}^{\varphi}\colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ within $\langle \mathcal{E} \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$, analogously to Equation 9.2.5:

$$\tau_{\mathrm{g}}^{\varphi}[\![\,\texttt{while}\ {}^{l}\mathit{bexp}\ \texttt{do}\ \mathit{stmt}\ \texttt{od}\,]\!]f \stackrel{\text{def}}{=} \mathrm{lfp}_{\dot{\emptyset}}^{\sqsubseteq}\ \phi_{\mathrm{g}}^{\varphi} \tag{9.2.10}$$

where the *computational order* is defined as in Equation 4.2.15:

$$f_1 \sqsubseteq f_2 \iff \mathrm{dom}(f_1) \subseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_1) : f_1(x) \leq f_2(x).$$

The function $\phi_{\mathrm{g}}^{\varphi}\colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ takes as input a ranking function $x\colon \mathcal{E} \rightharpoonup \mathbb{O}$, resets its value for the environments that satisfy φ, and adds to its domain the environments for which one more loop iteration is needed before φ. In the following, the guarantee semantics $\tau_{\mathrm{g}}^{\varphi}[\![\,\mathit{stmt}\,]\!]x$ of the loop body is denoted by S. The function $\phi_{\mathrm{g}}^{\varphi}$ is defined by means of the ranking function $F[x]\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments that belong to the domain of S and to the domain of the input function f, and that may both satisfy and not satisfy the boolean expression *bexp*:

$$F[x] \stackrel{\text{def}}{=} \lambda\rho \in \mathrm{dom}(S) \cap \mathrm{dom}(f).\ \begin{cases} \sup\{S(\rho) + 1, f(\rho) + 1\} & \\ & [\![\mathit{bexp}]\!]\rho = \{\text{true}, \text{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

and the ranking function $F_1[x]\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments $\rho \in \mathcal{E}$ that belong to the domain of S and that must satisfy *bexp*:

$$F_1[x] \stackrel{\text{def}}{=} \lambda\rho \in \mathrm{dom}(S).\ \begin{cases} S(\rho) + 1 & [\![\mathit{bexp}]\!]\rho = \{\text{true}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

and the ranking function $F_2[f]\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments that belong to the domain of the input function f and that cannot satisfy *bexp*:

$$F_2[f] \stackrel{\text{def}}{=} \lambda\rho \in \mathrm{dom}(f).\ \begin{cases} f(\rho)+1 & [\![bexp]\!]\rho = \{\text{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The resulting ranking function is defined by joining $F[x]$, $F_1[x]$, and $F_2[f]$, and resetting the value of the function for the environments that satisfy φ:

$$\phi_{\mathrm{g}}^{\varphi}(x) \stackrel{\text{def}}{=} \lambda\rho. \begin{cases} 0 & \langle l,\rho\rangle \models \varphi \\ G(\rho) & \langle l,\rho\rangle \not\models \varphi \wedge \rho \in \mathrm{dom}(G) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{9.2.11}$$

where $G \stackrel{\text{def}}{=} F[x] \dot\cup F_1[x] \dot\cup F_2[f]$.

Finally, the guarantee semantics of the sequential combination of instructions $stmt_1\ stmt_2$, takes as input a ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$, determines from f the guarantee semantics $\tau_{\mathrm{g}}^{\varphi}[\![\, stmt_2 \,]\!]f$ of $stmt_2$, and outputs the ranking function determined by the guarantee semantics of $stmt_1$ from $\tau_{\mathrm{g}}^{\varphi}[\![\, stmt_2 \,]\!]f$:

$$\tau_{\mathrm{g}}^{\varphi}[\![\, stmt_1\ stmt_2 \,]\!]f \stackrel{\text{def}}{=} \tau_{\mathrm{g}}^{\varphi}[\![\, stmt_1 \,]\!](\tau_{\mathrm{g}}^{\varphi}[\![\, stmt_2 \,]\!]f) \tag{9.2.12}$$

The guarantee semantics $\tau_{\mathrm{g}}^{\varphi}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$ of a program *prog* is a ranking function whose domain represents the environments always leading to φ, which is determined by taking as input the constant function equal to zero for the environments that satisfy φ, and undefined otherwise:

Definition 9.2.8 (Guarantee Semantics) *The* guarantee semantics $\tau_g^{\varphi}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$ *of a program prog is:*

$$\tau_g^{\varphi}[\![\, prog \,]\!] = \tau_g^{\varphi}[\![\, stmt^{\,l} \,]\!] \stackrel{def}{=} \tau_g^{\varphi}[\![\, stmt \,]\!]\left(\lambda\rho. \begin{cases} 0 & \langle l,\rho\rangle \models \varphi \\ undefined & otherwise \end{cases}\right) \tag{9.2.13}$$

where the function $\tau_g^{\varphi}[\![\, stmt \,]\!]\colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ *is the guarantee semantics of each program instruction stmt.*

Note that, as pointed out in Remark 3.2.2, possible run-time errors are ignored. Thus, all environments leading to run-time errors do not belong to the domain of the guarantee semantics of a program *prog*.

9.2.3 Abstract Guarantee Semantics

In the following, we propose a sound decidable abstraction of the guarantee semantics $\tau_{\mathrm{g}}^{\varphi}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$, which is based on the piecewise-defined ranking functions presented in Chapter 5 and Chapter 6. The abstraction is sound with respect to the same *approximation order* defined in Equation 4.2.12 as:

$$f_1 \preccurlyeq f_2 \iff \mathrm{dom}(f_1) \supseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_2) : f_1(x) \leq f_2(x).$$

In particular, we complement the operators of the decision trees abstract domain presented in Chapter 5 with a new unary operator $\textsc{reset}_{\mathrm{T}}[\![\, \varphi \,]\!]$, which *resets* the leaves of a decision tree that satisfy a given property φ.

Reset. The operator $\textsc{reset}_{\mathrm{T}}[\![\, \varphi \,]\!] \colon \mathcal{L} \to \mathcal{D} \to \mathcal{T} \to \mathcal{T}$ for the reset of decision trees is implemented by Algorithm 24: the function RESET, given a program control point $p \in \mathcal{L}$, a sound over-approximation $d \in \mathcal{D}$ of the reachable environments, and a decision tree $t \in \mathcal{T}$, reasons by induction on the structure of the property φ. In particular, when φ specifies the property at a particular program control point $l \in \mathcal{L}$ (cf. Line 55), the decision tree is reset only if l coincides with p (cf. Line 57). Instead, when φ is a conjunction of two properties φ_1 and φ_2 (cf. Line 58), or a conjunction of two boolean expressions $bexp_1$ and $bexp_2$ (cf. Line 44), the resulting decision trees are merged by the function MEET defined in Algorithm 8 (cf. Line 59 and Line 47). Similarly, when φ is a disjunction of two properties φ_1 and φ_2 (cf. Line 60), or a disjunction of two boolean expressions $bexp_1$ and $bexp_2$ (cf. Line 48), the resulting decision trees are merged by the approximation join A-JOIN defined in Algorithm 6 (cf. Line 61 and Line 51). Finally, when φ is a comparison of arithmetic expressions $aexp_1 \bowtie aexp_2$ (cf. Line 52), a set of constraints J is produced by the operator $\textsc{filter}_{\mathrm{C}}$ defined in Equation 5.2.23 (cf. Line 53), which is used by the auxiliary function RESET-AUX (cf. Line 54).

The function RESET-AUX augments a decision tree with a given set of linear constraints. Then, only the subtrees whose paths from the root of the decision tree satisfy these constraints are reset. In particular, the function RESET-AUX takes as input a decision tree $t \in \mathcal{T}$, a set $C \in \mathcal{P}(\mathcal{C})$ of linear constraints representing an over-approximation of the reachable environments, and a set $J \in \mathcal{P}(\mathcal{C})$ of linear constraints that need to be added to the decision tree. When J is not empty (cf. Line 12), the linear constraints J are added to the decision tree in descending order with respect to $<_{\mathrm{C}}$: at each iteration a linear constraint $j \in \mathcal{C}$ is extracted from J (cf. Line 13), which is the largest constraint in J with respect to the constraints in canonical form. Then, the function RESET-AUX possibly adds a decision node for the linear constraint j

Algorithm 24 : Tree Reset

```
1:  function RESET-AUX(t, C, J)                                  ▷ t ∈ 𝒯, C, J ∈ 𝒫(𝒞)
2:      if isEmpty(J) then
3:          if isLeaf(t) then return LEAF : RESET_F(t.f)
4:          else if isNode(t) then
5:              if isRedundant(t.c, C) then return RESET-AUX(t.l, C, J)
6:              else if isRedundant(¬j, C) then
7:                  return RESET-AUX(t.r, C, J)
8:              else                                             ▷ t.c can be kept in t
9:                  l ← RESET-AUX(t.l, C ∪ {t.c}, J)
10:                 r ← RESET-AUX(t.r, C ∪ {¬t.c}, J)
11:                 return NODE{t.c} : l, r
12:     else if ¬isEmpty(J) then
13:         j ← max J          ▷ j is the largest linear constraint appearing in J
14:         if isLeaf(t) ∨ (isNode(t) ∧ (t.c <_C j ∨ t.c <_C ¬j) then
15:             if isRedundant(j, C) then return RESET-AUX(t, C, J \ {j})
16:             else if isRedundant(¬j, C) then return t
17:             else if ¬j <_C j then                            ▷ j can be added to t
18:                 return NODE{j} : RESET-AUX(t, C ∪ {j}, J \ {j}), t
19:             else if j <_C ¬j then                            ▷ ¬j can be added to t
20:                 return NODE{¬j} : t, RESET-AUX(t, C ∪ {j}, J \ {j})
21:         else if isNode(t) ∧ j <_C t.c ∧ ¬j <_C t.c then
22:             if isRedundant(t.c, C) then return RESET-AUX(t.l, C, J)
23:             else if isRedundant(¬j, C) then
24:                 return RESET-AUX(t.r, C, J)
25:             else                                             ▷ t.c can be kept in t
26:                 l ← RESET-AUX(t.l, C ∪ {t.c}, J)
27:                 r ← RESET-AUX(t.r, C ∪ {¬t.c}, J)
28:                 return NODE{t.c} : l, r
29:         else if isNode(t) ∧ ¬j <_C j then                    ▷ t.c and j are equal
30:             if isRedundant(j, C) then return RESET-AUX(t.l, C, J \ {j})
31:             else if isRedundant(¬j, C) then return t
32:             elsereturn NODE{j} : RESET-AUX(t.l, C ∪ {j}, J \ {j}), t
33:         else if isNode(t) ∧ j <_C ¬j then                    ▷ t.c and ¬j are equal
34:             if isRedundant(j, C) then return RESET-AUX(t.r, C, J \ {j})
35:             else if isRedundant(¬j, C) then return t
36:             else return NODE{¬j} : t, RESET-AUX(t.r, C ∪ {j}, J \ {j})
```

Algorithm 24 : Tree Reset

```
37: function RESET⟦ φ ⟧(p, d, t)                          ▷ p ∈ L, d ∈ D, t ∈ T
38:     switch φ do
39:         case bexp :
40:             switch bexp do
41:                 case ? : return t
42:                 case not bexp :
43:                     return RESET⟦ ¬bexp ⟧(p, d, t)
44:                 case bexp_1 and bexp_2 :
45:                     t_1 ← RESET⟦ bexp_1 ⟧(p, d, t)
46:                     t_2 ← RESET⟦ bexp_2 ⟧(p, d, t)
47:                     return MEET(d, t_1, t_2)
48:                 case bexp_1 or bexp_2 :
49:                     t_1 ← RESET⟦ bexp_1 ⟧(p, d, t)
50:                     t_2 ← RESET⟦ bexp_2 ⟧(p, d, t)
51:                     return A-JOIN(d, t_1, t_2)
52:                 case aexp_1 ⋈ aexp_2 :
53:                     J ← FILTER_C⟦ aexp_1 ⋈ aexp_2 ⟧d
54:                     return RESET-AUX(t, γ_C(d), J)
55:         case l : bexp :
56:             if p ≠ l then return t
57:             else return RESET⟦ bexp ⟧(p, d, t)
58:         case φ_1 and φ_2 :
59:             return MEET(d, RESET⟦ bexp_1 ⟧(p, d, t), RESET⟦ bexp_2 ⟧(p, d, t))
60:         case φ_1 or φ_2 :
61:             return A-JOIN(d, RESET⟦ bexp_1 ⟧(p, d, t), RESET⟦ bexp_2 ⟧(p, d, t))
```

or its negation $\neg j$ (cf. Lines 14-20), or continues the descent along the paths of the decision tree (cf. Lines 21-36). In the first case, RESET-AUX tests j and $\neg j$ for redundancy with respect to C (cf. Lines 15-16): when $\neg j$ is redundant with respect to C the whole decision tree is returned unmodified; otherwise, RESET-AUX adds a decision node for j while leaving unmodified its right subtree (cf. Line 18), if j is already in canonical form (cf. Line 17), or it adds a decision node for for $\neg j$ while leaving unmodified its left subtree (cf. Line 20), if $\neg j$ is the canonical form of j (cf. Line 19). In the second case, RESET-AUX accumulates in C the encountered linear constraints (cf. Lines 26-27), possibly removing redundant decision nodes (cf. Lines 22-24) and leaving the right

$$
\begin{aligned}
&\tau_{\mathrm{g}}^{\varphi\natural}[\![\, {}^{l}\mathtt{skip} \,]\!]t \stackrel{\text{def}}{=} \textsc{reset}_{\mathrm{T}}[\![\, \varphi \,]\!](l, R, \textsc{step}_{\mathrm{T}}(t)) \\
&\tau_{\mathrm{g}}^{\varphi\natural}[\![\, {}^{l}X := aexp \,]\!]t \stackrel{\text{def}}{=} \textsc{reset}_{\mathrm{T}}[\![\, \varphi \,]\!](l, R, (\textsc{b-assign}_{\mathrm{T}}[\![\, X := aexp \,]\!]R)(t)) \\
&\tau_{\mathrm{g}}^{\varphi\natural}[\![\, \mathtt{if}\ {}^{l}bexp\ \mathtt{then}\ stmt_1\ \mathtt{else}\ stmt_2\ \mathtt{fi} \,]\!]t \stackrel{\text{def}}{=} \\
&\qquad \textsc{reset}_{\mathrm{T}}[\![\, \varphi \,]\!](l, R, F_1^{\natural}[t] \curlyvee_{\mathrm{T}} [R]\ F_2^{\natural}[f]) \\
&\qquad F_1^{\natural}[t] \stackrel{\text{def}}{=} (\textsc{filter}_{\mathrm{T}}[\![\, bexp \,]\!]R)(\tau_{\mathrm{g}}^{\varphi\natural}[\![\, stmt_1 \,]\!]t) \\
&\qquad F_2^{\natural}[t] \stackrel{\text{def}}{=} (\textsc{filter}_{\mathrm{T}}[\![\, \mathtt{not}\ bexp \,]\!]R)(\tau_{\mathrm{g}}^{\varphi\natural}[\![\, stmt_2 \,]\!]t) \\
&\tau_{\mathrm{g}}^{\varphi\natural}[\![\, \mathtt{while}\ {}^{l}bexp\ \mathtt{do}\ stmt\ \mathtt{od} \,]\!]t \stackrel{\text{def}}{=} \mathrm{lfp}^{\natural}\ \phi_{\mathrm{g}}^{\varphi\natural} \\
&\qquad \phi_{\mathrm{g}}^{\varphi\natural}(x) \stackrel{\text{def}}{=} \textsc{reset}_{\mathrm{T}}[\![\, \varphi \,]\!](l, R, F^{\natural}[x] \curlyvee_{\mathrm{T}} [R]\ (\textsc{filter}_{\mathrm{T}}[\![\, \mathtt{not}\ bexp \,]\!]R)(t)) \\
&\qquad F^{\natural}[x] \stackrel{\text{def}}{=} (\textsc{filter}_{\mathrm{T}}[\![\, bexp \,]\!]R)(\tau_{\mathrm{g}}^{\varphi\natural}[\![\, stmt \,]\!]x) \\
&\tau_{\mathrm{g}}^{\varphi\natural}[\![\, stmt_1\ stmt_2 \,]\!]t \stackrel{\text{def}}{=} \tau_{\mathrm{g}}^{\varphi\natural}[\![\, stmt_1 \,]\!](\tau_{\mathrm{g}}^{\varphi\natural}[\![\, stmt_2 \,]\!]t)
\end{aligned}
$$

Figure 9.1: Abstract guarantee semantics of instructions *stmt*.

or left decision subtree unmodified when the encountered linear constraints coincide with those appearing in J (cf. Lines 29-32) or their negations (cf. Lines 33-36). When J is empty (cf. Line 2), RESET-AUX accumulates in C the linear constraints encountered along the paths (cf. Lines 9-10) up to a leaf node, possibly removing constraints that are redundant (cf. Line 5) or whose negation is redundant (cf. Line 6) with respect to C.

Once reached a leaf node, the operator $\textsc{reset}_{\mathrm{F}}[\![\, \varphi \,]\!] \colon \mathcal{F} \to \mathcal{F}$ is invoked (cf. Line 3): given a function $f \in \mathcal{F}$, it simply resets its value to zero:

$$\textsc{reset}_{\mathrm{F}}(f) \stackrel{\text{def}}{=} \lambda X_1, \ldots, X_k.\ 0 \tag{9.2.14}$$

Abstract Guarantee Semantics. The operators of the decision trees abstract domain can now be used to define the abstract guarantee semantics.

