



INTRODUCTION TO STATIC ANALYSIS

AN ABSTRACT INTERPRETATION PERSPECTIVE

XAVIER RIVAL AND KWANGKEUN YI



Introduction to Static Analysis

Introduction to Static Analysis

An Abstract Interpretation Perspective

Xavier Rival and Kwangkeun Yi

The MIT Press

Cambridge, Massachusetts

London, England

© 2020 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form or by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

Library of Congress Cataloging-in-Publication Data

Names: Rival, Xavier, author. | Yi, Kwangkeun, author.

Title: Introduction to static analysis : an abstract interpretation perspective /
Xavier Rival and Kwangkeun Yi.

Description: Cambridge, MA : The MIT Press, [2020] | Includes
bibliographical references and index.

Identifiers: LCCN 2019010151 | ISBN 9780262356657

Subjects: LCSH: Statics.

Classification: LCC TA351 .R58 2020 | DDC 620.1/03—dc23 LC record
available at <https://lccn.loc.gov/2019010151>

Author names are in alphabetical order. Both authors share equal
authorship.

d_r0

To our parents

Contents

Copyright

Preface

1 Program Analysis

- 1.1 Understanding Software Behavior
- 1.2 Program Analysis Applications and Challenges
- 1.3 Concepts in Program Analysis
 - 1.3.1 What to Analyze
 - 1.3.2 Static versus Dynamic
 - 1.3.3 A Hard Limit: Uncomputability
 - 1.3.4 Automation and Scalability
 - 1.3.5 Approximation: Soundness and Completeness
- 1.4 Families of Program Analysis Techniques
 - 1.4.1 Testing: Checking a Set of Finite Executions
 - 1.4.2 Assisted Proof: Relying on User-Supplied Invariants
 - 1.4.3 Model Checking: Exhaustive Exploration of Finite Systems
 - 1.4.4 Conservative Static Analysis: Automatic, Sound, and Incomplete Approach
 - 1.4.5 Bug Finding: Error Search, Automatic, Unsound, Incomplete, Based on Heuristics
 - 1.4.6 Summary
- 1.5 Roadmap

2 A Gentle Introduction to Static Analysis

- 2.1 Semantics and Analysis Goal: A Reachability Problem
- 2.2 Abstraction
- 2.3 A Computable Abstract Semantics: Compositional Style
 - 2.3.1 Abstraction of Initialization
 - 2.3.2 Abstraction of Post-Conditions
 - 2.3.3 Abstraction of Non-Deterministic Choice
 - 2.3.4 Abstraction of Non-Deterministic Iteration
 - 2.3.5 Verification of the Property of Interest
- 2.4 A Computable Abstract Semantics: Transitional Style
 - 2.4.1 Semantics as State Transitions
 - 2.4.2 Abstraction of States
 - 2.4.3 Abstraction of State Transitions
 - 2.4.4 Analysis by Global Iterations
- 2.5 Core Principles of a Static Analysis

3 A General Static Analysis Framework Based on a Compositional Semantics

- 3.1 Semantics
 - 3.1.1 A Simple Programming Language
 - 3.1.2 Concrete Semantics
- 3.2 Abstractions
 - 3.2.1 The Concept of Abstraction
 - 3.2.2 Non-Relational Abstraction
 - 3.2.3 Relational Abstraction
- 3.3 Computable Abstract Semantics
 - 3.3.1 Abstract Interpretation of Assignment
 - 3.3.2 Abstract Interpretation of Conditional Branching
 - 3.3.3 Abstract Interpretation of Loops
 - 3.3.4 Putting Everything Together
- 3.4 The Design of an Abstract Interpreter

4 A General Static Analysis Framework Based on a Transitional Semantics

- 4.1 Semantics as State Transitions
 - 4.1.1 Concrete Semantics
 - 4.1.2 Recipe for Defining a Concrete Transitional Semantics
- 4.2 Abstract Semantics as Abstract State Transitions
 - 4.2.1 Abstraction of the Semantic Domain
 - 4.2.2 Abstraction of Semantic Functions
 - 4.2.3 Recipe for Defining an Abstract Transition Semantics
- 4.3 Analysis Algorithms Based on Global Iterations
 - 4.3.1 Basic Algorithms
 - 4.3.2 Worklist Algorithm
- 4.4 Use Example of the Framework
 - 4.4.1 Simple Imperative Language
 - 4.4.2 Concrete State Transition Semantics
 - 4.4.3 Abstract State
 - 4.4.4 Abstract State Transition Semantics

5 Advanced Static Analysis Techniques

- 5.1 Construction of Abstract Domains
 - 5.1.1 Abstraction of Boolean-Numerical Properties
 - 5.1.2 Describing Conjunctive Properties
 - 5.1.3 Describing Properties Involving Case Splits
 - 5.1.4 Construction of an Abstract Domain
- 5.2 Advanced Iteration Techniques
 - 5.2.1 Loop Unrolling
 - 5.2.2 Fixpoint Approximation with More Precise Widening Iteration
 - 5.2.3 Refinement of an Abstract Approximation of a Least Fixpoint
- 5.3 Sparse Analysis
 - 5.3.1 Exploiting Spatial Sparsity
 - 5.3.2 Exploiting Temporal Sparsity
 - 5.3.3 Precision-Preserving Def-Use Chain by Pre-Analysis
- 5.4 Modular Analysis
 - 5.4.1 Parameterization, Summary, and Scalability

- 5.4.2 Case Study
- 5.5 Backward Analysis
 - 5.5.1 Forward Semantics and Backward Semantics
 - 5.5.2 Backward Analysis and Applications
 - 5.5.3 Precision Refinement by Combined Forward and Backward Analysis

6 Practical Use of Static Analysis Tools

- 6.1 Analysis Assumptions and Goals
- 6.2 Setting Up the Static Analysis of a Program
 - 6.2.1 Definition of the Source Code and Proof Goals
 - 6.2.2 Parameters to Guide the Analysis
- 6.3 Inspecting Analysis Results
- 6.4 Deployment of a Static Analysis Tool

7 Static Analysis Tool Implementation

- 7.1 Concrete Semantics and Concrete Interpreter
- 7.2 Abstract Domain Implementation
- 7.3 Static Analysis of Expressions and Conditions
- 7.4 Static Analysis Based on a Compositional Semantics
- 7.5 Static Analysis Based on a Transitional Semantics

8 Static Analysis for Advanced Programming Features

- 8.1 For a Language with Pointers and Dynamic Memory Allocations
 - 8.1.1 Language and Concrete Semantics
 - 8.1.2 An Abstract Semantics
- 8.2 For a Language with Functions and Recursive Calls
 - 8.2.1 Language and Concrete Semantics
 - 8.2.2 An Abstract Semantics
- 8.3 Abstractions for Data Structures
 - 8.3.1 Arrays
 - 8.3.2 Buffers and Strings
 - 8.3.3 Pointers
 - 8.3.4 Dynamic Heap Allocation
- 8.4 Abstraction for Control Flow Structures
 - 8.4.1 Functions and Procedures
 - 8.4.2 Parallelism

9 Classes of Semantic Properties and Verification by Static Analysis

- 9.1 Trace Properties
 - 9.1.1 Safety
 - 9.1.2 Liveness
 - 9.1.3 General Trace Properties
- 9.2 Beyond Trace Properties: Information Flows and Other Properties

10 Specialized Static Analysis Frameworks

- 10.1 Static Analysis by Equations
 - 10.1.1 Data-Flow Analysis

- 10.2 Static Analysis by Monotonic Closure
 - 10.2.1 Pointer Analysis
 - 10.2.2 Higher-Order Control-Flow Analysis
- 10.3 Static Analysis by Proof Construction
 - 10.3.1 Type Inference

11 Summary and Perspectives

A Reference for Mathematical Notions and Notations

- A.1 Sets
- A.2 Logical Connectives
- A.3 Definitions and Proofs by Induction
- A.4 Functions
- A.5 Order Relations and Ordered Sets
- A.6 Operators over Ordered Structures and Fixpoints

B Proofs of Soundness

- B.1 Properties of Galois Connections
- B.2 Proofs of Soundness for Chapter
 - B.2.1 Soundness of the Abstract Interpretation of Expressions
 - B.2.2 Soundness of the Abstract Interpretation of Conditions
 - B.2.3 Soundness of the Abstract Join Operator
 - B.2.4 Abstract Iterates with Widening
 - B.2.5 Soundness of the Abstract Interpretation of Commands
- B.3 Proofs of Soundness for Chapter
 - B.3.1 Transitional-Style Static Analysis over Finite-Height Domains
 - B.3.2 Transitional-Style Static Analysis with Widening
 - B.3.3 Use Example of the Transitional-Style Static Analysis

Bibliography

Index

Preface

Static program analysis or, for short, *static analysis* aims at discovering semantic properties of programs without running them. Initially, it was introduced in the seventies to enable compiler optimizations and guide the synthesis of efficient machine code. Since then, the interest in static analysis has grown considerably, as it plays an important role in all phases of program development, including the verification of specifications, the verification of programs, the synthesis of optimized code, and the refactoring and maintenance of software applications. It is commonly used to certify critical software. It also helps improve the quality of general-purpose software and enables aggressive code optimizations. Consequently, static analysis is now relevant not only to computer scientists who study the foundations of programming but also to many working engineers and developers who can use it for their everyday work.

The purpose of this book is to provide an introduction to static analysis and to cover the basics of both theoretical foundations and practical considerations underlying the implementation and use of static analysis tools. The scientific literature on static analysis is huge and may seem hard for students or working engineers to get started quickly. Indeed, while we can recommend great scientific articles on many specific topics, these are often too advanced for a general audience; thus, it is harder to find good references that could provide a quick and comprehensive introduction for nonexperts. The aim of this book is precisely to provide such a general introduction, which explains the key basic concepts (both in the theory and from a practical point of view), gives the gist of the state of the art in static program analysis, and can be read rather quickly and easily. More precisely, we cover the mathematical foundations of static analysis (semantics, semantic abstraction, computation of program invariants), more advanced notions and techniques (abstractions for advanced programming features

and for answering a wide range of semantic questions), and techniques to implement and use static analysis tools.

On the other hand, it is not possible to provide an exhaustive description of the state of the art, simply because the sheer number of scientific reports and articles published every year on the topic far exceeds what a single person can assimilate or will even need. A huge number of practical techniques and fundamental constructions could be studied, yet they may be useful to only a small group of readers. Therefore, we believe that exhaustivity should not be the purpose of this book, and it is likely that most readers will have to complement it with additional material that covers their personal interest. However, we hope that this book will help all readers quickly reach the point where they master the foundational principles of static analysis, and help them acquire additional knowledge more easily when they need it. To better achieve this, we try to present, in an intuitive manner, some ideas and principles that often take years of research and practical experience to fully understand and learn.

We divide our material into chapters that can often be skipped or omitted in the first reading, depending on the motivations of the reader. Furthermore, we distinguish several reader profiles, which we consider part of the target audience of this book and for whom we can provide specific advice on where to start and how to handle each chapter:

- **Students** (abbreviated as [S] in the introduction to each chapter) who follow a course on programming languages, compilers, program analysis, or program verification will find here material to construct a theoretical and practical knowledge of static analysis; we expect advanced students (e.g., graduate students specializing in program analysis or verification) to read most of the chapters and complement them with additional readings, whereas more junior students can focus on the first chapters.
- **Developers** ([D]) of programming tools (e.g., compilers and program verifiers) who need to implement static analysis techniques to solve practical problems will find both the technical basics that they need and practical examples with actual code to learn about common approaches to implementing static analyzers.

- **Users ([U])** of static analysis tools need to know about the general principles underlying static analyzers and how to set up an analysis; they will find not only general notions on static analysis but also a chapter devoted specifically to the use of static analysis tools, including their configuration and the exploitation of the analysis results.

Structure of the Book Chapters 1 and 2 provide a high-level introduction to the main principles of static analysis. Chapter 1 provides the background to understand the goals of static analysis and the kind of questions it can answer. In particular, it discusses the consequence of the fact that interesting semantic properties about programs are not computable. It also compares static analysis with other techniques for reasoning about programs. Chapter 2 presents an intuitive graphical introduction to the abstraction of program semantics and to the static analysis of programs. These two chapters are fundamental for all readers.

Chapters 3–5 formalize the scientific foundations of program analysis techniques, including the definition of the semantics of a programming language, the abstraction of the semantics of programs, and the computation of conservative program invariants. These chapters present not only the most general methods but also some advanced static analysis techniques. The notion of abstraction is central to the whole book as it defines the logical predicates that a static analysis may use to compute program properties; thus, it is extensively studied here. Chapters 3 and 4 cover the notions of semantics, abstraction, and abstract interpretation and show how they work using a toy language as an example. Chapter 5 describes a few advanced methods to construct abstractions and static analysis algorithms. Readers who seek a comprehensive understanding of the foundations should read these three chapters. On the other hand, users of static analysis tools may skip part of this material in a first reading and refer back to these chapters when the need arises. Developers of static analyzers may take a similar approach and skip chapter 5 in a first reading, returning to it when needed.