In the following, as in Section 5.3 we assume to have, for each program control point $l \in \mathcal{L}$, a sound numerical over-approximation $R \in \mathcal{D}$ of the reachable environments $\tau_{\mathrm{I}}(l) \in \mathcal{P}(\mathcal{E})$: $\tau_{\mathrm{I}}(l) \subseteq \gamma_{\mathrm{D}}(R)$ (cf. Section 3.4).

In Figure 9.1 we define the semantics $\tau_{\mathrm{g}}^{\varphi\natural}[\![\, stmt \,]\!] \colon \mathcal{T} \to \mathcal{T}$, for each program instruction *stmt*. Each function $\tau_{\mathrm{g}}^{\varphi\natural}[\![\, stmt \,]\!] \colon \mathcal{T} \to \mathcal{T}$ takes as input a decision tree over-approximating the ranking function corresponding to the final control point of the instruction, and outputs a decision tree defined over a subset of the reachable environments $R \in \mathcal{D}$, which over-approximates the ranking function corresponding to the initial control point of the instruction. Note that each function $\tau_{\mathrm{g}}^{\varphi\natural}[\![\, stmt \,]\!]$ invokes the reset operator $\textsc{reset}_{\mathrm{T}}$. For a `while` loop,

lfp$^\natural$ $\phi_{\mathrm{g}}^{\varphi\natural}$ is the limit of the iteration sequence with widening:

$$\begin{aligned} y_0 &\stackrel{\text{def}}{=} \bot_{\mathrm{T}} \\ y_{n+1} &\stackrel{\text{def}}{=} \begin{cases} y_n & \phi_{\mathrm{g}}^{\varphi\natural}(y_n) \sqsubseteq_{\mathrm{T}} [R]\ y_n \wedge \phi_{\mathrm{g}}^{\varphi\natural}(y_n) \preccurlyeq_{\mathrm{T}} [R]\ y_n \\ y_n \nabla_{\mathrm{T}}\ \phi_{\mathrm{g}}^{\varphi\natural}(y_n) & \text{otherwise} \end{cases} \end{aligned} \tag{9.2.15}$$

In absence of run-time errors, the abstract semantics $\tau_{\mathrm{g}}^{\varphi\natural}[\![\, stmt \,]\!]$, given sound over-approximations $R \in \mathcal{D}$ of $\tau_{\mathrm{I}}(i[\![\, stmt \,]\!])$ and $D \in \mathcal{D}$ of $\tau_{\mathrm{I}}(f[\![\, stmt \,]\!])$, is a sound over-approximation of the semantics $\tau_{\mathrm{g}}^{\varphi}[\![\, stmt \,]\!]$ defined in Section 9.2.2:

Lemma 9.2.9 $\tau_g^{\varphi}[\![\, stmt \,]\!]\gamma_T[D]t \ \preccurlyeq\ \gamma_T[R]\tau_g^{\varphi\natural}[\![\, stmt \,]\!]t.$

Proof.
See Appendix A.6. ■

The abstract guarantee semantics $\tau_{\mathrm{g}}^{\varphi\natural}[\![\, prog \,]\!] \in \mathcal{T}$ of a program *prog* outputs the decision tree over-approximating the ranking function corresponding to the initial program control point $i[\![\, prog \,]\!] \in \mathcal{L}$. It is defined by taking as input the leaf node $\mathrm{LEAF} : \bot_{\mathrm{F}}$ as:

Definition 9.2.10 (Abstract Guarantee Semantics) *The* abstract guarantee semantics $\tau_g^{\varphi\natural}[\![\, prog \,]\!] \in \mathcal{T}$ *of a program prog is:*

$$\tau_g^{\varphi\natural}[\![\, prog \,]\!] = \tau_g^{\varphi\natural}[\![\, stmt^{\ l} \,]\!] \stackrel{def}{=} \tau_g^{\varphi\natural}[\![\, stmt \,]\!]\mathrm{RESET}_T[\![\, \varphi \,]\!](l, R, \mathrm{LEAF} : \bot_F) \tag{9.2.16}$$

where the abstract guarantee semantics $\tau_g^{\varphi\natural}[\![\, stmt \,]\!] : \mathcal{T} \to \mathcal{T}$ *of each program instruction stmt is defined in Figure 9.1.*

In absence of run-time errors, the following result proves the soundness of the abstract guarantee semantics $\tau_{\mathrm{g}}^{\varphi\natural}[\![\, prog \,]\!] \in \mathcal{T}$ with respect to the guarantee semantics $\tau_{\mathrm{g}}^{\varphi}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$, given a sound numerical over-approximation $R \in \mathcal{D}$ of the reachable environments $\tau_{\mathrm{I}}(i[\![\, prog \,]\!])$:

Theorem 9.2.11 $\tau_g^{\varphi}[\![\, prog \,]\!] \preccurlyeq \gamma_T[R]\tau_g^{\varphi\natural}[\![\, prog \,]\!]$

Proof (Sketch).
The proof follows from the soundness of the operators of the decision trees abstract domain (cf. Lemma 9.2.9) used for the definition of $\tau_{\mathrm{g}}^{\varphi\natural}[\![\, prog \,]\!] \in \mathcal{T}$. ■

In particular, the abstract guarantee semantics provides *sufficient preconditions* for ensuring a guarantee property $\Diamond\varphi$ for a given over-approximation $R \in \mathcal{D}$ of the set of initial states $\mathcal{I} \subseteq \Sigma$:

Corollary 9.2.12 *A program satisfies a guarantee property $\Diamond\varphi$ for execution traces starting from a set of states $\gamma_D(R)$ if $\gamma_D(R) \subseteq \mathrm{dom}(\gamma_T[R]\tau_g^{\varphi\natural}[\![\, prog \,]\!])$.*

9.3 Recurrence Semantics

In the following, along the same approach used in the previous Section 9.2 for guarantee properties, we define a sound and complete semantics for proving recurrence temporal properties by abstract interpretation of the program maximal trace semantics (cf. Equation 2.2.5). Then, we propose a sound decidable abstraction based on piecewise-defined ranking functions.

9.3.1 Recurrence Semantics

The *recurrence semantics*, given a set of desirable states $S \subseteq \Sigma$, is a ranking function $\tau_{\mathrm{r}}[S] \in \Sigma \rightharpoonup \mathbb{O}$ defined starting from the states in S, where the function has value zero, and retracing the program *backwards* while mapping every state in Σ definitely leading *infinitely often* to a state in S (i.e., a state such that all the traces to which it belongs reach a state in S infinitely often) to an ordinal in $\mathbb{O}$ representing an upper bound on the number of program execution steps remaining to the next state in S. The domain $\mathrm{dom}(\tau_{\mathrm{r}}[S])$ of $\tau_{\mathrm{r}}[S]$ is the set of states definitely leading infinitely often to a desirable state in S: all traces branching from a state $s \in \mathrm{dom}(\tau_{\mathrm{r}}[S])$ reach the next state in S in at most $\tau_{\mathrm{r}}[S]s$ execution steps, while at least one trace branching from a state $s \notin \mathrm{dom}(\tau_{\mathrm{g}}[S])$ never reaches S.

In particular, the recurrence semantics reuses the guarantee semantics of Section 9.2 as a building block: from the guarantee that some desirable event happens once during program execution, the recurrence semantics ensures that the event happens infinitely often. A finite subsequence of a program trace satisfies a recurrence property if and only if it terminates in the desirable set of states, and its neighborhood in the subsequences of the program semantics consists only of sequences that are terminating in the desirable set of states, and that are *prefixes* of traces in the program semantics that reach infinitely often the desirable set of states. The corresponding *recurrence abstraction* $\alpha^{\mathrm{r}}[S] \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^{+})$ is parameterized by a set of desirable states $S \subseteq \Sigma$

and it is defined as follows:

$$\begin{aligned} \alpha^{\mathrm{r}}[S]T &\stackrel{\text{def}}{=} \mathrm{gfp}^{\subseteq}_{\alpha^{\mathrm{g}}[S]T}\, \psi^{\mathrm{r}}[T,S] \\ \psi^{\mathrm{r}}[T,S]T', &\stackrel{\text{def}}{=} \alpha^{\mathrm{g}}[\widetilde{\mathrm{pre}}[T]T' \cap S]T \end{aligned} \tag{9.3.1}$$

where $\widetilde{\mathrm{pre}}[T]T' \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall \sigma \in \Sigma^*, \sigma' \in \Sigma^{*\infty} : \sigma s \sigma' \in T \Rightarrow \mathrm{pf}(\sigma') \cap T' \neq \emptyset\}$ is the set of states whose successors all belong to a given set of subsequences, and $\alpha^{\mathrm{g}}[S] \colon \mathcal{P}(\Sigma^{+\infty}) \to \mathcal{P}(\Sigma^+)$ is the guarantee abstraction of Equation 9.2.2.

To explain intuitively Equation 9.3.1, we use the dual of Kleene's Fixpoint Theorem (cf. Theorem 2.1.4) to rephrase $\alpha^{\mathrm{r}}[S]$ as follows:

$$\begin{aligned} \alpha^{\mathrm{r}}[S]T &= \bigcap_{i \in \mathbb{N}} T_{i+1} \\ T_{i+1} &\stackrel{\text{def}}{=} [\psi^{\mathrm{r}}[T,S]]^i\, (\alpha^{\mathrm{g}}[S]T) \end{aligned}$$

Then, for $i = 0$, we get the set $T_1 = \alpha^{\mathrm{g}}[S]T$ of subsequences of T that guarantee S at least *once*. For $i = 1$, starting from T_1, we derive the set of states $S_1 = \widetilde{\mathrm{pre}}[T]T_1 \cap S$ (i.e., $S_1 \subseteq S$) whose successors all belong to the subsequences in T_1, and we get the set $T_2 = \alpha^{\mathrm{g}}[S_1]T$ of subsequences of T that guarantee S_1 at least once and thus guarantee S at least *twice*. Note that all the subsequences in T_2 terminate with a state $s \in S_1$ and therefore are *prefixes* of subsequence of T that reach S at least twice. More generally, for each $i \in \mathbb{N}$, we get the set T_{i+1} of subsequences which are *prefixes* of subsequences of T that reach S at least $i+1$ times, i.e., the subsequences that guarantee S at least $i+1$ times. The greatest fixpoint thus guarantees S *infinitely often*.

Example 9.3.1

Let $T \stackrel{\text{def}}{=} \{(cd)^\omega, ca^\omega, d(be)^\omega\}$ and let $S \stackrel{\text{def}}{=} \{b, c, d\}$. For $i = 0$, we have $T_1 = \alpha^{\mathrm{g}}[S]T = \{b, eb, c, d\}$. For $i = 1$, we derive $S_1 = \{b, d\}$, since $c(dc)^\omega \in T$ and $\mathrm{pf}((dc)^\omega) \cap T_1 = \{d\} \neq \emptyset$ but $ca^\omega \in T$ and $\mathrm{pf}(a^\omega) \cap T_1 = \emptyset$. We get $T_2 = \alpha^{\mathrm{g}}[S_1]T = \{b, eb, d\}$. For $i = 2$, we derive $S_2 = \{b\}$, since $d(be)^\omega \in T$ and $\mathrm{pf}((be)^\omega) \cap T_1 = \{b\} \neq \emptyset$ but $d(cd)^\omega \in T$ and $\mathrm{pf}((cd)^\omega) \cap T_2 = \emptyset$. We get $T_3 = \alpha^{\mathrm{g}}[S_2]T = \{b, eb\}$ which is the greatest fixpoint: the only subsequences of sequences in T that guarantee S infinitely often start with b or eb.

We can now define the recurrence semantics $\tau_{\mathrm{r}}[S] \in \Sigma \rightharpoonup \mathbb{O}$:

Definition 9.3.2 (Recurrence Semantics) *Given a desirable set of states $S \subseteq \Sigma$, the* recurrence semantics $\tau_r[S] \in \Sigma \rightharpoonup \mathbb{O}$ *is an abstract interpretation of the maximal trace semantics $\tau^{+\infty} \in \mathcal{P}(\Sigma^{+\infty})$ (cf. Equation 2.2.5):*

$$\tau_r[S] \stackrel{def}{=} \alpha^{Mrk}(\alpha^r[S](\tau^{+\infty})) \tag{9.3.2}$$

where the abstraction $\alpha^{\mathrm{Mrk}} \colon \mathcal{P}(\Sigma^+) \to (\Sigma \rightharpoonup \mathbb{O})$ *is the same ranking abstraction defined in Equation 4.2.14.*

The following result provides a fixpoint definition of the recurrence semantics within the partially ordered set $\langle \Sigma \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$, where the computational order is defined as in Equation 4.2.15 as:

$$f_1 \sqsubseteq f_2 \iff \mathrm{dom}(f_1) \subseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_1) : f_1(x) \leq f_2(x).$$

Theorem 9.3.3 (Recurrence Semantics) *Given a desirable set of states* $S \subseteq \Sigma$*, the recurrence semantics* $\tau_r[S] \in \Sigma \rightharpoonup \mathbb{O}$ *can be expressed as a greatest fixpoint in the partially ordered set* $\langle \Sigma \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$*:*

$$\begin{aligned} \tau_r[S] &= \mathrm{gfp}^{\sqsubseteq}_{\tau_g[S]}\ \phi_r[S] \\ \phi_r[S]f &\stackrel{def}{=} \lambda s. \begin{cases} f(s) & s \in \mathrm{dom}(\tau_g[\widetilde{\mathrm{pre}}(\mathrm{dom}(f)) \cap S]) \\ \text{undefined} & \text{otherwise} \end{cases} \end{aligned} \tag{9.3.3}$$

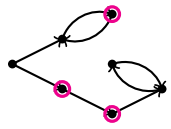
Note that, the recurrence semantics can be equivalently simplified as:

$$\begin{aligned} \tau_{\mathrm{r}}[S] &= \mathrm{gfp}^{\sqsubseteq}_{\tau_{\mathrm{g}}[S]}\ \phi_{\mathrm{r}}[S] \\ \phi_{\mathrm{r}}[S]f &\stackrel{\mathrm{def}}{=} \lambda s. \begin{cases} f(s) & s \in \widetilde{\mathrm{pre}}(\mathrm{dom}(f)) \\ \text{undefined} & \text{otherwise} \end{cases} \end{aligned} \tag{9.3.4}$$

Indeed, there is not need to redefine $\tau_{\mathrm{g}}[S]$ at each iterate since the various subsequences of a program traces manipulated by the recurrence abstraction (cf. Equation 9.3.1) have been abstracted into program states. The rest of the chapter refers to this simplified version of the recurrence semantics.

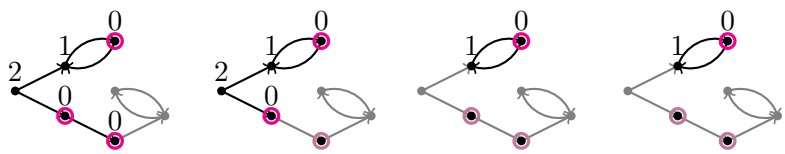
Example 9.3.4

Let us consider the following trace semantics:



where the highlighted states are the set S of desirable states.

The fixpoint iterates of the recurrence semantics $\tau_{\mathrm{r}}[S] \in \Sigma \rightharpoonup \mathbb{O}$ are:



where unlabelled states are outside the domain of the function.

Let φ be a state property. The *φ-recurrence semantics* $\tau_{\mathrm{r}}^{\varphi} \in \Sigma \rightharpoonup \mathbb{O}$:

$$\tau_{\mathrm{r}}^{\varphi} \stackrel{\mathrm{def}}{=} \tau_{\mathrm{r}}[\varphi] \tag{9.3.5}$$

is sound and complete for proving a recurrence property $\Box\Diamond\varphi$:

Theorem 9.3.5 *A program satisfies a recurrence property $\Box\Diamond\varphi$ for execution traces starting from a given set of states $\mathcal{I}$ if and only if $\mathcal{I} \subseteq \mathrm{dom}(\tau_r^{\varphi})$.*

Proof.
See Appendix A.6. ■

9.3.2 Denotational Recurrence Semantics

We now provide a structural definition of the φ-recurrence semantics $\tau_{\mathrm{r}}^{\varphi} \in \Sigma \rightharpoonup \mathbb{O}$ by induction on the syntax of our idealized imperative language.

We partition $\tau_{\mathrm{r}}^{\varphi}$ with respect to the program control points: $\tau_{\mathrm{r}}^{\varphi} \in \mathcal{L} \to (\mathcal{E} \rightharpoonup \mathbb{O})$. In this way, to each program control point $l \in \mathcal{L}$ corresponds a partial function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$, and to each program instruction *stmt* corresponds a recurrence semantics $\tau_{\mathrm{r}}^{\varphi}[\![\, \mathit{stmt} \,]\!]\colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$.

The recurrence semantics $\tau_{\mathrm{r}}^{\varphi}[\![\, \mathit{stmt} \,]\!]\colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ of each program instruction *stmt* takes as input a ranking function whose domain represents the environments always leading *infinitely often* to φ from the final label of *stmt*, and determines the ranking function whose domain represents the environments always leading *infinitely often* to φ from the initial label of *stmt*, and whose value represents an upper bound on the number of program execution steps remaining to the next occurrence of φ. In particular, each function $\tau_{\mathrm{r}}^{\varphi}[\![\, \mathit{stmt} \,]\!] \in (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ behaves as described in Section 9.2.2 and also ensures that each time φ is satisfied, it will be satisfied again in the future: the value of the input ranking function is reset for the environments that satisfy φ only if all their successors by means of the instruction *stmt* belong to the domain of the input ranking function.