The following two chapters focus on the practical implementation and use of static analysis techniques. Chapter 6 discusses the practical use of a static analysis tool, its configuration, and how the analysis results can be

used to answer specific semantic questions. It is particularly targeted at users of static analyzers, even though its content should be useful to all readers. Chapter 7 reviews the implementation of a simple static analyzer, provides source code, and discusses implementation choices. It was written for readers who intend to develop static analyzers, although other readers may also find it useful for gaining deeper understanding of how a static analyzer works. Readers who are mainly interested in the foundations may skip these two chapters, at least in the first reading.

The next two chapters consider more advanced applications. Chapter 8 reviews more advanced programming language features than those included in the examples shown in chapters 3 and 4 and discusses abstractions that apply to them. Chapter 9 studies more complex semantic properties than those considered in earlier chapters and highlights abstractions to cope with them. Chapter 10 discusses a few specialized frameworks that are less general than those presented in chapters 3 and 4, yet take advantage of specific programming language features or properties of interest to build efficient static analyses. These chapters often focus on the high-level ideas and main abstractions to solve a class of problems. We expect readers interested in these issues to pursue reading of additional material.

Finally, chapter 11 summarizes the main concepts of static analysis, several routes for further studies, and a few challenges for the future.

For reference, appendix A recalls standard mathematical notations used throughout the book (to make it accessible even without a background in discrete mathematics and logics), and appendix B collects the proofs of the theorems presented in the core of the formal chapters.

Acknowledgments

We are grateful to the many people who helped bring this book to fruition. First and foremost, we thank all the great researchers and professors who contributed to the emergence and development of the static analysis field. In particular, Patrick Cousot and Radhia Cousot have built a very robust, powerful, and general framework for designing static analysis and more generally for reasoning about program semantics. Since these foundational works have been achieved, many research scientists, developers, engineers, and pioneer end users have contributed to the development and adoption of static analysis and have greatly advanced the state of the art. We would like to thank them all here.

Many colleagues have contributed directly or indirectly in the genesis of this book with discussions, ideas, and encouragement, and we thank them all, even though we cannot gather an exhaustive list. We especially thank Alex Aiken, Bor-Yuh Evan Chang, Cezara Drăgoi, Jérôme Feret, Kihong Heo, Chung-kil Hur, Woosuk Lee, Mayur Naik, Peter O’Hearn, Hakjoo Oh, Sukyoung Ryu, Makoto Tatsuda, and Hongseok Yang. We also thank the anonymous reviewers commissioned by the publisher, who were extremely helpful with their productive comments and suggestions.

This book is based on our work throughout our careers, which was supported by a series of research grants. Kwangkeun is grateful to National Research Foundation of Korea for continued and generous grants, including the National Creative Research Initiatives and the Engineering Research Center of Excellence. Kwangkeun also thanks Samsung Electronics and SparrowFasoo.com for their support, which drove his team to go the extra mile to industrialize a static analysis tool beyond the laboratory workbench. Xavier is grateful to the French Agence Nationale de la Recherche and the European Research Council for their support.

We would also like to acknowledge the organizations that supported the building of this book, including Seoul National University, the Institut

National de Recherche en Informatique et Automatique (INRIA), the Centre National de la Recherche Scientifique (CNRS), the École Normale Supérieure de Paris, and Université Paris Sciences et Lettres (PSL University). We are also grateful to MIT Press editors Marie Lee and Stephanie Cohen for their professional assistance in realizing this book project since our proposal.

Last, but not least, Kwangkeun expresses his deep appreciation for his wife Young-ae, for her love and care, and both authors gratefully acknowledge the love and encouragement of their family members and relatives.

1 Program Analysis

Goal of This Chapter Before we dive into the way static analysis tools operate, we need to define their scope and describe the kinds of questions they can help solve. In section 1.1 we discuss the importance of understanding the behavior of programs by semantic reasoning, and we show applications in section 1.2. Section 1.3 sets up the main concepts of static analysis and shows the intrinsic limitations of automatic program reasoning techniques based on semantics. Section 1.4 classifies the main approaches to semantic-based program reasoning and clarifies the position of static analysis in this landscape.

Recommended Reading: [S], [D], [U]

1.1 Understanding Software Behavior

In every engineering discipline, a fundamental question is, will our design work in reality as we intended? We ask and answer that question when we design mechanical machines, electrical circuits, or chemical processes. The answer comes from analyzing our designs using our knowledge about nature that will carry out the designs. For example, using Newtonian mechanics, Maxwell equations, Navier-Stokes equations, or thermodynamic equations, we analyze our design to predict its actual behavior. When we design a bridge, for example, we analyze how nature runs the design, namely, how various forces (e.g., gravitation, wind, and vibration) are applied to the bridge and whether the bridge structure is strong enough to withstand them.

The same question applies to computer software. We want to ensure that our software will work as intended. The intention for analysis ranges widely. For general reliability, we want to ensure that the software will not crash with abrupt termination. If the software interacts with the outside world, we want to ensure it will not be deceived to violate the host computer's security. For specific functionalities, we want to check if the software will realize its functional goal. If the software is to control cars, we want to ensure it will not drive them to an accident. If the software is to learn our preference, we want to ensure it will not

degrade as we teach more. If the software transforms the medical images of our bodies, we want to ensure it will not introduce false pixels. If the software is to bookkeep the ledgers for crypto currency, we want to ensure it will not allow double spending. If the software translates program text, we want to ensure the source's meaning is not lost in translation.

There is, however, one difference between analyzing software and analyzing other types of engineering designs: for computer software, it is not nature that will run the software but the computer itself. The computer will execute the software according to the meanings of the software's source language. Software's run-time behavior is solely defined by the meanings of the software's source language. The computer is just an undiscerning tool that blindly executes the software exactly as it is written. Any execution behavior that deviates from our intention is because the software is mistakenly written to behave that way.

Hence, for computer software, to answer the question of whether our design will work as we intended, we need knowledge by which we can somehow analyze the meanings of software source language. Such knowledge corresponds to knowledge that natural sciences have accumulated about nature. We need knowledge that computer science has accumulated about handling the meanings of software source languages.

We call a formal definition of a software's run-time behavior, which is determined by its source language's meanings, *semantics*:

Definition 1.1 (Semantics and semantic properties) *The semantics of a program is a (generally formal—although we do not make it so in this chapter) description of its run-time behaviors. We call semantic property any property about the run-time behavior (semantics) of a program.*

Hence, *checking if a software will run as we intended* is equivalent to *checking if this software satisfies a semantic property of interest*.

In the following, we call a technique to check that a program satisfies a semantic property *program analysis*, and we refer to an implementation of program analysis as a *program analysis tool*.

Figure 1.1 illustrates the correspondence between program analysis and design analysis of other engineering disciplines.

1.2 Program Analysis Applications and Challenges

Program analysis can be applied wherever understanding program semantics is important or beneficial. First, software developers (both humans and machines) may be the biggest beneficiaries. Software developers can use program analysis for quality assurance, to locate errors of any kind in their software. Software maintainers can use program analysis to understand legacy software that they

maintain. System security gatekeepers can use program analysis to proactively screen out programs whose semantics can be malicious.

Software that handles programs as data can use program analysis for the programs' performance improvement too. Language processors such as translators or compilers need program analysis to translate the input programs into optimized ones. Interpreters, virtual machines, and query processors need program analysis for optimized execution of the input programs. Automatic program synthesizers can use program analysis to check and tune what they synthesize. Mobile operating systems need to understand an application's semantics in order to minimize the energy consumption of the application. Automatic tutoring systems for teaching programming can use program analysis to hint to students a direction to amend their faulty programs.

| | Computing area | Other engineering areas |
|-------------------|---------------------------|--|
| Object | Software | Machine/building/circuit/chemical process design |
| Execution subject | Computer runs it | Nature runs it |
| Our question | Will it work as intended? | Will it work as intended? |
| Our knowledge | Program analysis | Newtonian mechanics, Maxwell equations, Navier-Stokes equations, thermodynamic equations, and other principles |

Figure 1.1

Program analysis addresses a basic question common in every engineering area

Use of program analysis is not limited to professional software or its developers. As programming becomes a way of living in a highly connected digitized environment, citizen programmers can benefit from program analysis, too, to sanity-check their daily program snippets.

The target of program analysis is not limited to executable software, either. Once the object's source language has semantics, program analysis can circumscribe its semantics to provide useful information. For example, program analysis of high-level system configuration scripts can provide information about any existing conflicting requests.

Though the benefits of program analysis are obvious, building a cost-effective program analysis is not trivial, since computer programs are complex and often

very large. For example, the number of lines of smartphone applications frequently reaches over half a million, not to mention larger software such as web browsers or operating systems, whose source sizes are over ten million lines. With semantics, the situation is much worse because a program execution is highly dynamic. Programs usually need to react to inputs from external, uncontrolled environments. The number of inputs, not to mention the number of program states, that can arise in all possible use cases is so huge that software developers are likely to fail to handle some corner cases. The number easily can be greater than the number of atoms in the universe, for example. The space of the inputs keeps exploding as we want our software to do more things. Also, constraints that could keep software simple and small quickly diminish because of the ever-growing capacity of computer hardware.

Given that software is in charge of almost all infrastructures in our personal, social, and global life, the need for cost-effective program analysis technology is greater than ever before. We have already experienced a sequence of appalling accidents whose causes are identified as mistakes in software. Such accidents have occurred in almost all sectors, including space, medical, military, electric power transmission, telecommunication, security, transportation, business, and administration. The long list includes accidents, the large-scale Twitter outage (2016), the fMRI software error (2016) that invalidated fifteen years of brain research, the Heartbleed bug (2014) in the popular OpenSSL cryptographic library that allows attackers to read the memory of any server that uses certain instances of OpenSSL, the stack overflow issues that can explain the Toyota sudden unintended acceleration (2004–2014), the Northeast blackout (2003), the explosion of the Ariane 5 Flight 501 (1996), which took ten years and \$7 billion to build, and the Patriot missile defense system in Dhahran (1991) that failed to intercept an incoming Scud missile, to name just a few of the more prominent software accidents.

Though building error-free software may be far-fetched, at least within reasonable costs for large-scale software, cost-effective ways to reduce as many errors as possible are always in high demand.

Static analysis, which is the focus of this book, is one kind of program analysis. We conclude this chapter by characterizing static analysis in comparison with other program analysis techniques.

1.3 Concepts in Program Analysis

The remainder of this chapter characterizes static program analysis and compares it with other program analysis techniques. We provide keys to understand how

each program analysis technique operates and to assess their strengths and weaknesses. This characterization will give basic intuitions of the strengths and limitations of static analysis.

1.3.1 What to Analyze

The first question to answer to characterize program analysis techniques is *what programs* they analyze in order to determine *what properties*.

Target Programs An obvious characterization of the target programs to analyze is the programming languages in which the programs are written, but this is not the only one.

- **Domain-specific analyses:** Certain analyses are aimed at specific families of programs. This specialization is a pragmatic way to achieve a cost-effective program analysis, because each family has a particular set of characteristics (such as program idioms) on which a program analysis can focus. For example, consider the C programming language. Though the language is widely used to write software, including operating systems, embedded controllers, and all sorts of utilities, each family of programs has a special character. Embedded software is often safety-critical (thus needs thorough verification) but rarely uses the most complex features of the C language (such as recursion, dynamic memory allocation, and non-local jumps **setjmp/longjmp**), which typically makes analyzing such programs easier than analyzing general applications. Device drivers usually rely on low-level operations that are harder to reason about (e.g., low-level access to sophisticated data structures) but are often of moderate size (a few thousand lines of code).
- **Non-domain-specific analyses:** Some analyses are designed without focus on a particular family of programs of the target language. Such analyses are usually those incorporated inside compilers, interpreters, or general-purpose programming environments. Such analyses collect information (e.g., constants variables, common errors such as buffer overruns) about the input program to help compilers, interpreters, or programmers for an optimized or safe execution of the program. Non-domain-specific analyses risk being less precise and cost-effective than domain-specific ones in order to have an overall acceptable performance for a wide range of programs.

Besides the language and family of programs to consider, the way input programs are handled may also vary and affects how the analysis works. An

obvious option is to handle source programs directly just like a compiler would, but some analyses may input different descriptions of programs instead. We can distinguish two classes of techniques:

- **Program-level analyses** are run on the source code of programs (e.g., written in C or in Java) or on executable program binaries and typically involve a front end similar to a compiler's that constructs the syntax trees of programs from the program source or compiled files.
- **Model-level analyses** consider a different input language that aims at modeling the semantics of programs; then the analyses input not a program in a language such as C or Java but a description that *models* the program to analyze. Such models either need to be constructed manually or are computed by a separate tool. In both cases, the construction of the model may hide either difficulties or sources of inaccuracy that need to be taken precisely into account.

Target Properties A second obvious element of characterization of a program analysis is the set of semantic properties it aims at computing. Among the most important families of target properties, we can cite safety properties, liveness properties, and information flow properties.

- A **safety property** essentially states that a program will never exhibit a behavior observable within finite time. Such behaviors include termination, computing a particular set of values, and reaching some kind of error state (such as integer overflows, buffer overruns, uncaught exceptions, or deadlocks). Hence, a program analysis for some safety properties chases program behaviors that are observable within finite time. Historically this class is called *safety property* because the goal of the analysis is to prove the absence of bad behaviors, and the bad behaviors are mostly those that occur on finite executions.
- A **liveness property** essentially states that a program will never exhibit a behavior observable only after infinite time. Examples of such behaviors include non-termination, live-lock, or starvation.

Hence, program analysis for liveness properties searches for the existence of program behaviors that are observable after infinite time.

- **Information flow properties** define a large class of program properties stating the absence of dependence between pairs of program behaviors. For instance, in the case of a web service, users should not be able to derive the credential of another user from the information they can access. Unlike safety and liveness properties, information flow

properties require reasoning about pairs of executions. More generally, so-called *hyperproperties* define a class of program properties that are characterized over several program executions.

The techniques to reason about these classes of semantic properties are different. As indicated above, safety properties require considering only finite executions, whereas liveness properties require reasoning about infinite executions. As a consequence, the program analysis techniques and algorithms dedicated to each family of semantic properties will differ as well.

1.3.2 Static versus Dynamic

An important characteristic of a program analysis technique is *when* it is performed or, more precisely, whether it operates during or before program execution.

A first solution is to make the analysis at run-time, that is, during the execution of the program. Such an approach is called *dynamic*, as it takes place while the program computes, typically over several executions.

Example 1.1 (User assertions) User assertions provide a classic case of a dynamic approach to checking whether some conditions are satisfied by all program executions. Once the assertions are inserted, the process is purely dynamic: whenever an assertion is executed, its condition is evaluated, and an error is returned if the result is **false**.

Note that some programming languages perform run-time checking of specific properties. For instance, in Java, any array access is preceded by a dynamic bound check, which returns an exception if the index is not valid; this mechanism is equivalent to an assertion and is also dynamic.

A second solution is to make the analysis *before* program execution. We call such an approach a *static* analysis, as it is done once and for all and independently from any execution.

Example 1.2 (Strong typing) Many programming languages require compilers to carry out some type-checking stage, which ensures that certain classes of errors will never occur during the execution of the input program. This is a perfect example of a static analysis since typing takes place independently from any program execution, and the result is known before the program actually runs.

Static and dynamic techniques are radically different and come with distinct sets of advantages and drawbacks. While dynamic approaches are often easier to design and implement, they also often incur a performance cost at run-time, and they do not force developers to fix issues before program execution. On the other hand, after a static analysis is done once, the program can be run as usual, without any slowdown. Also, some properties cannot be checked dynamically. For example, if the property of interest is termination, dynamically detecting a non-terminating execution would require constructing an infinite program run. Dynamic and static analyses have different aftermaths once they detect a property violation. A dynamic analysis upon detecting a property violation can simply

abort the program execution or apply an unobtrusive surgery to the program state and let the execution continue with a risk of having behaviors unspecified in the programs. On the other hand, when a static analysis detects a property violation, developers can still fix the issue before their software is in use.

1.3.3 A Hard Limit: Uncomputability

Given a language of programs to analyze and a property of interest, an ideal program analysis would always compute in a fully automated way the exact result in finite time. For instance, let us consider the certification that a program (e.g., a piece of safety-critical embedded software) will never crash due to a run-time error. Then we would like to use a static program analysis that will always successfully catch any possible run-time error, that will always say when a program is run-time error free, and that will never require any user input.

Unfortunately, this is, in general, impossible.

The Halting Problem Is Not Computable The canonical example of a semantic property for which no exact and fully automatic program analysis can be found is *termination*. Given a programming language, we cannot have a program analysis that, for any program in that language, correctly decides in finite time whether the program will terminate or not.

Indeed, it is well known that the halting problem is *not computable*. We explain more precisely the meaning of this statement. In the following, we consider a *Turing-complete* language, that is, a language that is as expressive as a Turing machine (common general-purpose programming languages all satisfy this condition), and we denote the set of all the programs in this language by L . Second, given a program p in L , we say that an execution e terminates if it reaches the end of p after finitely many computation steps. Last, we say that a program terminates if and only if its executions terminate.

Theorem 1.1 (Halting problem) *The halting problem consists in finding an algorithm halt such that,*

for every program $p \in L$, $\text{halt}(p) = \text{true}$ if and only if p terminates.

The halting problem is not computable: there is no such algorithm halt , as proved simultaneously by Alonso Church [20] and Alan Turing [106] in 1936.

This means that termination is beyond the reach of a fully automatic and precise program analysis.

Interesting Semantic Properties Are Not Computable More generally, any *nontrivial semantic properties* are also not computable. By *semantic property* we mean a property that can be completely defined with respect to the set of executions of a

program (as opposed to a syntactic property, which can be decided directly based on the program text). We call a semantic property nontrivial when there are programs that satisfy it and programs that do not satisfy it. Obviously only such properties are worth the effort of designing a program analysis.

It is easy to see that a particular nontrivial semantic property is uncomputable; that is, the property cannot have an exact decision procedure (analyzer). Otherwise, the exact decision procedure solves the halting problem. For example, consider a property: this program prints 1 and finishes. Suppose there exists an analyzer that correctly decides the property for any input program. This analyzer solves the halting problem as follows. Given an input program P , the analyzer checks its slightly changed version “ P ; `print 1`.” That the analyzer says “yes” means P stops, and “no” means P does not stop.

Indeed, Rice’s theorem settles the case that any nontrivial semantic property is not computable:

Theorem 1.2 (Rice theorem) *Let L be a Turing-complete language, and let P be a nontrivial semantic property of programs of L . There exists no algorithm such that,*

for every program $p \in L$, it returns `true` if and only if p satisfies the semantic property P .

As a consequence, we should also give up hope of finding an ideal program analysis that can determine fully automatically when a program satisfies any interesting property such as the absence of run-time errors, the absence of information flows, and functional correctness.

Toward Computability However, this does not mean that no useful program analysis can be designed. It means only that the analyses we are going to consider will all need to suffer some kind of limitation, by giving up on automation, by targeting only a restricted class of programs (i.e., by giving up the *for every program* in theorem 1.2), or by not always being able to provide an exact answer (i.e., by giving up the *if and only if* in theorem 1.2). We discuss these possible compromises in the next sections.

1.3.4 Automation and Scalability

The first way around the limitation expressed in Rice’s theorem is to give up on *automation* and to let program analyses require some amount of user input. In this case, the user is asked to provide some information to the analysis, such as global or local invariants (an *invariant* is a logical property that can be proved to be inductive for a given program). This means that the analysis is partly manual since users need to compute part of the results themselves.

Obviously, having to supply such information can often become quite cumbersome when programs are large or complex, which is the main drawback of manual methods.

Worse still, this process may be error prone, such that human error may ultimately lead to wrong results. To avoid such mistakes, program analysis tools may independently verify the user-supplied information. Then, when the user-supplied information is wrong, the analysis tool will simply reject it and produce an error message. When the analysis tool can complete the verification and check the validity of the user-supplied information, the correctness of the final result will be guaranteed.

Even when a program analysis is automatic, it may not always produce a result within a reasonable time. Indeed, depending on the complexity of the algorithms, a program analysis tool may not be able to scale to large programs due to time costs or other resource constraints (such as memory usage). Thus, *scalability* is another important characteristic of a program analysis tool.

1.3.5 Approximation: Soundness and Completeness

Instead of giving up on automation, we can relax the conditions about program analysis by letting it sometimes return inaccurate results.

It is important to note that *inaccurate* does not mean wrong. Indeed, if the kind of inaccuracy is known, the user may still draw (possibly partly) conclusive results from the analysis output. For example, suppose we are interested in program termination. Given an input program to verify, the program analysis may answer “yes” or “no” only when it is fully sure about the answer. When the analysis is not sure, it will just return an undetermined result: “don’t know.” Such an analysis would be still useful if the cases where it answers “don’t know” are not too frequent.

In the following paragraphs, we introduce two dual forms of inaccuracies (or, equivalently, approximations) that program analysis may make. To fix the notations, we assume a semantic property of interest P and an analysis tool `analysis`, to determine whether this property holds.

Ideally, if `analysis` were perfectly accurate, it would be such that,

$$\text{for every program } p \in L, \text{analysis}(p) = \text{true} \Leftrightarrow p \text{ satisfies } P.$$

This equivalence property can be decomposed into a pair of implications:

$$\left\{ \begin{array}{ll} \text{For every program } p \in L, & \text{analysis}(p) = \text{true} \implies p \text{ satisfies } P. \\ \text{For every program } p \in L, & \text{analysis}(p) = \text{true} \iff p \text{ satisfies } P. \end{array} \right.$$

Therefore, we can weaken the equivalence by simply dropping either of these two implications. In both cases, we get a partially accurate tool that may return either a conclusive answer or a nonconclusive one (“don’t know”).

We now discuss in detail both of these implications.

Soundness A *sound* program analysis satisfies the first implication.

Definition 1.2 (Soundness) *The program analyzer analysis is sound with respect to property P whenever, for any program p ∈ L, analysis(p) = true implies that p satisfies property P.*

When a sound analysis (or analyzer) claims that the program has property P , it guarantees that the input program indeed satisfies the property. We call such an analysis *sound* as it always errs on the side of caution: it will not claim a program satisfies P unless this property can be guaranteed. In other words, a sound analysis will reject all programs that do not satisfy P .

Example 1.3 (Strong typing) *A classic example is that of strong typing that is used in program languages such as ML, that is based on the principle that “well-typed programs do not go wrong”: indeed, well-typed programs will not present certain classes of errors whereas certain programs that will never crash may still be rejected.*

From a logical point of view, the soundness objective is very easy to meet since the trivial analysis defined to always return **false** obviously satisfies definition 1.2. Indeed, this trivial analysis will simply reject any program. This analysis is not useful since it will never produce a conclusive answer. Therefore, in practice, the design of a sound analysis will try to give a conclusive answer as often as possible. This is in practice possible. As an example, the case of an ML program that cannot be typed (i.e., is rejected) although there exists no execution that crashes due to a typing error is rare in practice.

Completeness A *complete* program analysis satisfies the second, opposite implication:

Definition 1.3 (Completeness) *The program analyzer analysis is complete with respect to property P whenever, for every program p ∈ L, such that p satisfies P, analysis(p) = true.*

A complete program analysis will accept every program that satisfies property P . We call such an analysis *complete* because it does not miss a program that has the property. In other words, when a complete analysis rejects an input program, the completeness guarantees that the program indeed fails to satisfy P .

Example 1.4 (User assertions) *The error search technique based on user assertions is complete in the sense of definition 1.3. User assertions let developers improve the quality of their software thanks to run-time checks inserted as conditions in the source code and that are checked during program executions. This practice can be seen as a very rudimentary form of verification for a limited class of safety properties, where a given condition should never be violated. Faults are reported during program executions, as assertion*

failures. If an assertion fails, this means that at least one execution will produce a state where the assertion condition is violated.