The recurrence semantics of a `skip` instruction resets the value of the input ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$ for the environments that belong to its domain and satisfy φ, and otherwise it increases its value to take into account another

program execution step from the initial label of the instruction:

$$\tau_{\mathrm{r}}^{\varphi}[\![\,^{l}\mathtt{skip}\,]\!]f \stackrel{\text{def}}{=} \lambda\rho.\begin{cases} 0 & \langle l,\rho\rangle \models \varphi \wedge \rho \in \mathrm{dom}(f) \\ f(\rho)+1 & \langle l,\rho\rangle \not\models \varphi \wedge \rho \in \mathrm{dom}(f) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{9.3.6}$$

Similarly, the recurrence semantics of a variable assignment $^{l}X := aexp$ is defined over the environments that when subject to the assignment always belong to the domain of the input ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$. The value of the input ranking function for these environments is reset when they satisfy φ; otherwise, it is increased by one to account for another execution step, and the value of the resulting ranking function is the least upper bound of these values:

$$\tau_{\mathrm{r}}^{\varphi}[\![\,^{l}X := aexp\,]\!]f \stackrel{\text{def}}{=} \lambda\rho.\begin{cases} 0 & \langle l,\rho\rangle \models \varphi \wedge \exists v \in [\![aexp]\!]\rho \wedge \\ & \qquad \forall v \in [\![aexp]\!]\rho : \rho[X \leftarrow v] \in \mathrm{dom}(f) \\ \sup\{f(\rho[X \leftarrow v])+1 \mid v \in [\![aexp]\!]\rho\} & \\ & \langle l,\rho\rangle \not\models \varphi \wedge \exists v \in [\![aexp]\!]\rho \wedge \\ & \qquad \forall v \in [\![aexp]\!]\rho : \rho[X \leftarrow v] \in \mathrm{dom}(f) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{9.3.7}$$

Example 9.3.6

Let $\mathcal{X} \stackrel{\text{def}}{=} \{x\}$. We consider the following ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$:

$$f(\rho) \stackrel{\text{def}}{=} \begin{cases} 2 & \rho(x) = 0 \\ 3 & \rho(x) = 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

the backward assignment $x{:=}x + [1,2]$ and the recurrence property $\Box\Diamond(x = 3)$. The recurrence semantics of the assignment is:

$$\tau_{\mathrm{r}}^{x=3}[\![\,x := x + [1,2]\,]\!]f(\rho) \stackrel{\text{def}}{=} \begin{cases} 4 & \rho(x) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

In particular, note that unlike Example 9.2.6, the function is not defined when $\rho(x) = 3$, since the environment satisfies the property $x = 3$ but $[\![x+[1,2]]\!]\rho = \{4,5\}$ and $\rho[x \leftarrow 4] \notin \mathrm{dom}(f)$ and $\rho[x \leftarrow 5] \notin \mathrm{dom}(f)$.

Given a conditional instruction `if` ^{l}*bexp* `then` *stmt*$_1$ `else` *stmt*$_2$ `fi`, its recurrence semantics takes as input a ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$ and derives the recurrence semantics $\tau_{\mathrm{r}}^{\varphi}[\![\, \mathit{stmt}_1 \,]\!] f$ of *stmt*$_1$, in the following denoted by S_1, and the recurrence semantics $\tau_{\mathrm{r}}^{\varphi}[\![\, \mathit{stmt}_2 \,]\!] f$ of *stmt*$_2$, in the following denoted by S_2. Then, the recurrence semantics of the conditional instruction is defined by means of the ranking function $F[f]\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments that belong to the domain of S_1 and to the domain of S_2, and that may both satisfy and not satisfy the boolean expression *bexp*:

$$F[f] \stackrel{\text{def}}{=} \lambda\rho \in \mathrm{dom}(S_1) \cap \mathrm{dom}(S_2).\ \begin{cases} \sup\{S_1(\rho)+1, S_2(\rho)+1\} & \\ & [\![\mathit{bexp}]\!]\rho = \{\mathrm{true}, \mathrm{false}\} \\ \mathrm{undefined} & \mathrm{otherwise} \end{cases}$$

and the ranking function $F_1[f]\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments $\rho \in \mathcal{E}$ that belong to the domain of S_1 and that must satisfy *bexp*:

$$F_1[f] \stackrel{\text{def}}{=} \lambda\rho \in \mathrm{dom}(S_1).\ \begin{cases} S_1(\rho)+1 & [\![\mathit{bexp}]\!]\rho = \{\mathrm{true}\} \\ \mathrm{undefined} & \mathrm{otherwise} \end{cases}$$

and the ranking function $F_2[f]\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments that belong to the domain of S_2 and that cannot satisfy *bexp*:

$$F_2[f] \stackrel{\text{def}}{=} \lambda\rho \in \mathrm{dom}(S_2).\ \begin{cases} S_2(\rho)+1 & [\![\mathit{bexp}]\!]\rho = \{\mathrm{false}\} \\ \mathrm{undefined} & \mathrm{otherwise} \end{cases}$$

The resulting ranking function is defined by joining $F[f]$, $F_1[f]$, and $F_2[f]$, and resetting the value of the function for the environments that belong to its domain and satisfy φ:

$$\tau_{\mathrm{r}}^{\varphi}[\![\, \texttt{if}\ {}^{l}\mathit{bexp}\ \texttt{then}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2\ \texttt{fi} \,]\!] f \stackrel{\text{def}}{=} \lambda\rho.\ \begin{cases} 0 & \langle l, \rho\rangle \models \varphi \wedge \\ & \rho \in \mathrm{dom}(R) \\ R(\rho) & \langle l, \rho\rangle \not\models \varphi \wedge \\ & \rho \in \mathrm{dom}(R) \\ \mathrm{undefined} & \mathrm{otherwise} \end{cases} \tag{9.3.8}$$

where $R \stackrel{\text{def}}{=} F[f] \mathbin{\dot\cup} F_1[f] \mathbin{\dot\cup} F_2[f]$.

Example 9.3.7

Let $\mathcal{X} \stackrel{\text{def}}{=} \{x\}$. We consider the recurrence property $\Box\Diamond(x = 3)$ and the recurrence semantics of the conditional statement `if` *bexp* `then` *stmt*$_1$ `else` *stmt*$_2$ `fi`.

We assume, given a ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$, that the recurrence semantics of $stmt_1$ is defined as:

$$\tau_{\mathrm{r}}^{x=3}[\![\, stmt_1 \,]\!] f(\rho) \stackrel{\text{def}}{=} \begin{cases} 1 & \rho(x) \leq 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

and that the recurrence semantics of $stmt_2$ is defined as

$$\tau_{\mathrm{r}}^{x=3}[\![\, stmt_2 \,]\!] f(\rho) \stackrel{\text{def}}{=} \begin{cases} 3 & 0 \leq \rho(x) < 3 \\ 0 & \rho(x) = 3 \\ 3 & 3 < \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Then, when the boolean expression *bexp* is for example $x \leq 3$, the recurrence semantics of the conditional statement is:

$$\tau_{\mathrm{r}}^{x=3}[\![\, \texttt{if}\ {}^{l}bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi} \,]\!] f(\rho) \stackrel{\text{def}}{=} \begin{cases} 2 & \rho(x) \leq 0 \\ 4 & 3 < \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Instead, when *bexp* is for example the non-deterministic choice `?`, we have:

$$\tau_{\mathrm{r}}^{x=3}[\![\, \texttt{if}\ {}^{l}bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi} \,]\!] f(\rho) \stackrel{\text{def}}{=} \begin{cases} 4 & \rho(x) = 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that, like Example 4.3.2 and unlike Example 9.2.7, both functions are undefined when $\rho(x) = 3$, even though the property $x = 3$ is satisfied. In fact, the ranking function for the `then` branch of the `if` is undefined when $\rho(x) = 3$.

The recurrence semantics of a loop instruction `while` ${}^{l}bexp$ `do` *stmt* `od` takes as input a ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain represents the environments leading infinitely often to φ from the final label of the instruction, and outputs the ranking function which is defined as a greatest fixpoint of the function $\phi_{\mathrm{r}}^{\varphi}\colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ within $\langle \mathcal{E} \rightharpoonup \mathbb{O}, \sqsubseteq \rangle$:

$$\tau_{\mathrm{r}}^{\varphi}[\![\, \texttt{while}\ {}^{l}bexp\ \texttt{do}\ stmt\ \texttt{od} \,]\!] f \stackrel{\text{def}}{=} \mathrm{gfp}_{G}^{\sqsubseteq}\ \phi_{\mathrm{r}}^{\varphi} \tag{9.3.9}$$

where $G \stackrel{\text{def}}{=} \tau_{\mathrm{g}}^{\varphi}[\![\, \texttt{while}\ {}^{l}bexp\ \texttt{do}\ stmt\ \texttt{od} \,]\!] f$ is the guarantee semantics of the loop instruction defined in Equation 9.2.10, and the *computational order* is defined as in Equation 4.2.15:

$$f_1 \sqsubseteq f_2 \iff \mathrm{dom}(f_1) \subseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_1) : f_1(x) \leq f_2(x).$$

In essence, from the guarantee that some desirable event eventually happens, the recurrence semantics ensures that the event happens infinitely often. The function $\phi_{\mathrm{r}}^{\varphi}\colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ takes as input a ranking function $x\colon \mathcal{E} \rightharpoonup \mathbb{O}$, resets its value for the environments that belong to its domain and that satisfy φ, and adds to its domain the environments for which one more loop iteration is needed before the next occurrence of φ. In the following, the recurrence semantics $\tau_{\mathrm{r}}^{\varphi}[\![\, \mathit{stmt} \,]\!]x$ of the loop body is denoted by S. The function $\phi_{\mathrm{r}}^{\varphi}$ is defined by means of the ranking function $F[x]\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments that belong to the domain of S and to the domain of the input function f, and that may both satisfy and not satisfy the boolean expression *bexp*:

$$F[x] \stackrel{\text{def}}{=} \lambda\rho \in \mathrm{dom}(S) \cap \mathrm{dom}(f).\ \begin{cases} \sup\{S(\rho)+1, f(\rho)+1\} & \\ & [\![\mathit{bexp}]\!]\rho = \{\text{true}, \text{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

and the ranking function $F_1[x]\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments $\rho \in \mathcal{E}$ that belong to the domain of S and that must satisfy *bexp*:

$$F_1[x] \stackrel{\text{def}}{=} \lambda\rho \in \mathrm{dom}(S).\ \begin{cases} S(\rho)+1 & [\![\mathit{bexp}]\!]\rho = \{\text{true}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

and the ranking function $F_2[f]\colon \mathcal{E} \rightharpoonup \mathbb{O}$ whose domain is the set of environments that belong to the domain of the input function f and that cannot satisfy *bexp*:

$$F_2[f] \stackrel{\text{def}}{=} \lambda\rho \in \mathrm{dom}(f).\ \begin{cases} f(\rho)+1 & [\![\mathit{bexp}]\!]\rho = \{\text{false}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The resulting ranking function is defined by joining $F[x]$, $F_1[x]$, and $F_2[f]$, and resetting its value for the environments that belong to its domain and satisfy φ:

$$\phi_{\mathrm{r}}^{\varphi}(x) \stackrel{\text{def}}{=} \lambda\rho.\ \begin{cases} 0 & \langle l, \rho\rangle \models \varphi \wedge \rho \in \mathrm{dom}(R) \\ R(\rho) & \langle l, \rho\rangle \not\models \varphi \wedge \rho \in \mathrm{dom}(R) \\ \text{undefined} & \text{otherwise} \end{cases} \tag{9.3.10}$$

where $R \stackrel{\text{def}}{=} F[x] \mathbin{\dot\cup} F_1[x] \mathbin{\dot\cup} F_2[f]$.

Finally, the recurrence semantics of the sequential combination of instructions $\mathit{stmt}_1\ \mathit{stmt}_2$, takes as input a ranking function $f\colon \mathcal{E} \rightharpoonup \mathbb{O}$, determines from f the recurrence semantics $\tau_{\mathrm{r}}^{\varphi}[\![\, \mathit{stmt}_2 \,]\!]f$ of stmt_2, and outputs the ranking function determined by the recurrence semantics of stmt_1 from $\tau_{\mathrm{r}}^{\varphi}[\![\, \mathit{stmt}_2 \,]\!]f$:

$$\tau_{\mathrm{r}}^{\varphi}[\![\, \mathit{stmt}_1\ \mathit{stmt}_2 \,]\!]f \stackrel{\text{def}}{=} \tau_{\mathrm{r}}^{\varphi}[\![\, \mathit{stmt}_1 \,]\!](\tau_{\mathrm{r}}^{\varphi}[\![\, \mathit{stmt}_2 \,]\!]f) \tag{9.3.11}$$

The recurrence semantics $\tau_{\mathrm{r}}^{\varphi}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$ of a program *prog* is a ranking function whose domain represents the environments always leading *infinitely often* to φ, which is determined by taking as input the totally undefined function, since the program final states cannot satisfy a recurrence property:

Definition 9.3.8 (Recurrence Semantics) *The* recurrence semantics $\tau_{r}^{\varphi}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$ *of a program prog is:*

$$\tau_{r}^{\varphi}[\![\, prog \,]\!] = \tau_{r}^{\varphi}[\![\, stmt^{\,l} \,]\!] \stackrel{def}{=} \tau_{r}^{\varphi}[\![\, stmt \,]\!]\dot{\emptyset} \tag{9.3.12}$$

where the function $\tau_{r}^{\varphi}[\![\, stmt \,]\!] \colon (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ *is the recurrence semantics of each program instruction stmt.*

As pointed out in Remark 3.2.2, possible run-time errors are ignored. Thus, all environments leading to run-time errors are discarded and do not belong to the domain of the recurrence semantics of a program *prog*.

9.3.3 Abstract Recurrence Semantics

We now propose a sound decidable abstraction of the recurrence semantics $\tau_{\mathrm{r}}^{\varphi}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$, based on the piecewise-defined ranking functions presented in Chapter 5 and Chapter 6. The abstraction is sound with respect to the usual *approximation order* defined in Equation 4.2.12:

$$f_1 \preccurlyeq f_2 \iff \mathrm{dom}(f_1) \supseteq \mathrm{dom}(f_2) \wedge \forall x \in \mathrm{dom}(f_2) : f_1(x) \leq f_2(x).$$

In particular, we revisit the definition of the unary operator $\textsc{reset}_{\mathrm{T}}[\![\, \varphi \,]\!]$ and we introduce a *dual widening* operator $\bar{\nabla}_{\mathrm{T}}$.

Reset. The operator $\textsc{reset}_{\mathrm{T}}[\![\, \varphi \,]\!] \colon \mathcal{T} \to \mathcal{T}$ should reset the leaves of a decision tree that not only satisfy a given property φ but also guarantee that the property will be satisfied again in the future. To this end, we redefine the operator $\textsc{reset}_{\mathrm{F}}[\![\, \varphi \,]\!] \colon \mathcal{F} \to \mathcal{F}$ invoked when Algorithm 24 reaches a leaf node (cf. Line 3): given a function $f \in \mathcal{F} \setminus \{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}$, $\textsc{reset}_{\mathrm{F}}[\![\, \varphi \,]\!]$ simply resets its value to zero; undefined leaf nodes are instead left unaltered:

$$\begin{aligned} \textsc{reset}_{\mathrm{F}}(\bot_{\mathrm{F}}) &\stackrel{\mathrm{def}}{=} \bot_{\mathrm{F}} \\ \textsc{reset}_{\mathrm{F}}(f) &\stackrel{\mathrm{def}}{=} \lambda X_1, \ldots, X_k.\ 0 \\ \textsc{reset}_{\mathrm{F}}(\top_{\mathrm{F}}) &\stackrel{\mathrm{def}}{=} \top_{\mathrm{F}} \end{aligned} \tag{9.3.13}$$

Algorithm 25 : Tree Dual Widening

```
1:  function DUAL-WIDEN-AUX(t1, t2, C)                    ▷ t1, t2 ∈ T, C ∈ P(C)
2:      if ISLEAF(t1) ∧ ISLEAF(t2) then
3:          if t2.f ⊑F [αC(C)] t1.f then return t2.f
4:          else return LEAF : ⊥F
5:      else if ISNODE(t1) ∧ ISNODE(t2) then
6:          l ← DUAL-WIDEN-AUX(t1.l, t2.l, C ∪ {t2.c})
7:          r ← DUAL-WIDEN-AUX(t1.r, t2.r, C ∪ {¬t2.c})
8:          return NODE{t2.c} : l; r
9:
10: function DUAL-WIDEN(d,t1,t2)                          ▷ d ∈ D, t1, t2 ∈ T
11:     (t1, t2) ← LEFT-UNIFICATION(ϒF, d, t1, t2)        ▷ domain widening
12:     return DUAL-WIDEN-AUX(t1, t2, γC(d))              ▷ value widening
```

Dual Widening. The recurrence semantics of a `while` loop instruction, as defined in Equation 9.3.9, involves a greatest fixpoint. Greatest fixpoints are solved by iterations using a new *dual widening* operator $\bar{\nabla}_{\mathrm{T}}$.