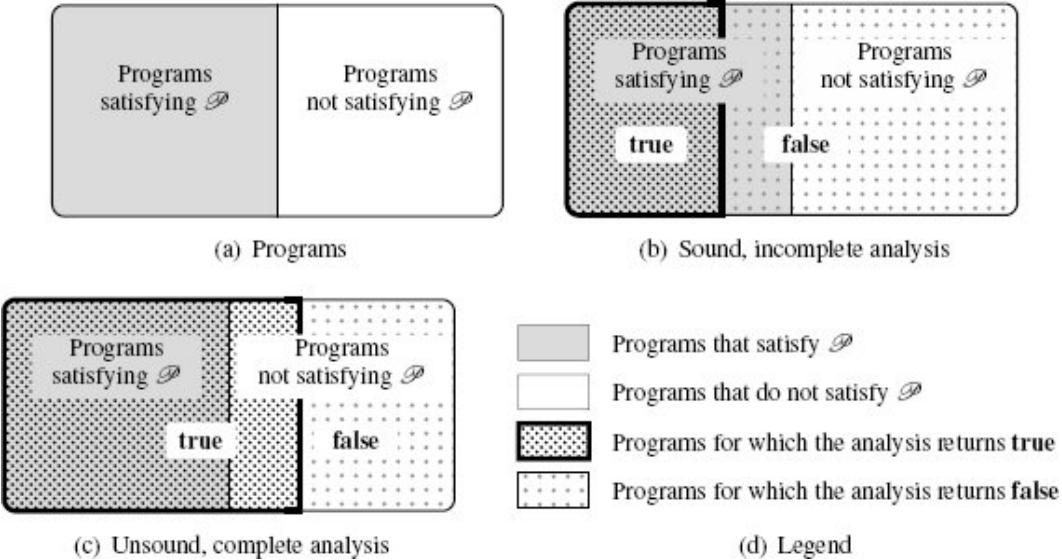


Figure 1.2

Soundness and completeness demonstrated with Venn diagrams

As in the case of soundness, it is very easy to provide a trivial but useless complete analysis. Indeed, if `analysis` always returns **true**, then it never rejects a program that satisfies the property of interest; thus, it is complete, though it is of course of no use. To be useful, a complete analyzer should often reject programs that do not satisfy the property of interest. Building such useful complete analyses is a difficult task in general (just as it is also difficult to build useful sound analyses).

Soundness and Completeness Soundness and completeness are dual properties. To better show them, we represent answers of sound and complete analyses using Venn diagrams in figure 1.2, following the legend in figure 1.2(d):

- Figure 1.2(a) shows the set of all programs and divides it into two subsets: the programs that satisfy the semantic property P and the programs that do not satisfy P . A sound and complete analysis would always return **true** exactly for the programs that are in the left part of the diagram.
- Figure 1.2(b) depicts the answers of an analysis that is *sound* but *incomplete*: it rejects all programs that do not satisfy the property but

also rejects some that do satisfy it; whenever it returns **true**, we have the guarantee that the analyzed program satisfies P .

- Figure 1.2(c) depicts the answers of an analysis that is *complete* but *unsound*: it accepts all programs that do satisfy the property but also accepts some that do not satisfy it; whenever it returns **false**, we have the guarantee that the analyzed program does not satisfy P .

Due to the computability barrier, we should not hope for a sound, complete, and fully automatic analysis when trying to determine which programs satisfy any nontrivial execution property for a Turing-complete language. In other words, when a program analysis is automatic, it is either unsound or incomplete. However, this does not mean it is impossible to design a program analysis that returns very accurate (sound and complete) results on a specific set of input programs. But even in that case, there will always exist input programs for which the analysis will return inaccurate (unsound or incomplete) results.

In the previous paragraphs, we have implicitly assumed that the program analysis tool **analysis** always terminates and never crashes. In general, non-termination or crashes of **analysis** should be interpreted conservatively. For instance, if **analysis** is meant to be sound, then its answer should be conservatively considered negative (**false**) whenever it does not return **true** within the allocated time bounds.

1.4 Families of Program Analysis Techniques

In this section, we describe several families of approaches to program analysis. Due to the negative result presented in section 1.3.3, no technique can achieve a fully automatic, sound, and complete computation of a nontrivial property of programs. We show the characteristics of each of these techniques using the definitions of section 1.3.

1.4.1 Testing: Checking a Set of Finite Executions

When trying to understand how a system behaves, often the first idea that comes to mind is to observe the executions of this system. In the case of a program that may not terminate and may have infinitely many executions, it is of course not feasible to fully observe all executions.

Therefore, the *testing* approach observes only a finite set of finite program executions. This technique is used by all programmers, from beginners to large teams designing complex computer systems. In industry, many levels of testing are performed at all stages of development, such as unit testing (execution of

sample runs on a basic function) and integration testing (execution of large series of tests on a completed system, including hardware and software).

Basic testing approaches, such as random testing [11], typically provide a low coverage of the tested code. However, more advanced techniques improve coverage. As an example, *concolic testing* [47] combines testing with symbolic execution (computation of exact relations between input and output variables on a single control flow path) so as to improve coverage and accuracy.

Testing has the following characteristics:

- It is in general easy to automate, and many techniques (such as concolic testing) have been developed to synthesize useful sets of input data to maximize various measures of coverage.
- In almost all cases, it is unsound, except in the cases of programs that have only a finite number of finite executions (though it is usually prohibitively costly in that case).
- It is complete since a failed testing run will produce an execution that is incorrect with respect to the property of interest (such a counter-example is very useful in practice since it shows programmers exactly how the property of interest may be violated and often gives precise information on how to fix the program).

Besides, testing is often costly, and it is hard to achieve a very high path coverage on very large programs. On the other hand, a great advantage of testing is that it can be applied to a program in the conditions in which it is supposed to be run; for instance, testing a program on the target hardware, with the target operating system and drivers, may help diagnose issues that are specific to this combination.

When the semantics of programs is non-deterministic, it may not be feasible to reproduce an execution, which makes the exploitation of the results produced by testing problematic. As an example, the execution of a set of concurrent tasks depends on the scheduling strategy so that two runs with the same input may produce different results, if this strategy is not fully deterministic.

Another consideration is that testing will not allow attacking certain classes of properties. For instance, it will not allow proving that a program terminates, even over a finite set of inputs.

1.4.2 Assisted Proof: Relying on User-Supplied Invariants

A second way to avoid the limitation shown in section 1.3.3 consists in giving up on automation.

This is essentially the approach followed by *machine-assisted* techniques. This means that users may be required to supply additional information together with the program to analyze. In most cases, the information that needs to be supplied consists of loop invariants and possibly some other intermediate invariants. This often requires some level of expertise. On the other hand, a large part of the verification can generally still be carried out in a fully automatic way.

We can cite several kinds of program analyses based on machine-assisted techniques. A first approach is based on theorem-proving tools like Coq [24], Isabelle/HOL [49], and PVS [88] and requires the user to formalize the semantics of programs and the properties of interest and to write down proof scripts, which are then checked by the prover. This approach is adapted to the proof of sophisticated program properties. It was applied to the verified CompCert compiler [77] from C to Power-PC assembly (the compiler is verified in the sense that it comes with a proof that it will compile any valid C program properly). It was also used for the design of the microkernel seL4 verified [69]. A second approach leverages a tool infrastructure to prove a specific set of properties over programs in a specific language. The B-method [1] tool set implements such an approach. Also, tools such as the Why C program verification framework [43] or Dafny [76] input a program with a property to verify and attempt to prove the property using automatic decision procedures, while relying on the user for the main program invariants (such as loop invariants) and when the automatic procedures fail.

Machine-assisted techniques have the following characteristics:

- They are not fully automatic and often require the most tedious logical arguments to come from the human user.
- In practice, they are sound with respect to the model of the program semantics used for the proof, and they are also complete up to the abilities of the proof assistant to verify proofs (the expressiveness of the logics of the proof assistant may prevent some programs to be proved, though this is rarely a problem in practice).

In practice, the main limitation of machine-assisted techniques is the significant resources they require, in terms of time and expertise.

1.4.3 Model Checking: Exhaustive Exploration of Finite Systems

Another approach focuses on finite systems, that is, systems whose behaviors can be exhaustively enumerated, so as to determine whether all executions satisfy the property of interest. This approach is called *finite-state model checking* [38, 93, 21] since it will check a model of a program using some kind of exhaustive

enumeration. In practice, model-checking tools use efficient data structures to represent program behaviors and avoid enumerating all executions thanks to strategies that reduce the search space.

Note that this solution is very different from the testing approach discussed in section 1.4.1. Indeed, testing samples a finite set of behaviors among a generally infinite set, whereas model checking attempts to check all executions of a finite system.

The finite model-checking approach has been used both in hardware verification and in software verification.

Model checking has the following characteristics:

- It is automatic.
- It is sound and complete *with respect to the model*.

An important caveat is that the verification is performed at the model level and not at the program level. As a first consequence, this means that a model of the program needs to be constructed, either manually or by some automatic means. In practice, most model-checking tools provide a front end for that purpose. A second consequence is that the relation between this model and the input program should be taken into account when assessing the results; indeed, if the model cannot capture exactly the behaviors of the program (which is likely as programs are usually infinite systems since executions may be of arbitrary length), the checking of the synthesized model may be either incomplete or unsound, with respect to the input program. Some model-checking techniques are able to automatically refine the model when they realize that they fail to prove a property due to a spurious counter-example; however, the iterations of the model checking and refinement may continue indefinitely, so some kind of mechanism is required to guarantee termination. In practice, model-checking tools are often conservative and are thus sound and incomplete with respect to the input program. A large number of model-checking tools have been developed for verifying different kinds of logical assertions on various models or programming languages. As an example, UPPAAL [8] verifies temporal logic formulas on timed automata.

1.4.4 Conservative Static Analysis: Automatic, Sound, and Incomplete Approach

Instead of constructing a finite model of programs, *static analysis* relies on other techniques to compute conservative descriptions of program behaviors using finite resources. The core idea is to finitely over-approximate the set of all program behaviors using a specific set of properties, the computation of which can be automated [26, 27]. A (very simple) example is the type inference present

in many modern programming languages such as variants of ML. Types [50, 81] provide a coarse view of what a function does but do so in a very effective manner, since the correctness of type systems guarantees that a function of type `int -> bool` will always input an integer and return a Boolean (when it terminates). Another contrived example is the removal of array bound checks by some compilers for optimization purposes, using numerical properties over program variables that are automatically inferred at compile-time. The next chapters generalize this intuition and introduce many other forms of static analyses.

Besides compilers, static analysis has been very heavily used to design program verifiers and program understanding tools for all sorts of programming languages. Among many others, we can cite the ASTRÉE [12] static analyzer for proving the absence of run-time errors in embedded C codes, the Facebook INFER [15] static analyzer for the detection of memory issues in C/C++/Java programs, the JULIA [103] static analyzer for discovering security issues in Java programs, the POLYSPACE [34] static analyzer for ADA/C/C++ programs, and the SPARROW [66] static analyzer for the detection of memory errors in C programs.

Static analysis approaches have the following characteristics:

- They are automatic.
- They produce sound results, as they compute a *conservative* description of program behaviors, using a limited set of logical properties. Thus, they will never claim the analyzed program satisfies the property of interest when it is not true.
- They are generally incomplete because they cannot represent all program properties and rely on algorithms that enforce termination of the analysis even when the input program may have infinite executions. As a consequence, they may fail to prove correct some programs that satisfy the property of interest.

Static analysis tools generally input the source code of programs and do not require modeling the source code using an external tool. Instead, they directly compute properties taken in a fixed set of logical formulas, using algorithms that we present throughout the following chapters.

While a static analysis is incomplete in general, it is often possible to design a sound static analysis that gives the best possible answer on classes of interesting input programs, as discussed in section 1.3.5. However, it is then always possible to craft a correct input program for which the analysis will fail to return a conclusive result.

Last, we remark that it is entirely possible to drop soundness so as to preserve automation and completeness. This leads to a different kind of analysis that produces an under-approximation of the program’s actual behaviors and answers a very different kind of question. Indeed, such an approach may guarantee that a given subset of the executions of the program can be observed. For instance, the approach may be useful to establish that this program has at least one successful execution. On the other hand, it does not prove properties such as the absence of run-time errors.

1.4.5 Bug Finding: Error Search, Automatic, Unsound, Incomplete, Based on Heuristics

Some automatic program analysis tools sacrifice not only completeness but also soundness. The main motivation to do so is to simplify the design and implementation of analysis tools and to provide lighter-weight verification algorithms. The techniques used in such tools are often similar to those used in model checking or static analysis, but they relax the soundness objective. For instance, they may construct unsound finite models of programs so as to quickly enumerate a subset of the executions of the analyzed program, such as by considering only what happens in the first iteration of each loop [110], whereas a sound tool would have to consider possibly unbounded iteration numbers. As an example, the commercial tool COVERITY [10] applies such techniques to programs written in a wide range of languages (e.g., Java, C/C++, JavaScript, or Python). Similarly, the tool CODESONAR [79] relies on such approaches so as to search for defects in C/C++ or Assembly programs. The CBMC tool (C Bounded Model Checker) [70] extracts models from C/C++ or Java programs and performs bounded model checking on them, which means that it explores models only up to a fixed depth. It is thus a case that a model checker gives up on soundness in order to produce fewer alarms.

Since the main motivation of this approach is to discover bugs (and not to prove their absence), it is often referred to as *bug finding*. Such tools are usually applied to improve the quality of noncritical programs at a low cost.

Bug-finding tools have the following characteristics:

- They are automatic.
- They are neither sound nor complete; instead, they aim at discovering bugs rather quickly, so as to help developers.

| | Auto-matic | Sound | Complete | Object | When |
|--------------------------------------|------------|-------|----------|--------------|---------|
| Testing | Yes | No | Yes | Program | Dynamic |
| Assisted proving | No | Yes | Yes/No | Model | Static |
| Model checking of finite-state model | Yes | Yes | Yes | Finite model | Static |
| Model checking at program level | Yes | Yes | No | Program | Static |
| Conservative static analysis | Yes | Yes | No | Program | Static |
| Bug finding | Yes | No | No | Program | Static |

Figure 1.3

An overview of program analysis techniques

1.4.6 Summary

Figure 1.3 summarizes the techniques for program analysis introduced in this chapter and compares them based on five criteria. As this comparsion shows, due to the computability barrier, no technique can provide fully automatic, sound, and complete analyses. Testing sacrifices soundness. Assisted proving is not automatic (even if it is often partly automated, the main proof arguments generally need to be human provided). Model-checking approaches can achieve soundness and completeness only with respect to finite models, and they generally give up completeness when considering programs (the incompleteness is often introduced in the modeling stage). Static analysis gives up completeness (though it may be designed to be precise for large classes of interested programs). Last, bug finding is neither sound nor complete.

As we remarked earlier, another important dimension is *scalability*. In practice, all approaches have limitations regarding scalability, although these limitations vary depending on the intended applications (e.g., input programs, target properties, and algorithms used).

1.5 Roadmap

From now on, we focus on *conservative static analysis*, from its design methodologies to its implementation techniques.

Definition 1.4 (Static analysis) Static analysis is an automatic technique for program-level analysis that approximates in a conservative manner semantic properties of programs before their execution.

After a gentle introduction to static analysis in chapter 2, we present a static analysis framework based on a compositional semantics in chapter 3, a static analysis framework based on a transitional semantics in chapter 4, and some advanced techniques in chapter 5. These frameworks, thanks to a semantics-based viewpoint, are general so that they can guide the design of conservative static analyses for any programming language and for any semantic property. In chapter 6, we present issues and techniques regarding the use of static analysis in practice. Chapter 7 discusses and demonstrates the implementation techniques to build a static analysis tool. In chapter 8, we present how we use the general static analysis framework to analyze seemingly complex features of realistic programming languages. Chapter 9 discusses several important families of semantic properties of interest and shows how to cope with them using static analysis. In chapter 10, we present several specialized yet high-level frameworks for specific target languages and semantic properties. Finally, in chapter 11 we summarize this book.

2 A Gentle Introduction to Static Analysis

Goal of This Chapter In this chapter, we provide an introduction to static analysis that does not require any background. This introduction aims at making the core concepts of static analysis intuitive and crisp. To this end, we define a basic programming language that describes sequences of transformations applied to points in a two-dimensional space.¹ The notions presented here extend to realistic programming languages. We formalize them thoroughly in chapter 3 in the case of a basic imperative language.

Recommended Reading: [S], [D], [U] We recommend this chapter to all readers, as it introduces the basic intuitions useful to understand many more advanced static analysis techniques and the way static analysis tools work. Understanding these concepts is important not only to design or implement a static analyzer but also to use it as well as possible.

2.1 Semantics and Analysis Goal: A Reachability Problem

Syntax of a Very Basic Language For the sake of making our first introduction to static analysis intuitive, we consider a very basic language with intuitive notions of states and executions. The language under study is inspired by drawing languages used for educational purposes, such as introducing children to programming.

A *state* describes the configuration of a computer running a program, observed at a given instant. In general, this includes a description of the memory contents, the registers, and the program counter. In this chapter, a state will simply denote a point in the two-dimensional space, described by its real coordinates (x,y) . We denote the set of such states by S .

We let programs define combinations of basic geometric operations. Basic operations comprise:

- initialization with a point that is non-deterministically chosen in a fixed region \mathfrak{R} (e.g., the $[0,1] \times [0,1]$ square or any other geometrical shape specified by a set of points);
- geometrical translations (specified by a vector); and
- geometrical rotations (specified by a center and an angle in degrees).

Moreover, a program is defined as a sequence of operations, or as a non-deterministic choice of two sequences of operations, or as a non-deterministic iteration of a sequence of operations (the number of iterations is chosen non-deterministically). To avoid starting from an undefined state, we assume that a program always begins with an initialization, which fixes the set of initial states.

The syntax of programs is defined by the grammar below (note that we only consider programs that start with an initialization statement, even though this grammar does not express that constraint):

| | |
|----------------------------------|---|
| $::=\mathbf{init}(\mathfrak{R})$ | initialization, with a state in \mathfrak{R} |
| translation (u,v) | translation by vector (u,v) |
| rotation (u,v,θ) | rotation defined by center (u,v) and angle θ |
| p p ; p | sequence of operations |
| {p}or{p} | choice (the branch taken is non-deterministic) |
| iter {p} | iteration (the number of iterations is non-deterministic) |

Semantics As observed in chapter 1, static analysis aims at computing semantic properties of programs. Therefore, before we look into the definition of a static analysis, we need to define the *semantics* of programs, which should characterize the program executions. A common way to achieve this is to simply let the semantics be the set of all the program executions. Such a semantics is often called *collecting semantics*.

An *execution* provides a complete view of a single run of the program. Since we assume that a program makes discrete computation steps at every clock tick, it is naturally described by a sequence of states. Each of the basic program constructions in the above grammar is quite simple, so we do not formalize them fully. Intuitively, their semantics is defined as follows:

- The initialization operation simply produces a state in a given region.
- Translation and rotation transformations induce basic execution steps, which perform the corresponding geometric transformations.
- Sequences of operations yield sequences of execution steps.
- Non-deterministic choices and iterations, respectively, select or repeat a block of code and construct executions that can be derived from those of the subprograms.

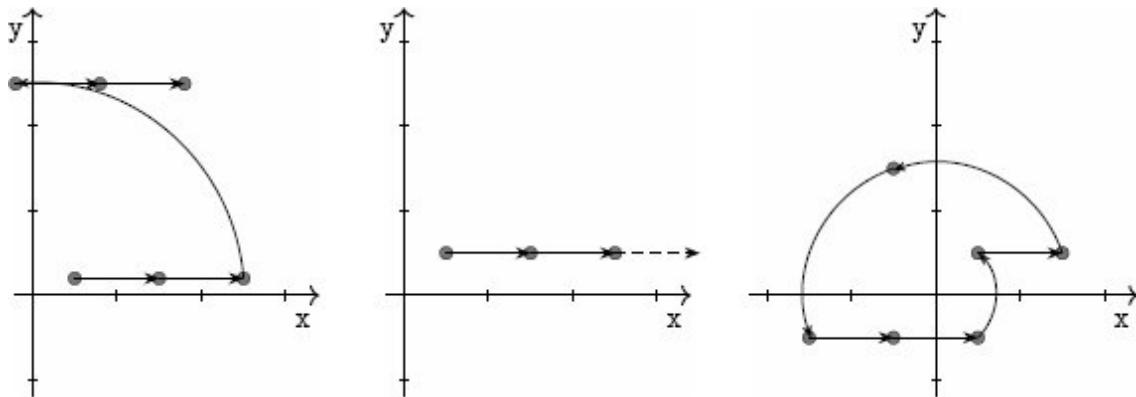


Figure 2.1

A few program executions

Example 2.1 (Semantics) To make this semantics more intuitive, we consider the following program:

```

init([0,1] × [0,1]);
translation(1,0);
iter{
{
    translation(1,0)
}or{
    rotation(0,0,90°)
}
}

```

This program starts in the $[0,1] \times [0,1]$ square, performs a translation, and then performs a number of translations or rotations that are chosen non-deterministically (i.e., an oracle decides at run-time both the number of operations and their nature). Figure 2.1 shows three executions:

- In the first execution (left), the program starts from (0.5,0.2); performs two translations, one rotation, and two translations; and then terminates.
- In the second execution (middle), the program starts at point (0.5,0.5) and repeats the same transition forever.
- In the third execution (right), the program starts at point (0.5,0.5) and then repeats forever the sequence made of one translation, two rotations, two translations, and one rotation.

While the first execution is finite, the other two are actually infinite (which means that the program runs forever).

Semantic Property of Interest: Reachability It is now time to set the property of interest that we are going to consider in this chapter.

We aim for a *reachability* property that is specified by a zone made of points that should not be reached by any execution of the program. Intuitively, we assume that a set of points is fixed and defines a zone that we expect program executions to *never* reach. In other words, if any execution reaches this zone, we would consider it an error. In the following, we search for a static analysis that is able to catch and reject any program with such an erroneous execution. When a program has no such offending execution, the analysis should, as often as possible, accept the program and issue a proof of correctness.

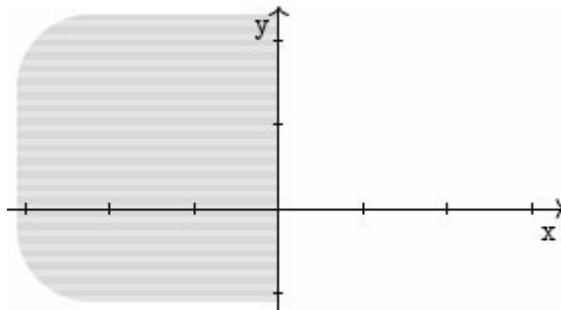


Figure 2.2

Region supposed to be unreachable: points with a negative x-coordinate

This semantic property is a classic example of safety property (section 1.3.1) (although not all safety properties are of that form). Very often, programmers would like to ensure similar properties in real programs. As an example, reaching a state where a C program will dereference a null pointer

will produce an abrupt run-time error. Similarly, if a C program reaches a state where it writes over a dangling pointer, either the execution will fail abruptly or some data will be corrupted. For these reasons, C programmers are usually interested in checking that their programs will never reach a state where they would dereference an invalid, null, or dangling pointer. Thus, the reachability property that we study here is actually quite realistic, even though we are looking at a contrived language.

In this chapter, we often use the region \mathfrak{D} defined by $\mathfrak{D} = \{(x,y) \mid x < 0\}$ to denote the set of states that program executions should never reach, although we will construct a program analysis technique that would work for other regions as well. This “error zone” is depicted in figure 2.2. We let $\neg\mathfrak{D}$ denote the property that we would like to verify since it expresses that all the states a program may reach are *not* in \mathfrak{D} . To give more intuition, we study a couple of programs.

Example 2.2 (Reachability and incorrect executions) First, we consider the program of example 2.1. Obviously, it violates the property $\neg \text{D}$ since figure 2.1 shows two executions that eventually reach a point (x,y) , where $x < 0$. As an example, figure 2.3(a) displays an execution of the program studied in example 2.1, which is incorrect because it reaches the error zone after three steps.

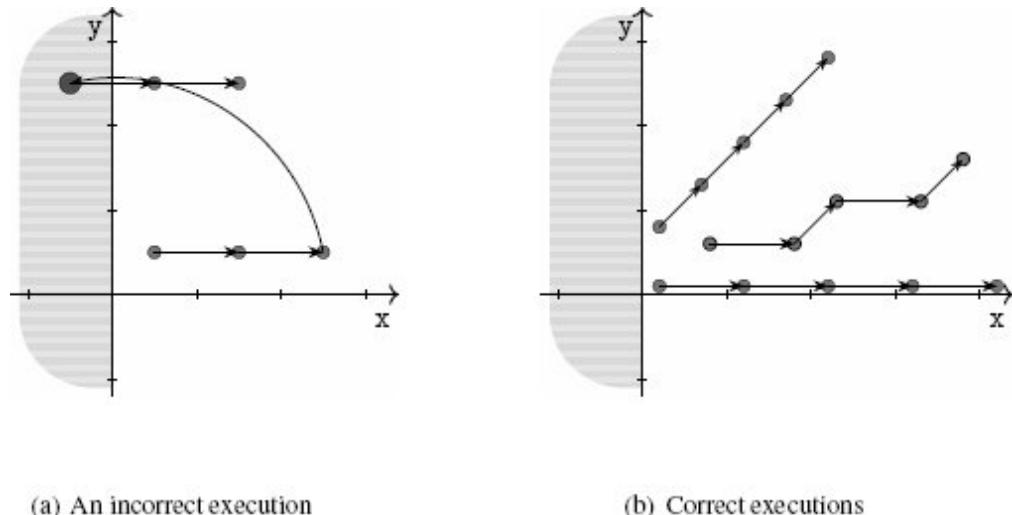


Figure 2.3

Reachability and programs

Example 2.3 (Reachability and program with only correct executions) In this example, we study a second program:

```

init([0, 1] × [0, 1]);
iter{
{
    translation(1, 0);
} or{
    translation(0.5, 0.5);
}
}

```

Figure 2.3(b) displays a few executions of this program, and we observe they are all correct, in the sense that they never enter the error zone \mathcal{D} . In fact, we can informally show that all executions of this program will stay in the safe zone $\neg\mathcal{D}$ at all times:

- They start at a point (x, y) such that $0 \leq x \leq 1$, which thus satisfies $\neg\mathcal{D}$.
- During a loop iteration, x is increased by either 1 or 0.5 depending on the result of a non-deterministic choice; thus, the coordinate x remains non-negative.