The dual widening $\bar{\nabla}_{\mathrm{T}}$ is implemented by Algorithm 25: the function DUAL-WIDEN, given a given a sound over-approximation $d \in \mathcal{D}$ of the reachable environments and two decision trees $t_1, t_2 \in \mathcal{T}$, calls the function LEFT-UNIFICATION (cf. Line 11) to limit the size of the decision trees in order to ensure convergence. Unlike Algorithm 13, Algorithm 25 invokes LEFT-UNIFICATION choosing the approximation join $\curlyvee_{\mathrm{F}}$. Then, Algorithm 25 calls the auxiliary function DUAL-WIDEN-AUX, which collects into a set $C \in \mathcal{P}(\mathcal{C})$ (initially equal to $\gamma_{\mathrm{C}}(d)$, cf. Line 12) the linear constraints encountered along the paths up to the leaf nodes (cf. Lines 6-7), which are compared (cf. Line 3) and, in case, turned into a $\bot_{\mathrm{F}}$-leaf (cf. Line 4).

In Figure 9.2 we depict an example of dual widening. In essence, the dual widening maintains the value of a piecewise-defined ranking function only on the pieces where it stays defined between two iterate, and if a piece where it is defined shrinks, we remove it entirely.

Abstract Recurrence Semantics. In the following, we assume to have, for each program control point $l \in \mathcal{L}$, a sound numerical over-approximation $R \in \mathcal{D}$ of the reachable environments $\tau_{\mathrm{I}}(l) \in \mathcal{P}(\mathcal{E})$: $\tau_{\mathrm{I}}(l) \subseteq \gamma_{\mathrm{D}}(R)$ (cf. Section 3.4).

In Figure 9.3 we define the semantics $\tau_{\mathrm{r}}^{\varphi\natural}[\![\, stmt \,]\!] \colon \mathcal{T} \to \mathcal{T}$, for each program instruction *stmt*. Each function $\tau_{\mathrm{r}}^{\varphi\natural}[\![\, stmt \,]\!] \colon \mathcal{T} \to \mathcal{T}$ takes as input a decision tree over-approximating the ranking function corresponding to the

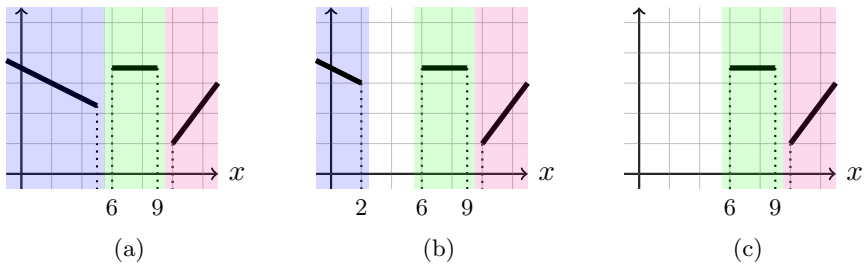


Figure 9.2: Example of dual widening of piecewise-defined function of one variable, shown in (a) and (b), respectively. The result is shown in (c).

$$
\begin{aligned}
&\tau_{\mathrm{r}}^{\varphi\natural}[\![\, {}^{l}\texttt{skip} \,]\!]t \stackrel{\text{def}}{=} \textsc{reset}_{\mathrm{T}}[\![\, \varphi \,]\!](l, R, \textsc{step}_{\mathrm{T}}(t)) \\
&\tau_{\mathrm{r}}^{\varphi\natural}[\![\, {}^{l}X := aexp \,]\!]t \stackrel{\text{def}}{=} \textsc{reset}_{\mathrm{T}}[\![\, \varphi \,]\!](l, R, (\textsc{b-assign}_{\mathrm{T}}[\![\, X := aexp \,]\!]R)(t)) \\
&\tau_{\mathrm{r}}^{\varphi\natural}[\![\, \texttt{if}\ {}^{l}bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi} \,]\!]t \stackrel{\text{def}}{=} \\
&\qquad \textsc{reset}_{\mathrm{T}}[\![\, \varphi \,]\!](l, R, F_1^{\natural}[t] \curlyvee_{\mathrm{T}}[R]\ F_2^{\natural}[f]) \\
&\qquad F_1^{\natural}[t] \stackrel{\text{def}}{=} (\textsc{filter}_{\mathrm{T}}[\![\, bexp \,]\!]R)(\tau_{\mathrm{r}}^{\varphi\natural}[\![\, stmt_1 \,]\!]t) \\
&\qquad F_2^{\natural}[t] \stackrel{\text{def}}{=} (\textsc{filter}_{\mathrm{T}}[\![\, \texttt{not}\ bexp \,]\!]R)(\tau_{\mathrm{r}}^{\varphi\natural}[\![\, stmt_2 \,]\!]t) \\
&\tau_{\mathrm{r}}^{\varphi\natural}[\![\, \texttt{while}\ {}^{l}bexp\ \texttt{do}\ stmt\ \texttt{od} \,]\!]t \stackrel{\text{def}}{=} \mathrm{gfp}^{\natural}_{G(t)}\ \phi_{\mathrm{r}}^{\varphi\natural} \\
&\qquad G \stackrel{\text{def}}{=} \tau_{\mathrm{g}}^{\varphi\natural}[\![\, \texttt{while}\ {}^{l}bexp\ \texttt{do}\ stmt\ \texttt{od} \,]\!] \\
&\qquad \phi_{\mathrm{r}}^{\varphi\natural}(x) \stackrel{\text{def}}{=} \textsc{reset}_{\mathrm{T}}[\![\, \varphi \,]\!](l, R, F^{\natural}[x] \curlyvee_{\mathrm{T}}[R]\ (\textsc{filter}_{\mathrm{T}}[\![\, \texttt{not}\ bexp \,]\!]R)(t)) \\
&\qquad F^{\natural}[x] \stackrel{\text{def}}{=} (\textsc{filter}_{\mathrm{T}}[\![\, bexp \,]\!]R)(\tau_{\mathrm{r}}^{\varphi\natural}[\![\, stmt \,]\!]x) \\
&\tau_{\mathrm{r}}^{\varphi\natural}[\![\, stmt_1\ stmt_2 \,]\!]t \stackrel{\text{def}}{=} \tau_{\mathrm{r}}^{\varphi\natural}[\![\, stmt_1 \,]\!](\tau_{\mathrm{r}}^{\varphi\natural}[\![\, stmt_2 \,]\!]t)
\end{aligned}
$$

Figure 9.3: Abstract recurrence semantics of instructions *stmt*.

final control point of the instruction, and outputs a decision tree defined over a subset of the reachable environments $R \in \mathcal{D}$, which over-approximates the ranking function corresponding to the initial control point of the instruction. Each function $\tau_{\mathrm{r}}^{\varphi\natural}[\![\, stmt \,]\!]$ invokes the redefined reset operator $\textsc{reset}_{\mathrm{T}}$. For a `while` loop, $\mathrm{gfp}^{\natural}\ \phi_{\mathrm{r}}^{\varphi\natural}$ is the limit of the iteration sequence with dual widening:

$$
\begin{aligned}
y_0 &\stackrel{\text{def}}{=} G(t) \\
y_{n+1} &\stackrel{\text{def}}{=} \begin{cases} y_n & y_n \sqsubseteq_{\mathrm{T}} [R]\ \phi_{\mathrm{r}}^{\varphi\natural}(y_n) \wedge y_n \preccurlyeq_{\mathrm{T}} [R]\ \phi_{\mathrm{r}}^{\varphi\natural}(y_n) \\ y_n \bar{\nabla}_{\mathrm{T}}\ \phi_{\mathrm{r}}^{\varphi\natural}(y_n) & \text{otherwise} \end{cases}
\end{aligned}
\tag{9.3.14}
$$

where G is the guarantee semantics of the loop (cf. Figure 9.1).

In absence of run-time errors, the abstract semantics $\tau_{\mathrm{r}}^{\varphi\natural}[\![\, stmt \,]\!]$, given sound over-approximations $R \in \mathcal{D}$ of $\tau_{\mathrm{I}}(i[\![\, stmt \,]\!])$ and $D \in \mathcal{D}$ of $\tau_{\mathrm{I}}(f[\![\, stmt \,]\!])$, is a sound over-approximation of the semantics $\tau_{\mathrm{r}}^{\varphi}[\![\, stmt \,]\!]$ defined in Section 9.3.2:

Lemma 9.3.9 $\tau_r^{\varphi}[\![\, stmt \,]\!]\gamma_T[D]t \preccurlyeq \gamma_T[R]\tau_r^{\varphi\natural}[\![\, stmt \,]\!]t$.

Proof.
See Appendix A.6. ■

The abstract recurrence semantics $\tau_{\mathrm{r}}^{\varphi\natural}[\![\, prog \,]\!] \in \mathcal{T}$ of a program *prog* outputs the decision tree over-approximating the ranking function corresponding to the initial program control point $i[\![\, prog \,]\!] \in \mathcal{L}$. It is defined by taking as input the leaf node $\mathrm{LEAF} : \bot_{\mathrm{F}}$ as:

Definition 9.3.10 (Abstract Recurrence Semantics) *The* abstract recurrence semantics $\tau_r^{\varphi\natural}[\![\, prog \,]\!] \in \mathcal{T}$ *of a program prog is:*

$$\tau_r^{\varphi\natural}[\![\, prog \,]\!] = \tau_r^{\varphi\natural}[\![\, stmt^{\,l} \,]\!] \stackrel{def}{=} \tau_r^{\varphi\natural}[\![\, stmt \,]\!]\mathrm{RESET}_T[\![\, \varphi \,]\!](l, R, \mathrm{LEAF} : \bot_F) \tag{9.3.15}$$

where the abstract recurrence semantics $\tau_r^{\varphi\natural}[\![\, stmt \,]\!] : \mathcal{T} \to \mathcal{T}$ *of each program instruction stmt is defined in Figure 9.3.*

In absence of run-time errors, the abstract recurrence semantics $\tau_{\mathrm{r}}^{\varphi\natural}[\![\, prog \,]\!] \in \mathcal{T}$, given a sound numerical over-approximation $R \in \mathcal{D}$ of the reachable environments $\tau_{\mathrm{I}}(i[\![\, prog \,]\!])$, is sound with respect to $\tau_{\mathrm{r}}^{\varphi}[\![\, prog \,]\!] \in \mathcal{E} \rightharpoonup \mathbb{O}$:

Theorem 9.3.11 $\tau_r^{\varphi}[\![\, prog \,]\!] \preccurlyeq \gamma_T[R]\tau_r^{\varphi\natural}[\![\, prog \,]\!]$

Proof (Sketch).
The proof follows from the soundness of the operators of the decision trees abstract domain (cf. Lemma 9.3.9) used for the definition of $\tau_{\mathrm{r}}^{\varphi\natural}[\![\, prog \,]\!] \in \mathcal{T}$. ■

In particular, the abstract recurrence semantics provides *sufficient preconditions* for ensuring a recurrence property $\Box\Diamond\varphi$ for a given over-approximation $R \in \mathcal{D}$ of the set of initial states $\mathcal{I} \subseteq \Sigma$:

Corollary 9.3.12 *A program satisfies a recurrence property* $\Box\Diamond\varphi$ *for execution traces starting from a set of states* $\gamma_D(R)$ *if* $\gamma_D(R) \subseteq \mathrm{dom}(\gamma_T[R]\tau_r^{\varphi\natural}[\![\, prog \,]\!])$.

9.4 Implementation

We have incorporated the static analysis methods for guarantee and recurrence temporal properties that we have presented in this chapter into our prototype static analyzer FuncTion that we have presented in Chapter 8.

The prototype, when the guarantee or recurrence analysis methods are selected, accepts state properties written as C-like pure expressions.

The following examples illustrate the potential and effectiveness of our new static analysis methods. These and additional examples are available from FuncTion web interface: http://www.di.ens.fr/~urban/FuncTion.html.

Example 9.4.1

Let us consider again the program of Example 9.1.1:

```
while 1(x ≥ 0) do
  2x := x + 1
od
while 3( true ) do
  if 4(x ≤ 10) then
    5x := x + 1
  then
    6x := −x
  fi
od7
```

and the guarantee property $\Diamond(x = 3)$. FuncTion, using interval constraints based on the *intervals abstract domain* (cf. Section 3.4.1) for the decision nodes and *affine functions* for the leaf nodes (cf. Equation 5.2.6), infers the following ranking function associated with program control point **1**:

$$\lambda x. \begin{cases} -3x + 10 & x < 0 \\ -2x + 6 & 0 \leq x \wedge x \leq 3 \\ \text{undefined} & \text{otherwise} \end{cases}$$

which bounds the wait (from the program control point **1**) for the desirable state $x = 3$ by $-3x + 10$ program execution steps when $x < 0$, and by $-2x + 6$ execution steps when $0 \leq x \wedge x \leq 3$. The analysis is inconclusive when $3 < x$. In this case, when $3 < x$, the guarantee property is never satisfied (cf. Example 9.1.3). Thus, the precondition $x \leq 3$ induced by the domain of the ranking function is the weakest precondition for $\Diamond(x = 3)$.

Let us consider now the recurrence property $\Box\Diamond(x = 3)$. FuncTion infers the following ranking function associated with program control point **1**:

$$\lambda x. \begin{cases} -3x + 10 & x < 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

which induces the precondition $x < 0$ for $\Box\Diamond(x = 3)$. Indeed, when $0 \leq x \wedge x \leq 3$, the desirable state $x = 3$ does not occur infinitely often but only once. Again $x < 0$ is the weakest precondition for $\Box\Diamond(x = 3)$.

Instead, for both $\Diamond(x = 3)$ and $\Box\Diamond(x = 3)$, FuncTion infers the following ranking function associated with program control point **3**:

$$\lambda x. \begin{cases} -3x + 9 & x \leq 3 \\ -3x + 72 & 3 < x \leq 10 \\ 3x + 12 & 10 < x \end{cases}$$

which bounds the wait (from the program control point **3**) for the next occurrence of $x = 3$ by $-3x + 9$ execution steps when $x \leq 3$, by $-3x + 72$ execution steps when $3 < x \leq 10$, and by $3x + 12$ execution steps when $10 < x$.

Example 9.4.2

Let us consider the following program:

```
1 c := 1
while 2( true ) do
  3 x := c
  while 4(0 < x) do
    5 x := x − 1
    6 c := c + 1
  od
od 7
```

Each iteration of the outer loop, assigns to the program variable x the value of some counter c, which initially has value one; then, the inner loop decreases the value of x and increases the value of the counter c until the value of x becomes less than or equal to zero.

FuncTion, using interval constraints based on the *intervals abstract domain* (cf. Section 3.4.1) for the decision nodes and *affine functions* for the leaf nodes (cf. Equation 5.2.6), is able to prove that the recurrence property $\Box\Diamond(x = 0)$ is always satisfied by the program. The piecewise-defined ranking function inferred at program control point **1** bounds the wait for the *next*

occurrence of the desirable state $x = 0$ by five program execution steps (i.e., executing the variable assignment $c := 1$, testing the outer loop condition, executing the assignment $x := c$, testing the inner loop condition and executing the assignment $x := x - 1$). The analysis infers a more interesting raking function associated to program control point 4:

$$\lambda x.\ \lambda c.\ \begin{cases} 3c + 2 & x < 0 \wedge 0 < c \\ 3 & x < 0 \wedge c = 0 \\ 1 & x = 0 \wedge 0 \leq c \\ 3x - 1 & (x = 1 \wedge -1 \leq c) \vee (2 \leq x \wedge -2 \leq c) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The function bounds the wait for the next occurrence of $x = 0$ by $3c + 2$ execution steps when $x < 0 \wedge 0 < c$, by 3 execution steps when $x < 0 \wedge c = 0$ (i.e., testing the inner loop condition, testing the outer loop condition and executing the assignment $x := c$), by 1 execution step when $x = 0 \wedge 0 \leq c$ (i.e., testing the inner loop condition) and by $3x - 1$ execution steps when $(x = 1 \wedge -1 \leq c) \vee (2 \leq x \wedge -2 \leq c)$. In the last case there is a precision loss due to a lack of expressiveness of the intervals abstract domain: if x is strictly positive at program control point 4, the weakest precondition ensuring infinitely many occurrences of the desirable state $x = 0$ is $-x \leq c$, which is not representable by the intervals abstract domain.

Example 9.4.3

Let us consider the following program:

```
while [1]( true ) do
  [2]x := ?
  while [3](x ≠ 0) do
    if [4](0 < x) then
      [5]x := x − 1
    else
      [6]x := x + 1
    fi
  od
od[7]
```

Each iteration of the outer loop, resets the value of the program variable x with the non-deterministic assignment $x :=$?; then, the inner loop decreases or increases the value of x until it becomes equal to zero.

The recurrence property $\Box\Diamond(x = 0)$ is clearly satisfied by the program. However, because of the non-deterministic assignment $x :=$?, the number of execution steps between two occurrences of the desirable state $x = 0$ is unbounded. FuncTion is able to prove that the property is satisfied using interval constraints based on the *intervals abstract domain* (cf. Section 3.4.1) for the decision nodes and *ordinal-valued functions* for the leaf nodes (cf. Chapter 6). The inferred ranking function at program control point **1**:

$$\lambda x.\ \omega + 8$$

means that, whatever the value of the variable x, the number of execution steps between two occurrences of $x = 0$ is unbounded but *finite*.