Static Analysis for Reachability In the rest of this chapter, we define a static analysis (actually, a family of static analyses) that attempts to determine whether an input program satisfies the semantic property $\neg\mathcal{D}$. The analysis should always return a sound result: if it returns **true** when applied to an input program p , we expect to have the guarantee that no execution of p will ever reach \mathcal{D} . Therefore, a program such as that of example 2.1 will be flagged as “possibly violating the property of interest.” Ideally, we would also like the analysis to be precise enough so that it can conclusively report that the program of example 2.3 is correct.

An obvious way to do this would be to enumerate all executions of the input program so as to determine all reachable configurations. But this would not be feasible, as even simple programs (such as those presented in example 2.1 and example 2.3) have infinitely many executions since the set of initial states is infinite, the length of executions is infinite, and the set of possible series of non-deterministic choices is infinite.

Therefore, we will seek other ways to determine the set of all reachable configurations.

2.2 Abstraction

Core Principle of Abstraction In this section, we search for a way to reason about program executions that will produce a superset of the reachable states and that should be rather simple to compute.

To choose this superset, we first try to draw some intuition from the program studied in example 2.3. Notice that in that example, no execution reaches the region \mathfrak{D} , and we gave an informal proof of this fact:

- In the beginning, the x -coordinate is non-negative.
- At each step it may only grow, which means that, if it is non-negative, it will remain so.

In the following, we aim at making such reasoning steps automatic. We remark that the steps of this proof do not use all the information present in program states:

- The value of the y -coordinate is completely ignored.
- Only the sign of the x -coordinate is considered in the proof (or, more precisely, the fact that the value of x may be either positive, zero, or negative).

This intuition forms the basis of *abstraction* [26]: by retaining only rather coarse information about program states and considering how the program runs, we can still infer interesting information about the set of all program executions (in this case, that $x \geq 0$ at all times). Such information is captured by a set of logical properties that the analysis may manipulate, using algorithms described in the next section. In example 2.3, the only logical properties that seem to matter in the proof are $x \geq 0$ (used in the case of the program of example 2.3) and the property **true**, which is satisfied by any state (that is needed to describe the states the program of example 2.1 may reach).

Of course, many possible sets of logical properties could be used in such proofs. Thus, we need to make clear what logical properties the analysis may manipulate.

Definition 2.1 (Abstraction) We call abstraction a set A of logical properties of program states, which are called abstract properties or abstract elements. A set of abstract properties is called an abstract domain.

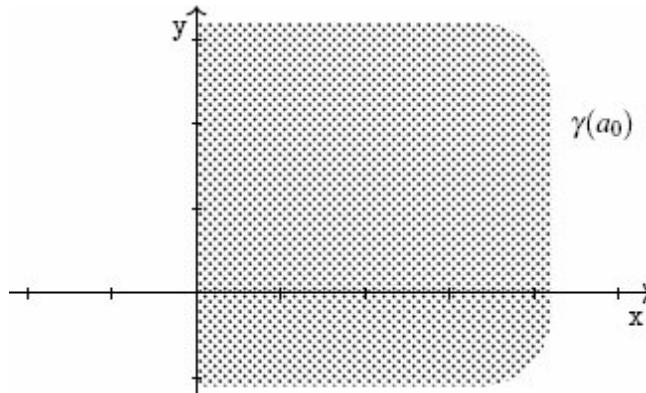


Figure 2.4

Abstraction based on the sign of the x component

In this definition, the word *abstract* is used here as opposed to the word *concrete*; in the following, we use the “concrete” qualifier to denote actual program behaviors, whereas the “abstract” qualifier applies to the properties used in the (automatic) proofs. As an example, the *concrete semantics* is the actual semantics of programs as defined in section 2.1. By contrast, an *abstract semantics* shall define a computable over-approximation of the concrete semantics expressed in terms of abstract states. Actually, the goal of static analysis is precisely to compute such a sound abstract semantics.

The mathematical and computer representation of abstract elements is crucial for the definition of all the static analysis algorithms that we are going to consider. Ideally, the abstract properties should come with an efficient computer representation and with analysis algorithms (the algorithms are discussed further in this chapter), since we intend to develop a static analyzer that relies on these predicates. Thus, it is important to distinguish the abstract elements from their meaning.

Definition 2.2 (Concretization) Given an abstract element a of A , we call *concretization* the set of program states that satisfy it. We denote it by $\gamma(a)$.

Example 2.4 (Abstraction by the sign of the x-coordinate) As a first example, we present the abstraction used above in order to informally demonstrate that the program of example 2.3 never reaches the region \mathfrak{D} . This abstraction has two elements a_0, a_1 , where

- a_0 denotes all the states (x, y) such that $x \geq 0$ ($\gamma(a_0)$ is the infinite half-plane area that is filled with dots in figure 2.4).

- a_1 denotes the set of all the states such that $y(a_1) = S$ (the whole two-dimensional space).

In figure 2.4 (and subsequent figures that depict abstract elements), we represent the points described by an abstract element as a zone filled with dots. This region is syntactically different from the abstract element itself: the latter is the representation of the former, and the analysis manipulates only the representation. Even though we often focus less on this distinction in this chapter (and sometimes implicitly assimilate abstract elements and the regions they denote), it will play a great role in subsequent chapters.

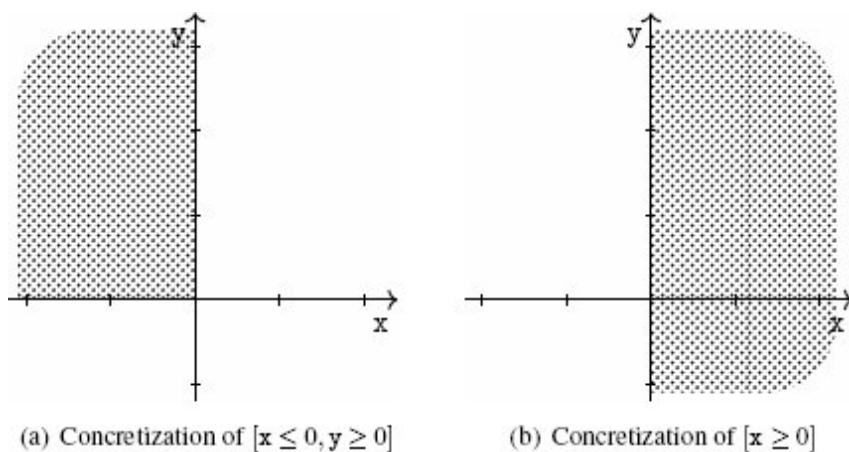


Figure 2.5

Signs abstraction

Of course many possible choices of abstractions are less contrived than the one of example 2.4. Some abstractions describe more expressive sets of logical properties than others. Furthermore, some abstractions yield simpler computer representations and less costly algorithms than others. In the following paragraphs, we present a few other examples of abstractions that also have a simple and intuitive graphical representation.

Signs Abstraction The abstraction of example 2.4 treats x and y differently and is very specific to the property \mathfrak{D} defined in section 2.1. It would not work if we wanted a static analysis to prove that y never becomes negative. Similarly, it would not apply if the property to prove was that x does not become positive. However, this abstraction generalizes into a more

expressive one, which describes a set of states using two pieces of information: the possible values of the sign of x and the possible values of the sign of y . For each variable, this abstraction records whether it may be positive, negative, non-negative, and so forth. The concretizations of a few abstract elements are shown in figure 2.5:

- The left diagram shows the concretization of the abstract element that expresses the fact that x is negative, and y is non-negative.
- The right diagram shows the concretization of the abstract element that expresses the fact that x is positive and this abstract element carries no information about y .

We can observe that this signs abstract domain can express any property the previous domain could express, but it can also describe some properties that were beyond the reach of the previous domain.

Intervals Abstraction In practice, abstractions based on signs are often too weak to capture strong program properties, but other more precise abstractions have been proposed.

Using inequalities and range constraints over variables is a very natural approach to reason over numerical properties. Similarly, we can use range constraints over program variables so as to more precisely describe what values they may take. This is the principle of *intervals abstraction* [26]:

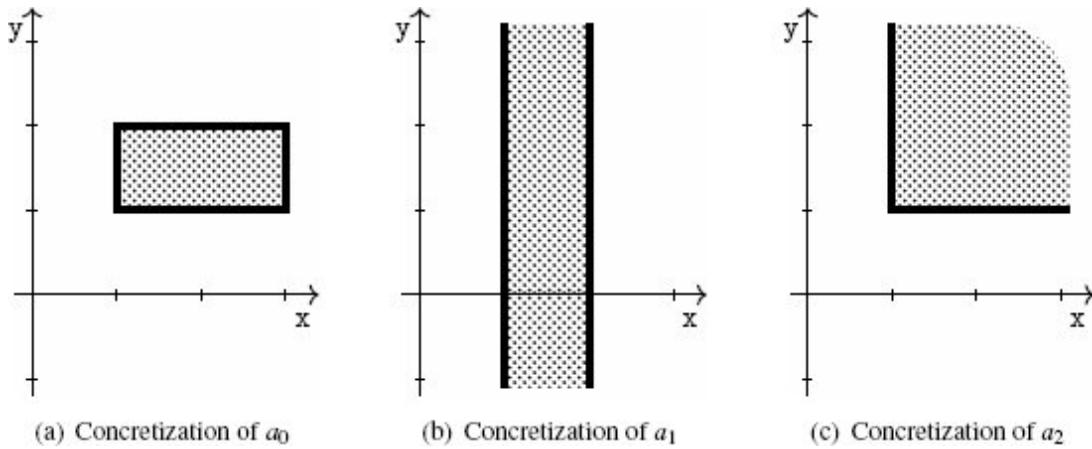


Figure 2.6

Intervals abstraction

Definition 2.3 (Intervals abstraction) *The abstract elements of the interval abstraction are defined by constraints of the form $l_x \leq x$, $x \leq h_x$, $l_y \leq y$, and $y \leq h_y$.*

An abstract element is thus composed of at most four finite bound constraints. We remark that such an abstract element may denote the empty set of points, since some sets of constraints cannot be satisfied, such as $1 \leq x$, $x \leq 0$. We do not write down infinite bound constraints (cases where no lower and/or upper bound is given for a given variable).

Intuitively, interval abstract domain elements correspond to rectangles in the two-dimensional space, the sides of which are parallel to the axes.

Example 2.5 (Intervals abstraction) *The following three abstract elements illustrate the kind of constraints that can be expressed by the intervals abstract domain, together with their concretizations, shown in figure 2.6:*

- a_0 corresponds to numerical constraints $1 \leq x \leq 3$ and $1 \leq y \leq 2$ (the concretization of a_0 is shown in figure 2.6(a));
- a_1 corresponds to numerical constraints $1 \leq x \leq 2$ (the concretization of a_1 is shown in figure 2.6(b));
- a_2 corresponds to numerical constraints $1 \leq x$ and $1 \leq y$ (the concretization of a_2 is shown in figure 2.6(c)).

Obviously, the intervals abstract domain is more expressive than the signs abstract domain. Indeed, any abstract element of the signs abstract domain also corresponds to an element in the intervals abstract domain.

The representation of an abstract element of the intervals domain boils down to at most two numerical constants per variable. Thus, this domain over-approximates a set of points in the two-dimensional space with at most four numerical constants, which take very little space in memory during the analysis.