Example 9.4.4

Let us consider Peterson's algorithm [Pet81] for mutual exclusion:

$$\begin{array}{l}
{}^{1}flag_1 := 0 \\
{}^{2}flag_2 := 0 \\
\left[\begin{array}{l}
\texttt{while}\ {}^{3}(\ \texttt{true}\ )\ \texttt{do} \\
\quad {}^{4}flag_1 := 1 \\
\quad {}^{5}turn := 2 \\
\quad \texttt{while}\ {}^{6}\begin{pmatrix} flag_2 \neq 0 \\ \wedge \\ turn \neq 1 \end{pmatrix}\ \texttt{do} \\
\qquad {}^{7}\texttt{skip} \\
\quad \texttt{od} \\
\quad {}^{8}\ \text{CRITICAL_SECTION} \\
\quad {}^{9}flag_1 := 0 \\
\texttt{od}^{10}
\end{array}\right] \parallel \left[\begin{array}{l}
\texttt{while}\ {}^{3}(\ \texttt{true}\ )\ \texttt{do} \\
\quad {}^{4}flag_2 := 1 \\
\quad {}^{5}turn := 1 \\
\quad \texttt{while}\ {}^{6}\begin{pmatrix} flag_1 \neq 0 \\ \wedge \\ turn \neq 2 \end{pmatrix}\ \texttt{do} \\
\qquad {}^{7}\texttt{skip} \\
\quad \texttt{od} \\
\quad {}^{8}\ \text{CRITICAL_SECTION} \\
\quad {}^{9}flag_2 := 0 \\
\texttt{od}^{10}
\end{array}\right]
\end{array}$$

Note that *weak fairness* assumptions are required to guarantee bounded bypass (i.e., a process cannot be bypassed by any other process in entering the critical section for more than a finite number of times. At the moment our prototype FuncTion is not able to directly analyze concurrent programs. Thus, we have modeled the algorithm as a fair non-deterministic sequential program which interleaves execution steps from both processes while enforcing 1-bounded bypass (i.e., a process cannot be bypassed by any other process in entering the critical section for more than once). FuncTion, using interval constraints based on the *intervals abstract domain* (cf. Section 3.4.1) for the decision nodes and *affine functions* for the leaf nodes (cf. Equation 5.2.6), is able to prove the recurrence property $\Box\Diamond(\mathbf{8} : \texttt{true})$, meaning that both processes are allowed to enter their critical section infinitely often.

9.5 Related Work

In the recent past, a large body of work has been devoted to proving liveness properties of (concurrent) programs.

A successful approach for proving liveness properties is based on a transformation from model checking of liveness properties to model checking of *safety* properties [BAS02]. The approach looks for and exploits lasso-shaped counterexamples. A similar search for lasso-shaped counterexamples has been used to generalize the model checking algorithm IC3 to deal with liveness properties [BSHZ11]. However, in general, counterexamples to liveness properties in infinite-state systems are not necessarily lasso-shaped. Our approach is not counterexample-based and is meant for proving liveness properties directly, without reduction to safety properties.

In [PR05], Andreas Podelski and Andrey Rybalchenko present a method for the verification of liveness properties based on transition invariants [PR04b]. The approach, as in [Var91], reduces the proof of a liveness property to the proof of *fair termination* by means of a program transformation. It is at the basis of the industrial-scale tool Terminator [CGP+07]. By contrast, our method is meant for proving liveness properties directly, without reduction to termination. Moreover, it avoids the cost of explicit checking for the well-foundedness of the transition invariants.

A distinguishing aspect of our work is the use of infinite height abstract domains, equipped with (dual) widening. We are aware of only one other such work: in [Mas03], Damien Massé proposes a method for proving arbitrary temporal properties based on abstract domains for lower closure operators. A small analyzer is presented in [Mas04] but the approach remains mainly theoretical. We believe that our framework, albeit less general, is more straightforward and of practical use.

An emerging trend focuses on proving *existential* temporal properties (e.g., proving that there exists a particular execution trace). The most recent approaches [BPR13, CK13] are based on counterexample-guided abstraction refinement [CGJ+03]. Our work is designed for proving universal temporal properties (i.e., valid for all program execution traces). We leave proving existential temporal properties as part of our future work.

Finally, to our knowledge, the inference of sufficient preconditions for guarantee and recurrence properties, and the ability to provide upper bounds on the time before a program reaches a desirable state, is unique to our work.

V

Conclusion

10

Future Directions

With this thesis, we have proposed new abstract interpretation-based methods for proving **termination** and other **liveness properties** of programs.

In particular, our first contribution is the design of new **abstract domains** suitable for the abstract interpretation framework for termination proposed by Patrick Cousot and Radhia Cousot [CC12]. These abstract domains automatically infer **sufficient preconditions** for program termination, and synthesize **piecewise-defined ranking functions** through backward analysis (cf. Chapter 5 and Chapter 6). The ranking functions provide upper bounds on the program execution time in terms of execution steps. The abstract domains are parametric in the choice between the expressivity and the cost of the underlying numerical abstract domain (cf. Section 3.4). They are shown to be effective for proving termination of **recursive programs** in Chapter 7.

Our second contribution is an **abstract interpretation framework for liveness properties**, which comes as a generalization of the framework proposed for termination [CC12]. In particular, we have proposed new abstract interpretation-based methods for proving **guarantee** and **recurrence** properties (cf. Chapter 9). We have also reused the abstract domains based on piecewise-defined ranking functions (cf. Chapter 5 and Chapter 6) to effectively infer sufficient preconditions for these properties, and to provide upper bounds on the time before a program reaches a desirable state.

Our last contribution is a prototype **implementation** of a static analyzer based on piecewise-defined ranking functions (cf. Chapter 8). The earlier versions of the prototype participated in the 3*rd International Competition on Software Verification (SV-COMP 2014)*, which featured a category for termination for the first time, and in the 4*th International Competition on Software Verification (SV-COMP 2015)*.

We conclude with some perspectives for future research.

Potential Termination and Non-Termination. In Chapter 4 we have introduced the *definite* and **potential termination semantics** for a program [CC12]. Then, throughout the rest of the thesis we have focused our attention only on the definite termination semantics, and proposed decidable abstractions in Chapter 5 and Chapter 6. As part of our future work, we plan to do a similar work for the potential termination semantics.

We plan to use the same domains based on piecewise-defined ranking functions (cf. Chapter 5 and Chapter 6) but adapt their operators in order to automatically infer **necessary preconditions** for program termination, which are not very interesting per se but provide complementary sufficient preconditions for **non-termination**. In this way, we could complement our analysis method for proving program termination with an analysis method for proving program non-termination. In particular, we envision to run the analyses simultaneously in order to improve precision.

Abstract Domains. In Chapter 5 we have proposed the *decision trees* abstract domain, which we have designed to be parameterized by a *convex* numerical abstract domain, such as intervals [CC76], convex polyhedra [CH78], and octagons [Min06]. In the future, we would like to also allow **non-convex abstract domain**, such as congruences [Gra89].

Moreover, the decision tree abstract domain has been instantiated with *affine* functions, in Chapter 5, and *ordinal-valued* functions, in Chapter 6. It remains for future work to support **non-linear functions**, such as polynomials [BMS05b] or exponentials [Fer05]. Non-linear functions would provide more precise upper bounds on the program execution time for programs with non-linear computational complexity. On this account, we plan to explore further the possible potential of our approach in the termination-related field of automatic **cost analysis** [DLH90, SLH14, Weg75].

In Chapter 5, we also mentioned various ideas for improving the **widening** of decision trees, by introducing thresholds [CC92c] and integrating state-of-the-art precise widening operators [BHRZ05]. We plan to investigate these and further possibilities.

The decision trees abstract domain could also offer an alternative disjunctive refinement of numerical abstract domains [CCM10, GR98, GC10a, SISG06, etc.]. We would also like to explore its potential to be adapted to other program semantics and for proving other program properties.

Fairness and Liveness Properties. In Chapter 9, we have proposed an abstract interpretation framework for *guarantee* and *recurrence* temporal properties [MP90]. The verification of liveness properties is a difficult problem, with not many satisfying solutions. We have made a first step that shows how Abstract Interpretation can also be used for proving liveness properties and that, we believe, opens many possibilities.

We mentioned in Chapter 1 that some of the theoretical work presented in this thesis is being used as part of ongoing research work aimed at the verification of real-time properties of avionics software.

It remains for future work the integration of **fairness** properties [Fra86]. We also plan to extend the present framework to the full hierarchy of temporal properties [MP90], and more generally to arbitrary (universal and existential [BPR13, CK13]) **liveness properties**.

Machine Integers and Floats In this thesis we have only considered programs that operate over *mathematical integers*. The reality is that most programs operate over variables that range over fixed-width numbers, such as **32-bit integers** or **64-bit floating-point numbers**, with the possibility of overflow or underflow. In particular, we cannot ignore the fixed-width semantics, as overflow and underflow can for example cause non-termination in programs that would otherwise terminate. We plan, as part of our future work, to make the decision trees abstract domain aware of the low-level memory representation of data-types.

Heap-Manipulating Programs. We also plan to prove termination and liveness properties of more complex programs, such as **heap-manipulating programs**. We would like to investigate the adaptability of existing methods [BCDO06] and existing abstract domains for shape analysis [CR13], and possibly design new techniques.

Concurrent Programs. Finally, in this thesis we have developed methods for proving termination and liveness properties of *sequential* programs. A natural future direction is considering **concurrent programs**. When analyzing concurrent programs, it is necessary to consider all possible interactions between concurrently executing threads. The usual method for proving concurrent programs correct is based on rely-guarantee or assume-guarantee style of reasoning, which considers every thread in isolation under assumptions on its environment and thus avoids reasoning about thread interactions directly [GCPV09, Min14]. We plan to extend these strategies to liveness properties.

Bibliography

[ADFG10] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-Dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *SAS*, pages 117–133, 2010.

[BAL09] Amir M. Ben-Amram and Chin Soon Lee. Ranking Functions for Size-Change Termination II. *Logical Methods in Computer Science*, 5(2), 2009.

[BAS02] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness Checking as Safety Checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[BCC+07] Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Variance Analyses from Invariance Analyses. In *POPL*, pages 211–224, 2007.

[BCC+10] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Static Analysis and Verification of Aerospace Software by Abstract Interpretation. In *AIAA*, 2010.

[BCDO06] Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In *CAV*, pages 386–400, 2006.

[BCF13] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better Termination Proving through Cooperation. In *CAV*, pages 413–429, 2013.

[BHRZ05] Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. Precise Widening Operators for Convex Polyhedra. *Science of Computer Programming*, 58(1-2):28–56, 2005.

[BMS05a] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear Ranking with Reachability. In *CAV*, pages 491–504, 2005.

[BMS05b] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The Polyranking Principle. In *ICALP*, pages 1349–1361, 2005.

[Bou93] Franois Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In *FMPA*, pages 128–141, 1993.

[BPR13] Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. Solving Existentially Quantified Horn Clauses. In *CAV*, pages 869–882, 2013.

[Bra11] Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In *VMCAI*, pages 70–87, 2011.

[Bry86] Randal E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[BSHZ11] Aaron R. Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. An Incremental Approach to Model Checking Progress Properties. In *FMCAD*, pages 144–153, 2011.

[Can95] Georg Cantor. Beiträge zur Begründung der Transfiniten Mengenlehre. *Mathematische Annalen*, 46:481–512, 1895.

[Can97] Georg Cantor. Beiträge zur Begründung der Transfiniten Mengenlehre. *Mathematische Annalen*, 49:207–246, 1897.

[CC76] Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Programs. In *Symposium on Programming*, pages 106–130, 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Constructive Versions of Tarski's Fixed Point Theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.

[CC92a] Patrick Cousot and Radhia Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.

[CC92b] Patrick Cousot and Radhia Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[CC92c] Patrick Cousot and Radhia Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *PLILP*, pages 269–295, 1992.

[CC10] Patrick Cousot and Radhia Cousot. A Gentle Introduction to Formal Verification of Computer Systems by Abstract Interpretation. In *Logics and Languages for Reliability and Security*, volume 25 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 1–29. IOS Press, 2010.

[CC12] Patrick Cousot and Radhia Cousot. An Abstract Interpretation Framework for Termination. In *POPL*, pages 245–258, 2012.

[CCM10] Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. A Scalable Segmented Decision Tree Abstract Domain. In *Essays in Memory of Amir Pnueli*, pages 72–95, 2010.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic. In *Workshop on Logics of Programs*, pages 52–71, 1981.

[CGJ+03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGLA+08] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. Proving Conditional Termination. In *CAV*, pages 328–340, 2008.

[CGP+07] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that Programs Eventually do Something Good. In *POPL*, pages 265–276, 2007.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–96, 1978.

[CK13] Byron Cook and Eric Koskinen. Reasoning About Nondeterminism in Programs. In *PLDI*, pages 219–230, 2013.

[Cou78] Patrick Cousot. *Méthodes Itératives de Construction et d'Approximation de Points Fixes d'Opérateurs Monotones sur un Treillis, Analyse Sémantique de Programmes.* Thèse d'État Ès Sciences Mathématiques, Université Joseph Fourier, Grenoble, France, 1978.

[Cou85] Radhia Cousot. *Fondements des Méthodes de Preuve d'Invariance et de Fatalité de Programmes Parallèles.* Thèse d'État Ès Sciences Mathématiques, Institut National Polytechnique de Lorraine, Nancy, France, 1985.

[Cou97] Patrick Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Electronic Notes in Theoretical Computer Science*, 6:77–102, 1997.

[Cou02] Patrick Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 277(1-2):47–103, 2002.

[Cou15] Patrick Cousot. Abstracting Induction by Extrapolation and Interpolation. In *VMCAI*, pages 19–42, 2015.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond Safety. In *CAV*, pages 415–418, 2006.

[CPR11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving Program Termination. *Communications of the ACM*, 54(5):88–98, 2011.

[CR13] Bor-Yuh Evan Chang and Xavier Rival. Modular construction of shape-numeric analyzers. In *Festschrift for Dave Schmidt*, pages 161–185, 2013.

[CS01] Michael Colón and Henny Sipma. Synthesis of Linear Ranking Functions. In *TACAS*, pages 67–81, 2001.

[CS02] Michael Colón and Henny Sipma. Practical Methods for Proving Program Termination. In *CAV*, pages 442–454, 2002.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[CSZ13] Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. Lexicographic Termination Proving. In *TACAS*, pages 47–61, 2013.

[DLH90] Saumya K. Debray, Nai-Wei Lin, and Manuel V. Hermenegildo. Task Granularity Analysis in Logic Programs. In *PLDI*, pages 174–188, 1990.

[DU15] Vijay D'Silva and Caterina Urban. Conflict-Driven Abstract Interpretation for Conditional Termination. In *CAV*, 2015.

[Fer05] Jérôme Feret. The Arithmetic-Geometric Progression Abstract Domain. In *VMCAI*, pages 42–58, 2005.

[FKN80] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On Visible Surface Generation by a Priori Tree Structures. *SIGGRAPH Computer Graphics*, 14(3):124–133, 1980.

[Flo67] Robert W. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[Fra86] Nissim Francez. *Fairness*. Springer, 1986.

[GC10a] Arie Gurfinkel and Sagar Chaki. BOXES: A Symbolic Abstract Domain of Boxes. In *SAS*, pages 287–303, 2010.

[GC10b] Arie Gurfinkel and Sagar Chaki. Combining Predicate and Numeric Abstraction for Software Model Checking. *STTT*, 12(6):409–427, 2010.

[GCPV09] Alexey Gotsman, Byron Cook, Matthew J. Parkinson, and Viktor Vafeiadis. Proving that Non-Blocking Algorithms Don't Block. In *POPL*, pages 16–28, 2009.

[GG13] Pierre Ganty and Samir Genaim. Proving Termination Starting from the End. In *CAV*, pages 397–412, 2013.

[GMR15] Laure Gonnord, David Monniaux, and Gabriel Radanne. Synthesis of Ranking Functions using Extremal Counterexamples. In *PLDI*, 2015.

[GR98] Roberto Giacobazzi and Francesco Ranzato. Optimal Domains for Disjunctive Abstract Intepretation. *Sci. Comput. Program.*, 32(1-3):177–210, 1998.

[Gra89] Philippe Granger. Static Analysis of Arithmetic Congruences. *International Journal of Computer Math*, pages 165–199, 1989.

[GSKT06] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. Automatic Termination Proofs in the Dependency Pair Framework. In *IJCAR*, pages 281–286, 2006.

[HDL+15] Matthias Heizmann, Daniel Dietsch, Jan Leike, Betim Musa, and Andreas Podelski. Ultimate Automizer with Array Interpolation (Competition Contribution). In *TACAS*, 2015.

[HHLP13] Matthias Heizmann, Jochen Hoenicke, Jan Leike, and Andreas Podelski. Linear Ranking for Linear Lasso Programs. In *ATVA*, pages 365–380, 2013.

[HJP10] Matthias Heizmann, Neil D. Jones, and Andreas Podelski. Size-Change Termination and Transition Invariants. In *SAS*, pages 22–50, 2010.

[Hoa69] Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Jea02] Bertrand Jeannet. Representing and Approximating Transfer Functions in Abstract Interpretation of Hetereogeneous Datatypes. In *SAS*, pages 52–68, 2002.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*, pages 661–667, 2009.

[KRP93] Owen Kaser, C. R. Ramakrishnan, and Shaunak Pawagi. On the Conversion of Indirect to Direct Recursion. *LOPLAS*, 2(1-4):151–164, 1993.

[Kun80] Kenneth Kunen. *Set Theory: An Introduction to Independence Proofs*. Studies in Logic and the Foundations of Mathematics. Elsevier, 1980.

[Lam77] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[Lee09] Chin Soon Lee. Ranking Functions for Size-Change Termination. *ACM Transactions on Programming Languages and Systems*, 31(3), 2009.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The Size-Change Principle for Program Termination. In *POPL*, pages 81–92, 2001.

[LLPY97] Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Efficient Verification of Real-Time Systems: Compact Data Structure and State-Space Reduction. In *RTSS*, pages 14–24, 1997.

[LORCR13] Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving Termination of Imperative Programs using Max-SMT. In *FMCAD*, pages 218–225, 2013.

[LPWY99] Kim Guldstrand Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Clock Difference Diagrams. *Nordic Journal of Computing*, 6(3):271–298, 1999.

[LQC15] Ton-Chanh Le, Shengchao Qin, and Wei-Ngan Chin. Termination and Non-Termination Specification Inference. In *PLDI*, 2015.

[Mas03] Damien Massé. Property Checking Driven Abstract Interpretation-Based Static Analysis. In *VMCAI*, pages 56–69, 2003.

[Mas04] Damien Massé. Abstract Domains for Property Checking Driven Analysis of Temporal Properties. In *AMAST*, pages 349–363, 2004.

[Mas14] Damien Massé. Policy Iteration-based Conditional Termination and Ranking Functions. In *VMCAI*, pages 453–471, 2014.

[MH92] Kalyan Muthukumar and Manuel V. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, 1992.

[Min04] Antoine Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, December 2004.

[Min06] Antoine Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[Min14] Antoine Miné. Relational Thread-Modular Static Value Analysis by Abstract Interpretation. In *VMCAI*, pages 39–58, 2014.

[MJ84] Lockwood Morris and C. B. Jones. An Early Program Proof by Alan Turing. *IEEE Annals of the History of Computing*, 6(2):139–143, 1984.

[MLAH99] Jesper B. Møller, Jakob Lichtenberg, Henrik Reif Andersen, and Henrik Hulgaard. Difference decision diagrams. In *CSL*, pages 111–125, 1999.

[Moo66] Ramon E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[MP90] Zohar Manna and Amir Pnueli. A Hierarchy of Temporal Properties. In *PODC*, pages 377–410, 1990.

[MP96] Zohar Manna and Amir Pnueli. The Temporal Verification of Reactive Systems: Progress, 1996.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *CADE*, pages 748–752, 1992.

[Par69] David Park. Fixpoint Induction and Proofs of Program Properties. *Machine Intelligence*, 5:59–78, 1969.

[Pet81] Gary L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, 1981.

[PR04a] Andreas Podelski and Andrey Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI*, pages 239–251, 2004.

[PR04b] Andreas Podelski and Andrey Rybalchenko. Transition Invariants. In *LICS*, pages 32–41, 2004.

[PR05] Andreas Podelski and Andrey Rybalchenko. Transition Predicate Abstraction and Fair Termination. In *POPL*, pages 132–144, 2005.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *5th International Symposium on Programming*, pages 337–351, 1982.

[SAF+15] Thomas Ströder, Cornelius Aschermann, Florian Frohn, Jera Hensel, and Jrgen Giesl. AProVE: Termination and Memory Safety of C Programs (Competition Contribution). In *TACAS*, 2015.

[SISG06] Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. Static Analysis in Disjunctive Numerical Domains. In *SAS*, pages 3–17, 2006.

[SLH14] Alejandro Serrano, Pedro López-García, and Manuel V. Hermenegildo. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *Theory and Practice of Logic Programming*, 14(4-5):739–754, 2014.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[TSWK11] Aliaksei Tsitovich, Natasha Sharygina, Christoph M. Wintersteiger, and Daniel Kroening. Loop Summarization and Termination Analysis. In *TACAS*, pages 81–95, 2011.

[Tur36] Alan Turing. On Computable Numbers, with An Application to the Entscheidungs Problem. *London Mathematical Society*, 42(2):230–265, 1936.

[Tur49] Alan Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.

[UM14a] Caterina Urban and Antoine Miné. An Abstract Domain to Infer Ordinal-Valued Ranking Functions. In *ESOP*, pages 412–431, 2014.

[UM14b] Caterina Urban and Antoine Miné. A Decision Tree Abstract Domain for Proving Conditional Termination. In *SAS*, pages 302–318, 2014.

[UM14c] Caterina Urban and Antoine Miné. To Infinity... and Beyond. In *WST*, pages 85–89, 2014.

[UM15] Caterina Urban and Antoine Miné. Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation. In *VMCAI*, 2015.

[Urb13a] Caterina Urban. The Abstract Domain of Segmented Ranking Functions. In *SAS*, pages 43–62, 2013.

[Urb13b] Caterina Urban. Piecewise-Defined Ranking Functions. In *WST*, pages 69–73, 2013.

[Urb15] Caterina Urban. FuncTion: An Abstract Domain Functor for Termination (Competition Contribution). In *TACAS*, 2015.

[Var91] Moshe Y. Vardi. Verification of Concurrent Programs: The Automata-Theoretic Framework. *Annals of Pure and Applied Logic*, 51(1-2):79–98, 1991.

[vN23] John von Neumann. Zur Einführung der Transfiniten Zahlen. *Acta Scientiarum Mathematicarum (Szeged)*, 1(4):199–208, 1923.

[Weg75] Ben Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, 1975.

A

Proofs

A.1 Missing Proofs from Chapter 1

Theorem A.1.1 (Halting Problem) *The* halting problem *is undecidable.*

Proof (by Geoffrey K. Pullum).

Scooping the Loop Snooper

No general procedure for bug checks will do.
Now, I won't just assert that, I'll prove it to you.
I will prove that although you might work till you drop,
you cannot tell if computation will stop.

For imagine we have a procedure called P
that for specified input permits you to see
whether specified source code, with all of its faults,
defines a routine that eventually halts.

You feed in your program, with suitable data,
and P gets to work, and a little while later
(in finite compute time) correctly infers
whether infinite looping behavior occurs.

If there will be no looping, then P prints out 'Good.'
That means work on this input will halt, as it should.
But if it detects an unstoppable loop,
then P reports 'Bad!' — which means you're in the soup.

Well, the truth is that P cannot possibly be,
because if you wrote it and gave it to me,
I could use it to set up a logical bind
that would shatter your reason and scramble your mind.

Here's the trick that I'll use — and it's simple to do.
I'll define a procedure, which I will call Q,
that will use P's predictions of halting success
to stir up a terrible logical mess.

For a specified program, say A, one supplies,
the first step of this program called Q I devise
is to find out from P what's the right thing to say
of the looping behavior of A run on A.

If P's answer is 'Bad!', Q will suddenly stop.
But otherwise, Q will go back to the top,
and start off again, looping endlessly back,
till the universe dies and turns frozen and black.

And this program called Q wouldn't stay on the shelf;
I would ask it to forecast its run on itself.
When it reads its own source code, just what will it do?
What's the looping behavior of Q run on Q?

If P warns of infinite loops, Q will quit;
yet P is supposed to speak truly of it!
And if Q's going to quit, then P should say 'Good.'
Which makes Q start to loop! (P denied that it would.)

No matter how P might perform, Q will scoop it:
Q uses P's output to make P look stupid.
Whatever P says, it cannot predict Q:
P is right when it's wrong, and is false when it's true!

I've created a paradox, neat as can be —
and simply by using your putative P.
When you posited P you stepped into a snare;
Your assumption has led you right into my lair.

So where can this argument possibly go?

I don't have to tell you; I'm sure you must know.
A reductio: There cannot possibly be
a procedure that acts like the mythical P.

You can never find general mechanical means
for predicting the acts of computing machines;
it's something that cannot be done. So we users
must find our own bugs. Our computers are losers!

A.2 Missing Proofs from Chapter 4

Theorem 4.2.18 *A program may terminate for execution traces starting from a given set of initial states $\mathcal{I}$ if and only if $\mathcal{I} \subseteq \mathrm{dom}(\tau_{mt})$.*

Proof of Theorem 4.2.18.
The proof follows by Park's Fixpoint Induction Principle (cf. Theorem 2.2.14) along the lines of the proof of Theorem 4.2.23 proposed in [CC12].

We have $\mathcal{I} \subseteq \mathrm{dom}(\tau_{\mathrm{mt}})$ if and only if $\exists v\colon \Sigma \rightharpoonup \mathbb{O} : \tau_{\mathrm{mt}} \sqsubseteq v \wedge \mathcal{I} \subseteq \mathrm{dom}(v)$. From Theorem 4.2.15, we have $\exists v\colon \Sigma \rightharpoonup \mathbb{O} : \tau_{\mathrm{mt}} \sqsubseteq v$ if and only if $\exists v\colon \Sigma \rightharpoonup \mathbb{O} : \mathrm{lfp}_{\dot{\emptyset}}^{\sqsubseteq}\ \phi_{\mathrm{mt}} \sqsubseteq v$. From Park's Fixpoint Induction Principle, we have $\exists v\colon \Sigma \rightharpoonup \mathbb{O} : \mathrm{lfp}_{\dot{\emptyset}}^{\sqsubseteq}\ \phi_{\mathrm{mt}} \sqsubseteq v$ if and only if $\exists v\colon \Sigma \rightharpoonup \mathbb{O} : \exists v'\colon \Sigma \rightharpoonup \mathbb{O} : \dot{\emptyset} \sqsubseteq v' \wedge \phi_{\mathrm{mt}}(v') \sqsubseteq v' \wedge v' \sqsubseteq v$ and, by definition of $\sqsubseteq$ (cf. Equation 4.2.8) and choosing $v' = v$, $\exists v\colon \Sigma \rightharpoonup \mathbb{O} : \phi_{\mathrm{mt}}(v) \sqsubseteq v$. Then, by definition of $\sqsubseteq$, we have $\exists v\colon \Sigma \rightharpoonup \mathbb{O} : \phi_{\mathrm{mt}}(v) \sqsubseteq v$ if and only if $\exists v\colon \Sigma \rightharpoonup \mathbb{O} : \mathrm{dom}(\phi_{\mathrm{mt}}(v)) \subseteq \mathrm{dom}(v) \wedge \forall s \in \mathrm{dom}(\phi_{\mathrm{mt}}(v)) : \phi_{\mathrm{mt}}(v)s \leq v(s)$. Now, by definition of ϕ_{mt} (cf. Equation 4.2.11), we have $\exists v\colon \Sigma \rightharpoonup \mathbb{O} : \mathrm{dom}(\phi_{\mathrm{mt}}(v)) \subseteq \mathrm{dom}(v) \wedge \forall s \in \mathrm{dom}(\phi_{\mathrm{mt}}(v)) : \phi_{\mathrm{mt}}(v)s \leq v(s)$ if and only if $\exists v\colon \Sigma \rightharpoonup \mathbb{O} : \forall s \in \mathrm{dom}(v) : \inf\{v(s') + 1 \mid \exists s' \in \mathrm{dom}(v) : \langle s, s'\rangle \in \tau\} \leq v(s)$, that is, if and only if $\exists v\colon \Sigma \rightharpoonup \mathbb{O} : \forall s \in \mathrm{dom}(v) : (\exists s' \in \mathrm{dom}(v) : \langle s, s'\rangle \in \tau) \Rightarrow \exists s' \in \mathrm{dom}(v) : \langle s, s'\rangle \in \tau \wedge v(s') < v(s)$. From Definition 4.1.3, $v\colon \Sigma \rightharpoonup \mathbb{O}$ is a potential ranking function. Thus, choosing $\mathcal{I} \subseteq \mathrm{dom}(v)$, concludes the proof. ■

A.3 Missing Proofs from Chapter 5

Lemma 5.2.2 $\forall f_1, f_2 \in \mathcal{F} : f_1 \preccurlyeq_F [D]\ f_2 \Rightarrow \gamma_F[D]f_1 \preccurlyeq \gamma_F[D]f_2$.

Proof of Lemma 5.2.2.

Let $D \in \mathcal{D}$. We reason by cases. First, we consider the case of defined and undefined leaf nodes. Then, we consider the case of defined leaf nodes.

Let $f_1 \in \mathcal{F} \setminus \{\bot_\mathrm{F}, \top_\mathrm{F}\}$ and let $f_2 \in \{\bot_\mathrm{F}, \top_\mathrm{F}\}$. We have $f_1 \preccurlyeq_\mathrm{F} [D] f_2$ from Figure 5.3a. Moreover, from Equation 5.2.9, we have $\gamma_\mathrm{F}[D]f_2 = \dot{\emptyset}$. Thus, since $\mathrm{dom}(\gamma_\mathrm{F}[D]f_2) = \emptyset$, we have $\mathrm{dom}(\gamma_\mathrm{F}[D]f_1) \supseteq \mathrm{dom}(\gamma_\mathrm{F}[D]f_2)$ and, from Equation 4.2.12, we have $\gamma_\mathrm{F}[D]f_1 \preccurlyeq \gamma_\mathrm{F}[D]f_2$.

Let $f_1, f_2 \in \mathcal{F} \setminus \{\bot_\mathrm{F}, \top_\mathrm{F}\}$ such that $f_1 \preccurlyeq_\mathrm{F} [D] f_2$. From Equation 5.2.9, we have $\mathrm{dom}(\gamma_\mathrm{F}[D]f_1) = \mathrm{dom}(\gamma_\mathrm{F}[D]f_2)$ and, in particular, $\mathrm{dom}(\gamma_\mathrm{F}[D]f_1) \supseteq \mathrm{dom}(\gamma_\mathrm{F}[D]f_2)$. Moreover, from Equation 5.2.7, for all $\rho \in \gamma_\mathrm{D}(D)$ we have $f_1(\rho(x_1), \ldots, \rho(x_k)) \leq f_2(\rho(x_1), \ldots, \rho(x_k))$. Thus, from Equation 4.2.12, we have $\gamma_\mathrm{F}[D]f_1 \preccurlyeq \gamma_\mathrm{F}[D]f_2$.

This concludes the proof that $\gamma_\mathrm{F}[D]$ is monotonic. ■

Lemma 5.2.4 $\forall t_1, t_2 \in \mathcal{T} : t_1 \preccurlyeq_T [D]\ t_2 \Rightarrow \gamma_T[D]f_1 \preccurlyeq \gamma_T[D]f_2$.

Proof of Lemma 5.2.4.

Let $D \in \mathcal{D}$ and let $t_1, t_2 \in \mathcal{T}$ such that $t_1 \preccurlyeq_\mathrm{T} [D]\ t_2$ The approximation ordering $\preccurlyeq_\mathrm{T}$ between decision trees is implemented by Algorithm 3: the functions A-ORDER calls (the function ORDER of Algorithm 2, which in turn calls) Algorithm 1 for tree unification, and then compares the decision trees "leaf-wise". Algorithm 1 forces the decision trees to have the same structure. Indeed, the missing linear constraints (cf. Line 6 and Line 15) are added to the decision trees (cf. Line 14 and Line 23). Thus, after the tree unification, all paths to the leaf nodes coincide between the decision trees. Let $C \in \mathcal{C}$ denote the set of linear constraints satisfied along a path of the unified decision trees, and let $f_1, f_2 \in \mathcal{F}$ denote the leaf nodes reached following the path C within the first and the second decision tree, respectively. We have $f_1 \preccurlyeq_\mathrm{F} [\alpha_\mathrm{D}(C)]\ f_2$, since by hypothesis $t_1 \preccurlyeq_\mathrm{T} [D]\ t_2$. The proof follows from Lemma 5.2.2. ■

Lemma 5.2.8 $\tau_{Mt}[\![{}^l\texttt{skip}]\!]\gamma_T[D]t \ \preccurlyeq\ \gamma_T[R]\mathrm{STEP}_T(t)$.

Proof of Lemma 5.2.8.

We have $\tau_\mathrm{I}[\![{}^l\texttt{skip}]\!]\gamma_\mathrm{D}(R) \subseteq \gamma_\mathrm{D}(D)$, since $R \in \mathcal{D}$ and $D \in \mathcal{D}$ are sound over-approximations of $\tau_\mathrm{I}(l)$ and $\tau_\mathrm{I}(f[\![{}^l\texttt{skip}]\!])$, respectively.

Let $C \stackrel{\mathrm{def}}{=} \tau_\mathrm{Mt}[\![{}^l\texttt{skip}]\!]\gamma_\mathrm{T}[D]t$ and let $A \stackrel{\mathrm{def}}{=} \gamma_\mathrm{T}[R]\mathrm{STEP}_\mathrm{T}(t)$. We prove that $\mathrm{dom}(C) \supseteq \mathrm{dom}(A)$ and $\forall \rho \in \mathrm{dom}(A) : C(\rho) \leq A(\rho)$ (cf. Equation 4.2.12).