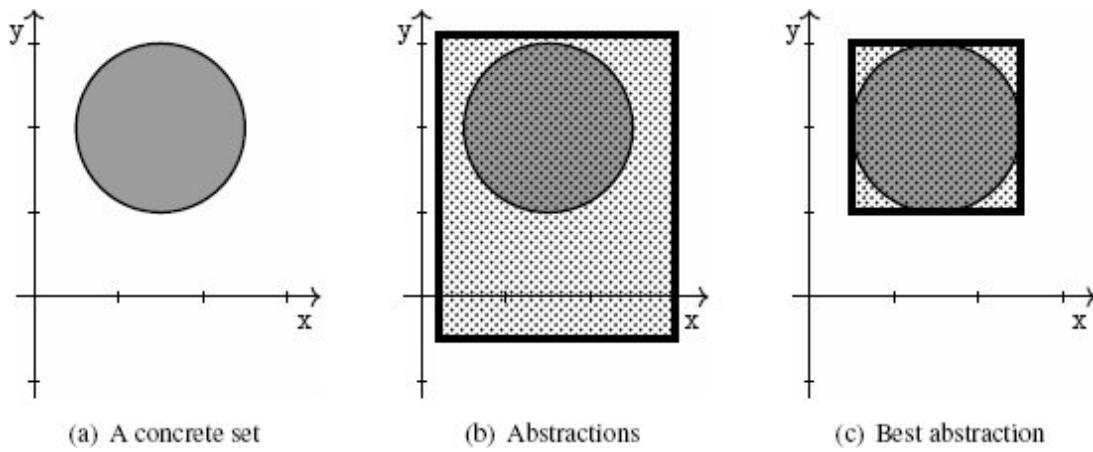


Figure 2.7

Best abstraction

We can introduce at this stage the concept of *best abstraction*. Given any set of points (which correspond to program states), we would like to define an abstract element in the intervals abstract domain that over-approximates our initial set. For instance, let us consider the set of program states defined by the disk shown in figure 2.7(a). Then any box that encloses the disk is a valid over-approximation of this set; indeed, any such box describes all the points in the disk (and more), so it provides a conservative approximation of the disk. However, there exist many such enclosing boxes. Yet, some of these abstractions are more desirable than others. As we mentioned earlier, the goal of abstraction is to account for the concrete set of points using a simple description, at the cost of adding some additional points that are not in the concrete set. Adding fewer points that are not in the concrete set is better since it means the abstraction characterizes the set of points in a less ambiguous and more informative way. In the case of the intervals abstract domain, we can actually solve this problem in an elegant manner; indeed,

the *smallest* rectangle that encloses any non-empty set of points is well defined, using the greatest lower bounds and least upper bounds over both coordinates (the case of the empty set of points is trivial, as the empty rectangle is also an element of the abstract domain). In particular, figure 2.7(c) shows the best approximation of the disk.

More generally, the best abstraction [26] is defined as a function that interprets any set of concrete points into an *optimal* abstract element.

Definition 2.4 (Best abstraction) *We say that a is the best abstraction of the concrete set S if and only if $S \subseteq \gamma(a)$ and for any a' that is an abstraction of S (i.e., $S \subseteq \gamma(a')$), then a' is a coarser abstraction than a . If S has a best abstraction, then the best abstraction is unique. When it is defined, we let α denote the function that maps any concrete set of states into the best abstraction of that set of states.*

As observed above, the intervals abstract domain has a best abstraction function. While computing a precise abstraction (if possible the best abstraction) is preferable in general, we will often encounter useful analyses that cannot compute the best abstraction, or such that the best abstraction cannot even be defined in the sense of definition 2.4. The impossibility to define or compute the best abstraction is in no way a serious flaw for the analysis, as it will only cause it to err on the side of caution (i.e., to return conservative but sound results). Using an over-approximating abstract element is fine, though we prefer to compute the most tightly encompassing one, if it exists.

Finally, we remark that interval constraints cannot capture in a precise manner any complex numerical constraint over both x and y . For instance, it cannot express in an exact manner the property that x is smaller than y . We thus call it a *non-relational abstraction*. Intuitively, an abstract state characterizes each variable by an interval independently from the other variables. On one hand, this simplifies the shape of abstract elements and their representation; on the other hand, it limits the expressiveness of the abstraction.

Convex Polyhedra Abstraction The obvious way to overcome the limitation inherent in the non-relational abstraction is to extend the abstract domain with relational constraints. Augmenting the abstract domain with all linear constraints allows this to be achieved.

Definition 2.5 (Convex polyhedra abstraction) *The abstract elements of the convex polyhedra abstract domain [30] are conjunctions of linear inequality constraints.*

This abstract domain can describe precisely any concrete set that can be described by the signs and intervals abstract domains. It can also describe many other sets of concrete points in a much more precise way than the previous abstractions.

Example 2.6 (Convex polyhedra abstraction) *Figure 2.8 displays the concretization of three convex polyhedra a_0 , a_1 , and a_2 :*

- a_0 describes the conjunction of the three linear constraints:

$$\begin{array}{rcl} x - y & \geq & -0.5 \\ x & \leq & 2.5 \\ x + 4y & \geq & 4.5 \end{array}$$

- a_1 consists of the conjunction of six linear constraints and is of bounded size (we do not list the constraint representation as it would be more involved);
- a_2 consists of the conjunction of four linear constraints and describes an unbounded zone (its concretization describes points where x, y may be arbitrarily large).

There exist several representations for the abstract elements of the convex polyhedra abstract domain. We have already mentioned the representation based on a conjunction of linear inequalities. Since we also noticed that their concretization corresponds exactly to convex polyhedra (hence the name of the abstraction), the abstract elements also have a geometrical representation, based on their sets of vertexes and edges. Actual static analysis algorithms based on convex polyhedra exploit both representations. However, these representations are significantly more complex and costly than for the previous abstract domains: while signs required only a couple of bits per variable, and intervals required at most only two bounds per variable, defining a convex polyhedron typically involves a large number of coefficients or vertexes and edges (in theory there exists no upper bound on the number of constraints).

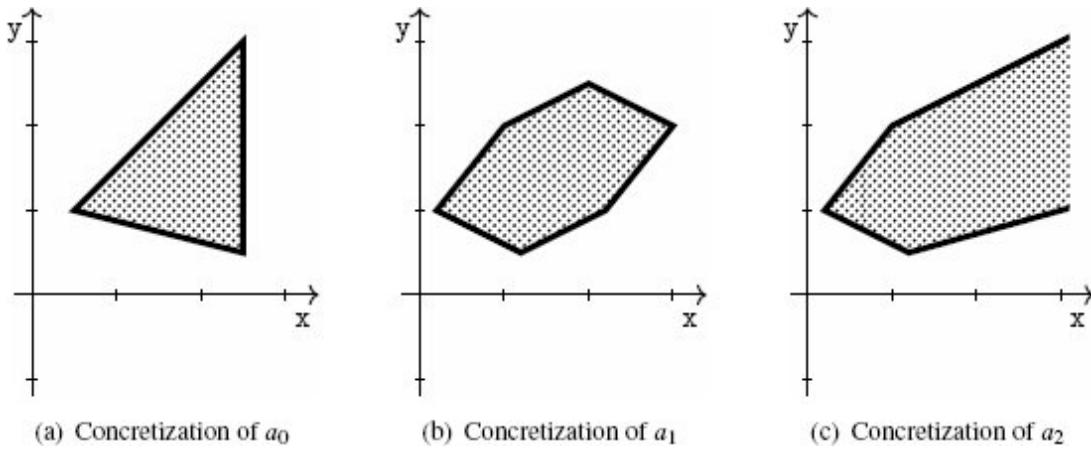


Figure 2.8

Convex polyhedra abstraction

Another interesting remark about the convex polyhedra abstraction is that concrete sets of points have no best abstraction in general. A disk of diameter 1 provides an example of a concrete set without a smallest enclosing convex polyhedron. On the other hand, *some* concrete sets have a best abstraction (in particular, any set that is a convex polyhedron is its own smallest enclosing convex polyhedron).

In the previous paragraphs, we have provided a few common examples of abstract domains, but many others can be defined and are useful to capture all sorts of constraints (simple or complex, relational or non-relational).

Abstraction of the Semantics of a Program We can now refine the goal of the rest of the chapter. We have set up the notion of abstraction of sets of program states and have shown a few basic abstract domains, adapted to express different kinds of properties. In the next sections, we aim at defining static analysis algorithms to compute in a fully automatic way an over-approximation of the states that a program may reach. Such an over-approximation will be described by an abstract element in one of the abstract domains that we have sketched. It is called a *conservative abstraction* of the program semantics.

Example 2.7 (Abstractions of reachable states) We consider the program of example 2.3. Figure 2.9(a) shows all the states that this program may reach; a few program executions were sketched in figure 2.3(b), and we consider here the set of all the states that can be reached in at least one execution. We then show the best abstractions that can be computed for this set of states:

- Using the intervals abstract domain in figure 2.9(b).
- Using the convex polyhedra abstract domain in figure 2.9(c).

Obviously, the abstraction based on convex polyhedra is much tighter, even though it is still approximate, due to convexity.

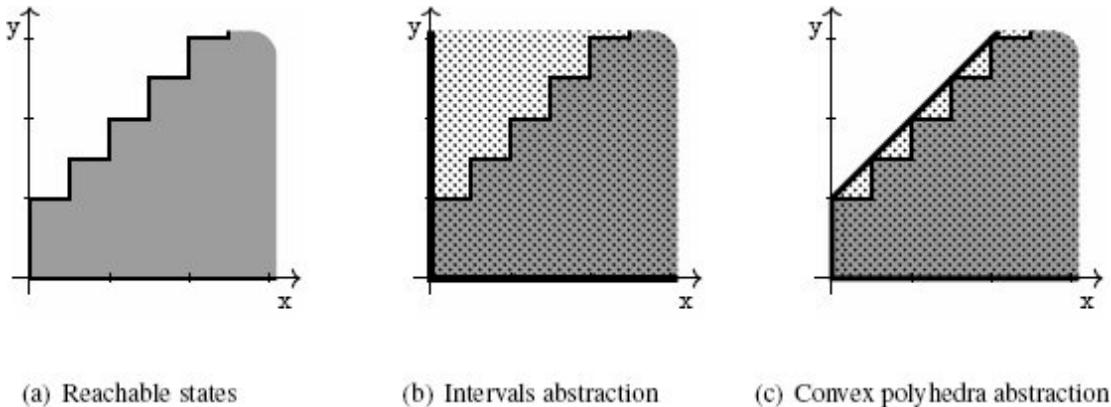


Figure 2.9

Program reachable states and abstraction

As shown in example 2.7, not all abstractions of the semantics of the program are equivalent. Abstractions that describe fewer points are more selective, since they filter out more concrete points, and are thus more likely to help prove that the reachable states are included in a specific set, to prove the property of interest. This means that set inclusion here is fundamental to our study:

- For an abstract element to be a conservative abstraction of the semantics of programs, it should include all the points that are reachable according to the semantics.
- If two abstract elements a_0, a_1 are such that $\gamma(a_0)$ is included into $\gamma(a_1)$, this means that a_0 is more precise than a_1 in the sense that it allows proving stronger semantic properties.

2.3 A Computable Abstract Semantics: Compositional Style

As the notion of abstraction has been set up in section 2.2, we now show how to derive step-by-step an over-approximation for the states that are visited by a program.

In this section, we introduce a compositional approach to static analysis, based on the step-by-step computation of the effect of each program command. More precisely, given an abstraction of a set of states that denotes a pre-condition (i.e., a set of program execution starting points), we propose to compute an abstract state that over-approximates the set of all the states that may be observed after running that command from the pre-condition (this set is usually called a post-condition). To analyze a sequence of commands, this technique composes the analyses of each sub-command, which is why it is called *compositional*.

Intuitively, this approach incrementally discovers an over-approximation of the set of reachable states of the program. Thus, this also makes it possible to verify that a program never reaches any state in the error zone. Using an accumulator, it is also possible to keep track of an abstraction of all the reachable states of the program.

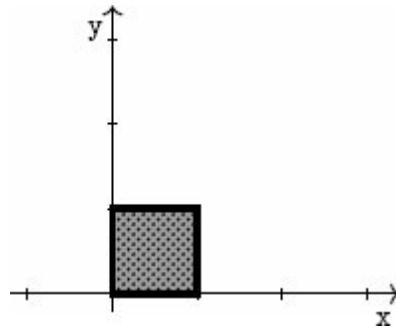


Figure 2.10

Analysis of initialization

2.3.1 Abstraction of Initialization

We start with the effect of the initialization statement that appears at the beginning of programs. At the concrete level, a program initialization statement simply asserts that the initial state of a program execution is located in a given region \mathcal{R} .

To produce an abstraction of the result of initialization, the static analysis simply needs to produce an abstract element that over-approximates the region \mathcal{R} .

When the abstract domain features a best abstraction function α and when the best abstraction of the region \mathcal{R} is computable, then the abstract element $\alpha(\mathcal{R})$ provides a solution. This is the case with the intervals abstract domain and with the signs abstract domain.

When the abstract domain does not have a best abstraction function or when this best abstraction is not computable, any abstract element a such that $\gamma(a)$ includes \mathcal{R} can be chosen. For instance, the convex polyhedra abstract domain does not feature a best abstraction function; however,

- if \mathcal{R} is a convex polyhedron, then it can be used as an over-approximation of itself;
- otherwise, an enclosing box can be found by using the intervals abstract domain abstraction, and this enclosing box is also an enclosing convex polyhedron, so it also provides an admissible solution (although an imprecise one).

Example 2.8 (Initialization) We consider the program of example 2.3. Figure 2.10 shows the best abstraction of the initial states, both with the intervals abstract domain and with the convex polyhedra abstract domain. We remark that this abstraction is exact; namely, it incurs no loss of precision.