Let us assume, by absurd, that $\mathrm{dom}(C) \subset \mathrm{dom}(A)$. Then, there exists an environment $\rho \in \mathcal{E}$ such that $\rho \in \mathrm{dom}(A)$ and $\rho \notin \mathrm{dom}(C)$. In particular, $\rho \in$

$\gamma_{\mathrm{D}}(R)$. Thus, since $\tau_{\mathrm{I}}[\![\,^l\texttt{skip}\,]\!]\gamma_{\mathrm{D}}(R) \subseteq \gamma_{\mathrm{D}}(D)$ and by definition of $\tau_{\mathrm{I}}[\![\,^l\texttt{skip}\,]\!]$ (cf. Figure 3.9), we have $\gamma_{\mathrm{D}}(R) \subseteq \gamma_{\mathrm{D}}(D)$ which implies $\rho \in \gamma_{\mathrm{D}}(D)$. Moreover, since $\rho \in \mathrm{dom}(A)$ and by definition of $\textsc{step}_{\mathrm{T}}$ (cf. Algorithm 9), we must have $\rho \in \mathrm{dom}(\gamma_{\mathrm{T}}[D]t)$. In fact, Algorithm 9 simply invokes $\textsc{step}_{\mathrm{F}}$ for every leaf node of a decision tree, which leaves undefined leaf nodes unaltered (cf. Equation 5.2.19). Thus, by definition of $\tau_{\mathrm{Mt}}[\![\,^l\texttt{skip}\,]\!]$ (cf. Equation 4.3.1), we have $\rho \in \mathrm{dom}(C)$, which is absurd. Therefore, we have $\mathrm{dom}(C) \supseteq \mathrm{dom}(A)$.

Let us assume now, by absurd, that $\exists \rho \in \mathrm{dom}(A) : C(\rho) > A(\rho)$. We have, by definition of $\tau_{\mathrm{Mt}}[\![\,^l\texttt{skip}\,]\!]$, $C(\rho) = (\gamma_{\mathrm{T}}[D]t)(\rho) + 1$. Moreover, by definition of $\textsc{step}_{\mathrm{T}}$ (cf. Algorithm 9), we have $(\gamma_{\mathrm{T}}[D]t)(\rho) < A(\rho)$. In fact, Algorithm 9 invokes $\textsc{step}_{\mathrm{F}}$, which increases the value of the defined leaf nodes of a decision tree (cf. Equation 5.2.19). Thus, $C(\rho) \leq A(\rho)$, which is absurd. Therefore, we conclude that $\forall \rho \in \mathrm{dom}(A) : C(\rho) \leq A(\rho)$.

This concludes the proof. ■

Lemma 5.2.14 $\tau_{Mt}[\![\,^l X := aexp\,]\!]\gamma_D[D]t \preccurlyeq \gamma_T[R]((\textsc{b-assign}_T[\![\,X := aexp\,]\!]R)(t))$.

Proof of Lemma 5.2.14.

We have $\tau_{\mathrm{I}}[\![\,^l X := aexp\,]\!]\gamma_{\mathrm{D}}(R) \subseteq \gamma_{\mathrm{D}}(D)$, since $R \in \mathcal{D}$ and $D \in \mathcal{D}$ are sound over-approximations of $\tau_{\mathrm{I}}(l)$ and $\tau_{\mathrm{I}}(f[\![\,^l X := aexp\,]\!])$, respectively.

Let $C \stackrel{\text{def}}{=} \tau_{\mathrm{Mt}}[\![\,^l X := aexp\,]\!]\gamma_{\mathrm{D}}[D]t$ and let $A \stackrel{\text{def}}{=} \gamma_{\mathrm{T}}[R]((\textsc{b-assign}_{\mathrm{T}}[\![\,X := aexp\,]\!]R)(t))$. We prove that $\mathrm{dom}(C) \supseteq \mathrm{dom}(A)$ and $\forall \rho \in \mathrm{dom}(A) : C(\rho) \leq A(\rho)$ (cf. Equation 4.2.12).

Let us assume, by absurd, that $\mathrm{dom}(C) \subset \mathrm{dom}(A)$. Then, there exists an environment $\rho \in \mathcal{E}$ such that $\rho \in \mathrm{dom}(A)$ and $\rho \notin \mathrm{dom}(C)$. In particular, $\rho \in \gamma_{\mathrm{D}}(R)$. Thus, since $\tau_{\mathrm{I}}[\![\,^l X := aexp\,]\!]\gamma_{\mathrm{D}}(R) \subseteq \gamma_{\mathrm{D}}(D)$ and by definition of $\tau_{\mathrm{I}}[\![\,^l X := aexp\,]\!]$ (cf. Figure 3.9), we have $\forall v \in [\![\,aexp\,]\!]\rho : \rho[X \leftarrow v] \in \gamma_{\mathrm{D}}(D)$. Moreover, since $\rho \in \mathrm{dom}(A)$ and by definition of $\textsc{b-assign}_{\mathrm{T}}$ (cf. Algorithm 11), we must have $\forall v \in [\![\,aexp\,]\!]\rho : \rho[X \leftarrow v] \in \mathrm{dom}(\gamma_{\mathrm{T}}[D]t)$. In fact, an environment $\rho[X \leftarrow z] \notin \mathrm{dom}(\gamma_{\mathrm{T}}[D]t)$ for some $z \in [\![\,aexp\,]\!]\rho$, must be represented by an undefined leaf node $f \in \mathcal{F}\backslash\{\bot_{\mathrm{F}}, \top_{\mathrm{F}}\}$ in $t \in \mathcal{T}$ (cf. Equation 5.2.11 and Equation 5.2.9). Moreover, Algorithm 11 invokes $\textsc{b-assign}_{\mathrm{F}}$ for every leaf node of a decision tree, which leaves undefined leaf nodes unaltered (cf. Equation 5.2.21), and $\textsc{a-join}$ to handle possibly overlapping partitions (cf. Line 10 and Line 20), which favors undefined leaf nodes over defined leaf nodes (cf. Algorithm 6 and Equation 5.2.13). Thus, by definition of $\tau_{\mathrm{Mt}}[\![\,^l X := aexp\,]\!]$ (cf. Equation 4.3.2) and in absence of run-time errors, we have $\rho \in \mathrm{dom}(C)$, which is absurd. Therefore, we conclude that $\mathrm{dom}(C) \supseteq \mathrm{dom}(A)$.

Let us assume now, by absurd, that $\exists \rho \in \mathrm{dom}(A) : C(\rho) > A(\rho)$. We have, by definition of $\tau_{\mathrm{Mt}}[\![\,^l X := aexp\,]\!]$, $C(\rho) = \sup\{(\gamma_{\mathrm{T}}[D]t)(\rho[X \to v]) + 1 \mid v \in$

$[\![aexp]\!]\rho\}$. Moreover, by definition of B-ASSIGN$_\mathrm{T}$ (cf. Algorithm 11), we have $\sup\{(\gamma_\mathrm{T}[D]t)(\rho[X \to v]) + 1 \mid v \in [\![aexp]\!]\rho\} \leq A(\rho)$. In fact, Algorithm 11 invokes B-ASSIGN$_\mathrm{F}$, which after the assignment increases the value of the defined leaf nodes of a decision tree (cf. Equation 5.2.21). Thus, $C(\rho) \leq A(\rho)$, which is absurd. Therefore, we conclude that $\forall \rho \in \mathrm{dom}(A) : C(\rho) \leq A(\rho)$.

This concludes the proof. ■

Lemma 5.2.15 *Let* $F_1^\natural[t] \stackrel{def}{=} (\mathrm{FILTER}_T[\![bexp]\!]R)(\tau_{Mt}^\natural[\![stmt_1]\!]t)$ *and* $F_2^\natural[t] \stackrel{def}{=} (\mathrm{FILTER}_T[\![\texttt{not}\ bexp]\!]R)(\tau_{Mt}^\natural[\![stmt_2]\!]t)$. *Then, for all* $t \in \mathcal{T}$, *we have:*

$$\tau_{Mt}[\![\texttt{if}\ ^l bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi}]\!]\gamma_T[D]t \preccurlyeq \gamma_T[R](F_1^\natural[t] \curlyvee_T[R] F_2^\natural[t])$$

Proof of Lemma 5.2.15.

We have $\tau_\mathrm{I}[\![\texttt{if}\ ^l bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi}]\!]\gamma_\mathrm{D}(R) \subseteq \gamma_\mathrm{D}(D)$, given that $R \in \mathcal{D}$ is a sound over-approximation of $\tau_\mathrm{I}(l)$ and $D \in \mathcal{D}$ is a sound over-approximations of $\tau_\mathrm{I}(f[\![\texttt{if}\ ^l bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi}]\!])$.

Let $C \stackrel{\mathrm{def}}{=} \tau_\mathrm{Mt}[\![\texttt{if}\ ^l bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi}]\!]\gamma_\mathrm{T}[D]t$ and let $A \stackrel{\mathrm{def}}{=} \gamma_\mathrm{T}[R](F_1^\natural[t] \curlyvee_\mathrm{T}[R] F_2^\natural[t])$. By structural induction, we have $\tau_\mathrm{Mt}[\![stmt_1]\!]\gamma_\mathrm{T}[D]t \preccurlyeq \gamma_\mathrm{T}[R](\tau_\mathrm{Mt}^\natural[\![stmt_1]\!]t)$ and $\tau_\mathrm{Mt}[\![stmt_2]\!]\gamma_\mathrm{T}[D]t \preccurlyeq \gamma_\mathrm{T}[R](\tau_\mathrm{Mt}^\natural[\![stmt_2]\!]t)$. We prove that $\mathrm{dom}(C) \supseteq \mathrm{dom}(A)$ and $\forall \rho \in \mathrm{dom}(A) : C(\rho) \leq A(\rho)$ (cf. Equation 4.2.12).

Let us assume, by absurd, that $\mathrm{dom}(C) \subset \mathrm{dom}(A)$. Then, there exists an environment $\rho \in \mathcal{E}$ such that $\rho \in \mathrm{dom}(A)$ and $\rho \notin \mathrm{dom}(C)$. Since $\rho \in \mathrm{dom}(A)$, by definition of $\preccurlyeq_\mathrm{T}[D]$ (cf. Algorithm 6) and FILTER$_\mathrm{T}$ (cf. Algorithm 12), we have $\rho \in \mathrm{dom}(\gamma_\mathrm{T}[R](\tau_\mathrm{Mt}^\natural[\![stmt_1]\!]t))$ or $\rho \in \mathrm{dom}(\gamma_\mathrm{T}[R](\tau_\mathrm{Mt}^\natural[\![stmt_2]\!]t))$, or both. In fact, Algorithm 12 prunes a decision tree (cf. Line 12) and Algorithm 6 favors undefined leaf nodes over defined leaf nodes (cf. Equation 5.2.13). Thus, by structural induction, we have $\rho \in \mathrm{dom}(\tau_\mathrm{Mt}[\![stmt_1]\!]\gamma_\mathrm{T}[D]t)$ or $\rho \in \mathrm{dom}(\tau_\mathrm{Mt}[\![stmt_2]\!]\gamma_\mathrm{T}[D]t)$, or both. In absence of run-time errors, by definition of $\tau_\mathrm{Mt}[\![\texttt{if}\ ^l bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi}]\!]$ (cf. Equation 4.3.3), we have $\rho \in \mathrm{dom}(C)$ which is absurd. Therefore, we conclude that $\mathrm{dom}(C) \supseteq \mathrm{dom}(A)$.

Let us assume now, by absurd, that $\exists \rho \in \mathrm{dom}(A) : C(\rho) > A(\rho)$. Let S_1 denote $\tau_\mathrm{Mt}[\![stmt_1]\!]\gamma_\mathrm{T}[D]t$ and let S_2 denote $\tau_\mathrm{Mt}[\![stmt_2]\!]\gamma_\mathrm{T}[D]t$. We have, by definition of $\tau_\mathrm{Mt}[\![\texttt{if}\ ^l bexp\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{fi}$, $C(\rho) = S_1(\rho) + 1$ or $C(\rho) = S_2(\rho) + 1$ or $C(\rho) = \sup\{S_1(\rho) + 1, S_2(\rho) + 1\}$. Let $S_1^\natural$ denote $\gamma_\mathrm{T}[R](\tau_\mathrm{Mt}^\natural[\![stmt_1]\!]t)$ and let $S_2^\natural$ denote $\gamma_\mathrm{T}[R](\tau_\mathrm{Mt}^\natural[\![stmt_2]\!]t)$. We have, by structural induction, $S_1(\rho) < A(\rho)$ or $S_2(\rho) < A(\rho)$ or $\sup\{S_1(\rho) + 1, S_2(\rho) + 1\} < A(\rho)$. In fact, Algorithm 12 invokes STEP which increases the value of the defined leaf nodes of a decision tree (cf. Algorithm 9. Thus, $C(\rho) \leq A(\rho)$, which is absurd. Therefore, we conclude that $\forall \rho \in \mathrm{dom}(A) : C(\rho) \leq A(\rho)$.

This concludes the proof. ■

Lemma 5.2.16 *Let* $F_1^\natural[x] \stackrel{def}{=} (\text{FILTER}_T[\![\, bexp \,]\!]R)(\tau_{Mt}^\natural[\![\, stmt \,]\!]x)$ *and* $F_2^\natural[t] \stackrel{def}{=} (\text{FILTER}_T[\![\, \texttt{not}\ bexp \,]\!]R)(t)$. *Then, given* $t \in \mathcal{T}$, *for all* $x \in \mathcal{T}$ *we have:*

$$\phi_{Mt}(\gamma_D[R]x) \preccurlyeq \gamma_T[R](\phi_{Mt}^\natural(x))$$

where $\phi_{Mt}^\natural(x) \stackrel{def}{=} F_1^\natural[x] \curlyvee_T[R]\ F_2^\natural[t]$.

Proof of Lemma 5.2.16.
Let $C \stackrel{\text{def}}{=} \phi_{\text{Mt}}(\gamma_{\text{D}}[R]x)$ and let $A \stackrel{\text{def}}{=} \gamma_{\text{T}}[R](\phi_{\text{Mt}}^\natural(x))$. By structural induction, we have $\tau_{\text{Mt}}[\![\, stmt \,]\!]\gamma_{\text{T}}[R]x \preccurlyeq \gamma_{\text{T}}[R](\tau_{\text{Mt}}^\natural[\![\, stmt \,]\!]x)$. We prove that $\text{dom}(C) \supseteq \text{dom}(A)$ and $\forall \rho \in \text{dom}(A) : C(\rho) \leq A(\rho)$ (cf. Equation 4.2.12).

Let us assume, by absurd, that $\text{dom}(C) \subset \text{dom}(A)$. Then, there exists an environment $\rho \in \mathcal{E}$ such that $\rho \in \text{dom}(A)$ and $\rho \notin \text{dom}(C)$. Since $\rho \in \text{dom}(A)$, by definition of $\preccurlyeq_{\text{T}}[D]$ (cf. Algorithm 6) and FILTER_{T} (cf. Algorithm 12), we have $\rho \in \text{dom}(\gamma_{\text{T}}[R](\tau_{\text{Mt}}^\natural[\![\, stmt \,]\!]x))$ or $\rho \in \text{dom}(\gamma_{\text{T}}[D]t)$, or both. In fact, Algorithm 12 prunes a decision tree (cf. Line 12) and Algorithm 6 favors undefined leaf nodes over defined leaf nodes (cf. Equation 5.2.13). Thus, by structural induction, we have $\rho \in \text{dom}(\tau_{\text{Mt}}[\![\, stmt \,]\!]\gamma_{\text{T}}[R]x)$ or $\rho \in \text{dom}(\gamma_{\text{T}}[D]t)$, or both. In absence of run-time errors, by definition of ϕ_{Mt} (cf. Equation 4.3.5), we have $\rho \in \text{dom}(C)$ which is absurd. Therefore, we conclude that $\text{dom}(C) \supseteq \text{dom}(A)$.

Let us assume now, by absurd, that $\exists \rho \in \text{dom}(A) : C(\rho) > A(\rho)$. Let S denote $\tau_{\text{Mt}}[\![\, stmt \,]\!]\gamma_{\text{T}}[R]x$. We have, by definition of ϕ_{Mt}, $C(\rho) = S_1(\rho) + 1$ or $C(\rho) = (\gamma_{\text{T}}[D]t)(\rho) + 1$ or $C(\rho) = \sup\{S_1(\rho) + 1, (\gamma_{\text{T}}[D]t)(\rho) + 1\}$. Let $S^\natural$ denote $\gamma_{\text{T}}[R](\tau_{\text{Mt}}^\natural[\![\, stmt \,]\!]x)$. We have, by structural induction, $S_1(\rho) < A(\rho)$ or $(\gamma_{\text{T}}[D]t)(\rho) < A(\rho)$ or $\sup\{S_1(\rho) + 1, (\gamma_{\text{T}}[D]t)(\rho) + 1\} < A(\rho)$. In fact, Algorithm 12 invokes STEP which increases the value of the defined leaf nodes of a decision tree (cf. Algorithm 9. Thus, $C(\rho) \leq A(\rho)$, which is absurd. Therefore, we conclude that $\forall \rho \in \text{dom}(A) : C(\rho) \leq A(\rho)$.

This concludes the proof. ■

Lemma 5.2.24 *Let* $\phi_{Mt}^\natural(x) \stackrel{def}{=} F_1^\natural[x] \curlyvee_T[R]\ F_2^\natural[t]$ *as defined in Lemma 5.2.16 for any given* $t \in \mathcal{T}$. *Then, we have:*

$$\tau_{Mt}[\![\, \texttt{while}\ ^l bexp\ \texttt{do}\ stmt\ \texttt{od} \,]\!]\gamma_T[D]t \preccurlyeq \gamma_T[R](\text{lfp}^\natural\ \phi_{Mt}^\natural)$$

where $\text{lfp}^\natural\ \phi_{Mt}^\natural$ *is the limit of the iteration sequence with widening:* $y_0 \stackrel{def}{=} \bot_T$, $y_{n+1} \stackrel{def}{=} y_n \triangledown \phi_{Mt}^\natural(y_n)$ *(cf. Equation 5.2.24).*

Proof of Lemma 5.2.24.

Let $C \stackrel{\text{def}}{=} \tau_{\text{Mt}}[\![\texttt{while}\ ^{l}\mathit{bexp}\ \texttt{do}\ \mathit{stmt}\ \texttt{od}]\!]\gamma_{\text{T}}[D]t$ and let $A \stackrel{\text{def}}{=} \gamma_{\text{T}}[R](\text{lfp}^{\natural}\ \phi^{\natural}_{\text{Mt}})$. We prove that $\text{dom}(C) \supseteq \text{dom}(A)$ and $\forall \rho \in \text{dom}(A) : C(\rho) \leq A(\rho)$ (cf. Equation 4.2.12).

The proof follows from Lemma 5.2.16, Lemma 5.2.17, and Lemma 5.2.19, and Lemma 5.2.20 and from the definition of ∇_{T} (cf. Algorithm 13). In fact from Lemma 5.2.16, whenever some iterate y_n under-approximates the value of the termination semantics or over-approximates its domain of definition (cf. Figure 5.10), it cannot be the limit of the iteration sequence with widening because it violates either $\phi^{\natural}_{\text{Mt}}(y_n) \sqsubseteq_{\text{T}}[R]\ y_n$ or $\phi^{\natural}_{\text{Mt}}(y_n) \preccurlyeq_{\text{T}}[R]\ y_n$ (cf. Equation 5.2.24). Moreover, from Lemma 5.2.17, and Lemma 5.2.19, and Lemma 5.2.20 and from the definition of ∇_{T} (cf. Algorithm 13), we know that further iterates resolve the issue. In fact, Algorithm 13 specifically invokes CASEA and CASEBORC to this end.

Thus, this concludes the proof since the limit of the iteration sequence with widening must over-approximate the value of the termination semantics and under-approximate its domain of definition. ■

A.4 Missing Proofs from Chapter 6

Lemma 6.3.1 $\forall p_1, p_2 \in \mathcal{W} : p_1 \preccurlyeq_W [D]\ p_2 \Rightarrow \gamma_W[D]p_1 \preccurlyeq \gamma_W[D]p_2$.

Proof of Lemma 6.3.1.

Let $D \in \mathcal{D}$. We reason by cases. First, we consider the case of defined and undefined leaf nodes. Then, we consider the case of defined leaf nodes.

Let $p_1 \in \mathcal{W} \setminus \{\bot_{\text{W}}, \top_{\text{W}}\}$ and let $p_2 \in \{\bot_{\text{W}}, \top_{\text{W}}\}$. We have $p_1 \preccurlyeq_{\text{W}} [D]p_2$ from Figure 6.2a. Moreover, from Equation 6.3.4, we have $\gamma_{\text{W}}[D]p_2 = \emptyset$. Thus, since $\text{dom}(\gamma_{\text{W}}[D]p_2) = \emptyset$, we have $\text{dom}(\gamma_{\text{W}}[D]p_1) \supseteq \text{dom}(\gamma_{\text{W}}[D]p_2)$ and, from Equation 4.2.12, we have $\gamma_{\text{W}}[D]p_1 \preccurlyeq \gamma_{\text{W}}[D]p_2$.

Let $p_1, p_2 \in \mathcal{W} \setminus \{\bot_{\text{W}}, \top_{\text{W}}\}$ such that $p_1 \preccurlyeq_{\text{W}} [D]p_2$, where $p_1 \stackrel{\text{def}}{=} \sum_i \omega^i \cdot f_{i_1}$ and $p_2 \stackrel{\text{def}}{=} \sum_i \omega^i \cdot f_{i_2}$. From Equation 6.3.4, we have $\text{dom}(\gamma_{\text{W}}[D]p_1) = \text{dom}(\gamma_{\text{W}}[D]p_2)$ and, in particular, $\text{dom}(\gamma_{\text{W}}[D]p_1) \supseteq \text{dom}(\gamma_{\text{W}}[D]p_2)$. Moreover, from Equation 6.3.2, for all $\rho \in \gamma_{\text{D}}(D)$ we have $\sum_i \omega^i \cdot f_{i_1}(\rho(X_1), \ldots, \rho(X_k)) \leq \sum_i \omega^i \cdot f_{i_2}(\rho(X_1), \ldots, \rho(X_k))$. Thus, from Equation 4.2.12, $\gamma_{\text{W}}[D]p_1 \preccurlyeq \gamma_{\text{W}}[D]p_2$.

This concludes the proof that $\gamma_{\text{W}}[D]$ is monotonic. ■

Lemma 6.4.1 $\tau_{Mt}[\![\mathit{stmt}]\!](\gamma_T[D]t) \ \preccurlyeq\ \gamma_T[R](\tau^{\natural}_{Mt}[\![\mathit{stmt}]\!]t)$.

Proof of Lemma 6.4.1 (Sketch).
The proof for a `skip` instruction follows from Lemma 5.2.8 and the definition of the STEP_W operator (cf. Equation 6.3.8). The proof for a variable assignment follows from Lemma 5.2.14 and the definition of the B-ASSIGN_W operator (cf. Algorithm 21). The proof for a conditional `if` instruction follows from Lemma 5.2.15 and the definition of the approximation join $\curlyvee_\text{W}$ (cf. Algorithm 19). The proof for a `while` loop follows from Lemma 5.2.24 and the definition of the computational join $\sqcup_\text{W}$ (cf. Algorithm 20) and the extrapolation operator $\blacktriangledown_\text{W}$ (cf. Algorithm 22). ■

A.5 Missing Proofs from Chapter 7

Lemma 7.3.3 $(\tau_{Mt}[\![^l\texttt{call}\ M]\!]_M\gamma_T[S]T)(\gamma_T[D]t) \preccurlyeq \gamma_T[R](\text{STEP}_T(t +_T T))$.

Proof of Lemma 7.3.3.
Let $C \stackrel{\text{def}}{=} (\tau_\text{Mt}[\![^l\texttt{call}\ M]\!]_M\gamma_\text{T}[S]T)(\gamma_\text{T}[D]t)$ and let $A \stackrel{\text{def}}{=} \gamma_\text{T}[R](\text{STEP}_\text{T}(t +_\text{T} T))$. We prove that $\text{dom}(C) \supseteq \text{dom}(A)$ and $\forall\rho \in \text{dom}(A) : C(\rho) \leq A(\rho)$ (cf. Equation 4.2.12).

Let us assume, by absurd, that $\text{dom}(C) \subset \text{dom}(A)$. Then, there exists an environment $\rho \in \mathcal{E}$ such that $\rho \in \text{dom}(A)$ and $\rho \notin \text{dom}(C)$. Thus, since $\rho \in \text{dom}(A)$ and by definition of the step operator STEP_T (cf. Algorithm 9) and the sum operator $+_\text{T}$ (cf. Algorithm 23), we must have $\rho \in \text{dom}(\gamma_\text{T}[D]t)$ and $\rho \in \text{dom}(\gamma_\text{T}[D]T)$. In fact, Algorithm 23 invokes the leaves sum operator $+_\text{F}$, which favors undefined leaf nodes over defined leaf nodes (cf. Equation 7.3.6 and Equation 7.3.7) and Algorithm 9 simply invokes STEP_F for every leaf node of a decision tree, which leaves undefined leaf nodes unaltered (cf. Equation 5.2.19). Thus, by definition of $\tau_\text{Mt}[\![^l\texttt{call}\ M]\!]_M$ (cf. Equation 7.3.1), we have $\rho \in \text{dom}(C)$, which is absurd. Therefore, we have $\text{dom}(C) \supseteq \text{dom}(A)$.

Let us assume now, by absurd, that $\exists\rho \in \text{dom}(A) : C(\rho) > A(\rho)$. We have, by definition of $\tau_\text{Mt}[\![^l\texttt{call}\ M]\!]_M$, $C(\rho) = (\gamma_\text{T}[D]t)(\rho) + (\gamma_\text{T}[D]T)(\rho) + 1$. Moreover, by definition of STEP_T (cf. Algorithm 9) and $+_\text{T}$ (cf. Algorithm 23), we have $(\gamma_\text{T}[D]t)(\rho) + (\gamma_\text{T}[D]T)(\rho) < A(\rho)$. In fact, Algorithm 23 invokes $+_\text{F}$, which additions the value of the leaf nodes that are defined in both decision trees (cf. Equation 7.3.7), and Algorithm 9 invokes STEP_F, which increases the value of the resulting defined leaf nodes (cf. Equation 5.2.19). Thus, $C(\rho) \leq A(\rho)$, which is absurd. Therefore, $\forall\rho \in \text{dom}(A) : C(\rho) \leq A(\rho)$.

This concludes the proof. ■

Lemma 7.3.4 $(\tau_{Mt}[\![^l\texttt{call}\ M]\!]_P\gamma_T[S]T)(\gamma_T[D]t) \preccurlyeq \gamma_T[R](\textsc{step}_T(I))$, *where* $I \stackrel{def}{=} \text{lfp}^\natural\ (\lambda T.\ (\tau^\natural_{Mt}(M)T)(t))$.

Proof of Lemma 7.3.4.
The soundness of I follows from the definition of $(\tau_{\text{Mt}}[\![^l\texttt{call}\ M]\!]_P$ (cf. Equation 7.3.2) and from Lemma 5.2.24.

Let $C \stackrel{\text{def}}{=} (\tau_{\text{Mt}}[\![^l\texttt{call}\ M]\!]_P\gamma_{\text{T}}[S]T)(\gamma_{\text{T}}[D]t)$ and let $A \stackrel{\text{def}}{=} \gamma_{\text{T}}[R]\textsc{step}_{\text{T}}(C)$. We prove that $\text{dom}(C) \supseteq \text{dom}(A)$ and $\forall\rho \in \text{dom}(A) : C(\rho) \leq A(\rho)$ (cf. Equation 4.2.12).

Let us assume, by absurd, that $\text{dom}(C) \subset \text{dom}(A)$. Then, there exists an environment $\rho \in \mathcal{E}$ such that $\rho \in \text{dom}(A)$ and $\rho \notin \text{dom}(C)$. Thus, since $\rho \in \text{dom}(A)$ and by definition of $\textsc{step}_{\text{T}}$ (cf. Algorithm 9), we must have $\rho \in \text{dom}(\gamma_{\text{T}}[D]t)$. In fact, Algorithm 9 simply invokes $\textsc{step}_{\text{F}}$ for every leaf node of a decision tree, which leaves undefined leaf nodes unaltered (cf. Equation 5.2.19). Thus, by definition of $(\tau_{\text{Mt}}[\![^l\texttt{call}\ M]\!]_P$, we have $\rho \in \text{dom}(C)$, which is absurd. Therefore, we have $\text{dom}(C) \supseteq \text{dom}(A)$.

Let us assume now, by absurd, that $\exists\rho \in \text{dom}(A) : C(\rho) > A(\rho)$. We have, by definition of $(\tau_{\text{Mt}}[\![^l\texttt{call}\ M]\!]_P$, $C(\rho) = (\gamma_{\text{T}}[D]t)(\rho) + 1$. Moreover, by definition of $\textsc{step}_{\text{T}}$ (cf. Algorithm 9), we have $(\gamma_{\text{T}}[D]t)(\rho) < A(\rho)$. In fact, Algorithm 9 invokes $\textsc{step}_{\text{F}}$, which increases the value of the defined leaf nodes of a decision tree (cf. Equation 5.2.19). Thus, $C(\rho) \leq A(\rho)$, which is absurd. Therefore, we conclude that $\forall\rho \in \text{dom}(A) : C(\rho) \leq A(\rho)$.

This concludes the proof. ■

Lemma 7.3.5 $(\tau_{Mt}[\![^l\texttt{return}]\!]_P\gamma_T[D]T)(\gamma_T[D]t) \preccurlyeq \gamma_T[R](\textsc{step}_T(t))$.

Proof of Lemma 7.3.5.
Let $C \stackrel{\text{def}}{=} (\tau_{\text{Mt}}[\![^l\texttt{return}]\!]_P\gamma_{\text{T}}[D]T)(\gamma_{\text{T}}[D]t)$ and let $A \stackrel{\text{def}}{=} \gamma_{\text{T}}[R]\textsc{step}_{\text{T}}(t)$. We prove that $\text{dom}(C) \supseteq \text{dom}(A)$ and $\forall\rho \in \text{dom}(A) : C(\rho) \leq A(\rho)$ (cf. Equation 4.2.12).

Let us assume, by absurd, that $\text{dom}(C) \subset \text{dom}(A)$. Then, there exists an environment $\rho \in \mathcal{E}$ such that $\rho \in \text{dom}(A)$ and $\rho \notin \text{dom}(C)$. Thus, since $\rho \in \text{dom}(A)$ and by definition of $\textsc{step}_{\text{T}}$ (cf. Algorithm 9), we must have $\rho \in \text{dom}(\gamma_{\text{T}}[D]t)$. In fact, Algorithm 9 simply invokes $\textsc{step}_{\text{F}}$ for every leaf node of a decision tree, which leaves undefined leaf nodes unaltered (cf. Equation 5.2.19). Thus, by definition of $\tau_{\text{Mt}}[\![^l\texttt{return}]\!]$ (cf. Equation 7.3.3), we have $\rho \in \text{dom}(C)$, which is absurd. Therefore, we have $\text{dom}(C) \supseteq \text{dom}(A)$.

Let us assume now, by absurd, that $\exists\rho \in \text{dom}(A) : C(\rho) > A(\rho)$. We have, by definition of $\tau_{\text{Mt}}[\![^l\texttt{return}]\!]$, $C(\rho) = (\gamma_{\text{T}}[D]t)(\rho) + 1$. Moreover, by definition of $\textsc{step}_{\text{T}}$ (cf. Algorithm 9), we have $(\gamma_{\text{T}}[D]t)(\rho) < A(\rho)$. In fact,

Algorithm 9 invokes STEP_F, which increases the value of the defined leaf nodes of a decision tree (cf. Equation 5.2.19). Thus, $C(\rho) \leq A(\rho)$, which is absurd. Therefore, we conclude that $\forall \rho \in \text{dom}(A) : C(\rho) \leq A(\rho)$.

This concludes the proof. ■

A.6 Missing Proofs from Chapter 9

Theorem 9.2.5 *A program satisfies a guarantee property $\Diamond\varphi$ for execution traces starting from a given set of initial states $\mathcal{I}$ if and only if $\mathcal{I} \subseteq \text{dom}(\tau_g^\varphi)$.*

Proof of Theorem 9.2.5 (Sketch).
The proof follows by Park's Fixpoint Induction Principle (cf. Theorem 2.2.14) along the lines of the proof of Theorem 4.2.23 proposed in [CC12] and the proof of Theorem 4.2.15 proposed in Appendix A.2. ■

Lemma 9.2.9 $\tau_g^\varphi[\![\, stmt \,]\!]\gamma_T[D]t \;\preccurlyeq\; \gamma_T[R]\tau_g^{\varphi\natural}[\![\, stmt \,]\!]t.$

Proof of Lemma 9.2.9 (Sketch).
The proof follows from Equation 9.2.7 and Lemma 5.2.8 (for a `skip` instruction), from Equation 9.2.8 and Lemma 5.2.14 (for a variable assignment), from Equation 9.2.9 and Lemma 5.2.15 (for a conditional `if` statement), and from Equation 9.2.10 and Lemma 5.2.24 (for a `while` loop), together with Lemma 6.4.1 and Algorithm 24 and Equation 9.2.14 (cf. Figure 9.1). ■

Theorem 9.3.5 *A program satisfies a guarantee property $\Diamond\varphi$ for execution traces starting from a given set of initial states $\mathcal{I}$ if and only if $\mathcal{I} \subseteq \text{dom}(\tau_g^\varphi)$.*

Proof of Theorem 9.3.5 (Sketch).
The proof follows by Park's Fixpoint Induction Principle (cf. Theorem 2.2.14) along the lines of the proof of Theorem 4.2.23 proposed in [CC12] and the proof of Theorem 4.2.15 proposed in Appendix A.2. ■

Lemma 9.3.9 $\tau_r^\varphi[\![\, stmt \,]\!]\gamma_T[D]t \;\preccurlyeq\; \gamma_T[R]\tau_r^{\varphi\natural}[\![\, stmt \,]\!]t.$

Proof of Lemma 9.3.9 (Sketch).
The proof follows from Equation 9.3.6 and Lemma 5.2.8 (for a `skip` instruction), from Equation 9.3.7 and Lemma 5.2.14 (for a variable assignment), from Equation 9.3.8 and Lemma 5.2.15 (for a conditional `if` statement), and from Equation 9.3.9 and Lemma 5.2.24 (for a `while` loop), together with Lemma 6.4.1 and Algorithm 24 and Equation 9.3.13 (cf. Figure 9.3). ■