2.3.2 Abstraction of Post-Conditions

We now discuss basic geometric transformations and try to find a systematic way to over-approximate their output, when given an abstraction of their input. We first fix some terminology:

- An *abstract pre-condition* is an abstraction of the states that can be observed *before* a program fragment.
- An *abstract post-condition* is an abstraction of the states that can be observed *after* that program fragment.

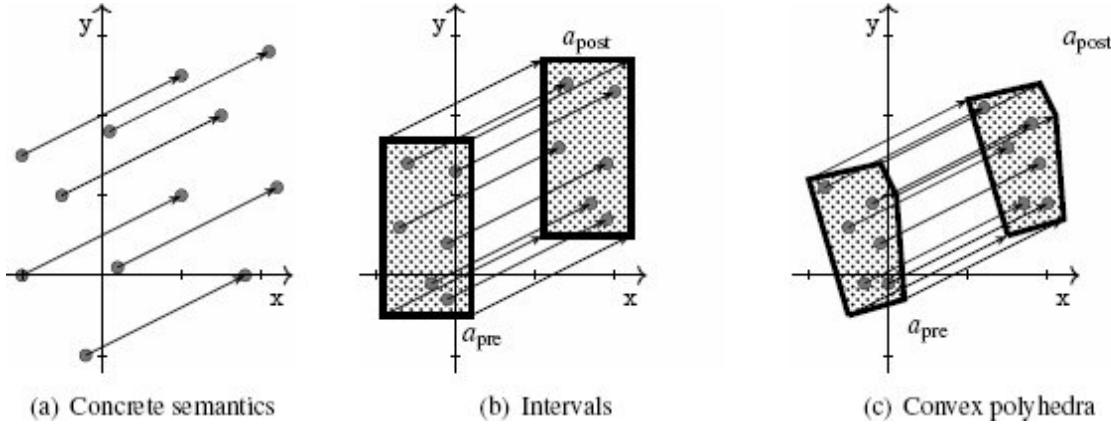


Figure 2.11

Abstraction of the result of a translation

Effect of a Translation We assume an abstract pre-condition a_{pre} and consider program **translation**(u, v). When the program is run in state (x, y) in $\gamma(a_{\text{pre}})$, the result is the state $(x + u, y + v)$. Thus, the set of all the images of the points in $\gamma(a_{\text{post}})$ can be obtained very simply by translating $\gamma(a_{\text{pre}})$ by (u, v) . Thus, to produce an over-approximation of the effect of the translation, we simply need to compute an abstract element a_{post} that contains all the points obtained by applying the translation to a point in $\gamma(a_{\text{pre}})$.

Example 2.9 (Translation) We consider **translation**(2,1) and the computation of abstract post-condition with a couple of example abstract domains. The effect of the program is shown in figure 2.11(a): any execution boils down to a pair of states.

Figure 2.11(b) demonstrates the computation of an abstract post-condition with the abstract domain of intervals under the assumption of a given abstract pre-condition. The element a_{post} is obtained directly from a_{pre} by applying translation (2,1). If we consider a translation defined by another vector, an abstract post-condition in the intervals abstract domain can be derived from the abstract pre-condition in a similar way.

The case of the abstract domain of convex polyhedra is similar to the case of intervals, as shown in figure 2.11(c).

In both cases, we note that the abstract post-condition not only contains all the points in the image of the concretization of the pre-condition but also contains no other point; in this sense, the post-condition is exact.

Effect of a Rotation We now consider a program of the form $\text{rotation}(u,v,\theta)$. The same reasoning toward the design of an automatic algorithm to compute abstract post-conditions from an abstract pre-condition as for the translation still holds. We discuss this transformation in the following example.

Example 2.10 (45° rotation) To fix the ideas, we assume that $(u,v) = (0,0)$ and that $a = 45^\circ$ (45° rotation around the origin). A few concrete executions are depicted in figure 2.12(a).

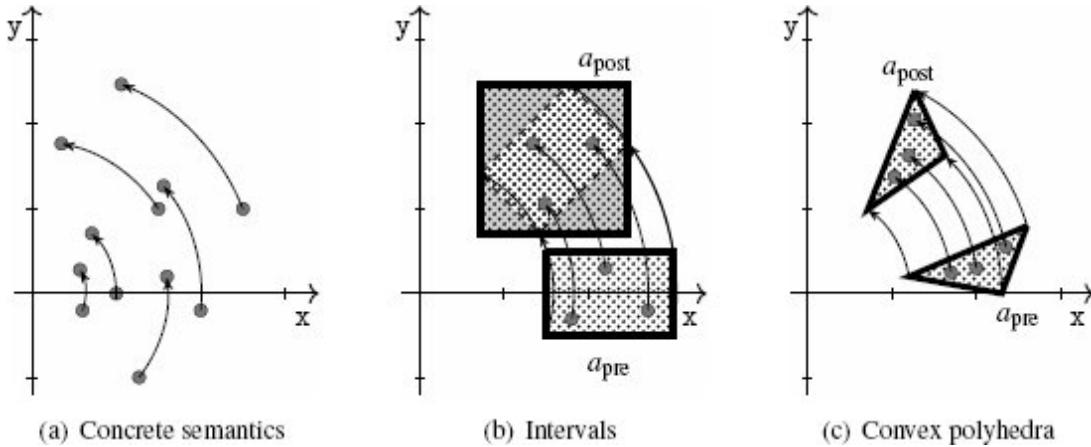


Figure 2.12

Abstraction of the result of a 45° rotation

First, we discuss the case of polyhedra, as it is actually simpler than the case of intervals. Figure 2.12(c) shows that the abstract post-condition can be computed exactly in the same way as for the translation in the previous paragraph. Indeed, if the analysis computes the image of the abstract pre-condition by the rotation, the resulting convex polyhedron contains all the images of the points in the concretization of the pre-condition and thus provides a precise over-approximation of the points that the program may reach after the rotation.

The case of intervals is shown in figure 2.12(b). Intuitively, rotating the pre-condition should give a safe over-approximation of the points that the program may produce after the rotation, but the rotated box is not a valid element of the intervals abstract domain. Indeed, we defined the intervals abstract domain as the set of (finite or infinite) rectangles that are parallel to the axes, and the image of the pre-condition by the rotation is not parallel to the axes. Therefore, to produce a conservative post-condition, that is also an element of the abstract domain, the analysis should produce a bigger box, which is parallel to the axes, as shown in figure 2.12(b). But this result is somewhat imprecise; indeed, as usual, the area filled with dots describes the result of the analysis, and the part of that zone that has a gray background corresponds to points that cannot be observed when running the program from any point in the pre-condition, yet these points have to be included in the result of the analysis due to the limited expressiveness of the intervals abstraction. Such imprecisions may ultimately prevent the analysis from proving the property of interest.

Conservative Abstract Transfer Functions Based on the two transformations that we have studied so far, we now summarize how the analysis should compute post-conditions in the abstract level. In general, we call an abstract operation that accounts for the effect of a basic program statement a *transfer function*. The definition below formalizes the soundness property that all transfer functions should satisfy.

Definition 2.6 (Sound analysis by abstract interpretation in compositional style) We consider a static analysis function *analysis* that inputs a program and an abstract pre-condition and returns an abstract post-condition. We say that *analysis* is sound if and only if the following condition holds:

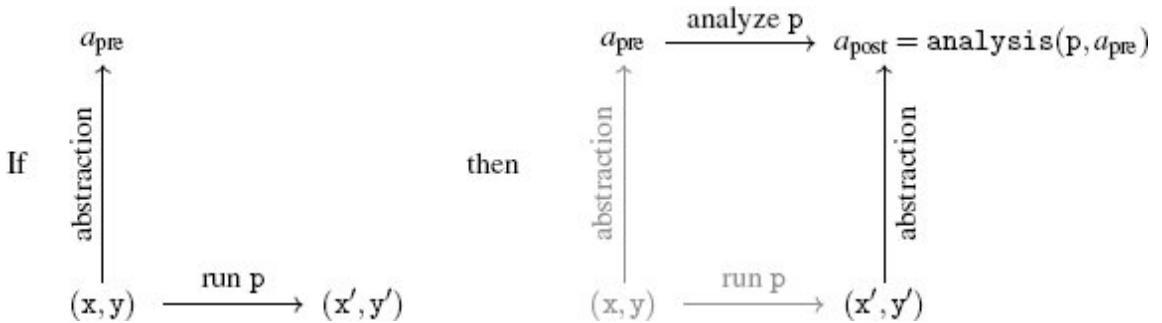


Figure 2.13

Sound analysis of a program p

If an execution of p from a state (x, y) generates the state (x', y') ,

then for all abstract element a such that $(x, y) \in \gamma(a)$,

$$(x', y') \in \gamma(\text{analysis}(p, a))$$

Intuitively, this property states that the analysis should cover all executions of the program: whenever there exists an execution starting from a state that lies inside the abstract pre-condition, the output state should also belong to the abstract post-condition. The diagram of figure 2.13 gives an intuitive presentation of the soundness property: when a concrete state can be described by an abstract pre-condition (bottom to top arrow in the left diagram) and is the starting point of an execution that reaches a final state (left to right arrow in the left diagram), running the analysis will close the

diagram and return an over-approximation of the post-state, as shown in the right diagram.

The transfer functions shown in the previous paragraphs for translations and rotations, for both the interval and polyhedra abstract domains, satisfy the soundness property. Furthermore, we will make sure in the following that the analysis algorithms that we design for other program constructions still preserve this property.

This technique is an instance of *abstract interpretation*: it lets the analysis evaluate each program construction one by one, a bit like a standard interpreter would, albeit in the abstract domain.

At this point, we have defined the following:

$$\begin{aligned} \text{analysis}(\text{translation}(u, v), a) &= \begin{cases} \text{return an abstract state that contains} \\ \text{the translation of } a \end{cases} \\ \text{analysis}(\text{rotation}(u, v, \theta), a) &= \begin{cases} \text{return an abstract state that contains} \\ \text{the rotation of } a \end{cases} \end{aligned}$$

Definition 2.6 entails that the analysis will produce sound results in the sense of definition 1.2 when considering the property $\neg \mathcal{D}$ of interest. Since the analysis over-approximates the states the program may reach, if it claims that $\neg \mathcal{D}$ is not reachable, then we are sure that the program cannot reach $\neg \mathcal{D}$.

On the other hand, this definition does not rule out imprecisions. Thus, it accepts analyses that produce coarse over-approximations. In the previous paragraphs we saw both precise analyses and imprecise analyses:

- With the convex polyhedra abstract domain, both the analysis functions for the translation and rotation are precise.
- On the other hand, with the intervals abstract domain, the analysis function for the rotation is imprecise.

Such imprecisions entail that the analysis is not complete in the sense of definition 1.3, and that it may fail to prove that a given region is unreachable.

In the following, we continue the definition of the `analysis` function that can compute sound abstract post-conditions for any program in our language. We proceed by induction over the syntax of programs. Indeed, we have already seen how to handle basic operations (initialization, translations, and rotations); thus, we now consider inductive cases.

The case of sequences of operations is trivial: to compute an abstract post-condition for $p_0;p_1$, we start from the abstract pre-condition a , compute an abstract post-condition $\text{analysis}(p_0,a)$ for p_0 , and then feed the result as the pre-condition to compute an abstract post-condition for p_1 :

$$\text{analysis}(p_0;p_1,a) = \text{analysis}(p_1, \text{analysis}(p_0, a))$$

The other cases (for non-deterministic choice and iteration) are a bit more complex than the sequence case.

2.3.3 Abstraction of Non-Deterministic Choice

We now assume that p_0 and p_1 are two programs that we already know how to analyze, and we propose constructing a way to over-approximate post-conditions for $\{p_0\}\text{or}\{p_1\}$.

Let a be an abstract pre-condition, and state $(x,y) \in \gamma(a)$. Intuitively, we should consider two cases:

- either p_0 is executed, and the result is in $\gamma(\text{analysis}(p_0,a))$; or
- p_1 is executed, and the result is in $\gamma(\text{analysis}(p_1,a))$.

Thus, the analysis should simply produce an over-approximation of both $\text{analysis}(p_0,a)$ and $\text{analysis}(p_1,a)$.

The computation of an over-approximation for two abstract elements can be done in a systematic way for all the abstract domains that we considered in section 2.2. We can remark that this operation computes an over-approximation for the union of two sets of points viewed as abstract elements. Thus, we denote this abstract operation by **union**. In the case of intervals, the analysis should simply compute the minimum of lower bounds and the maximum of greatest bounds for both dimensions. In the case of convex polyhedra, it should simply produce a convex hull for both abstract elements. To summarize:

$$\begin{aligned} \text{analysis}(\{p_0\}\text{or}\{p_1\},a) = \\ \text{union}(\text{analysis}(p_1,a),\text{analysis}(p_0,a)) \end{aligned}$$

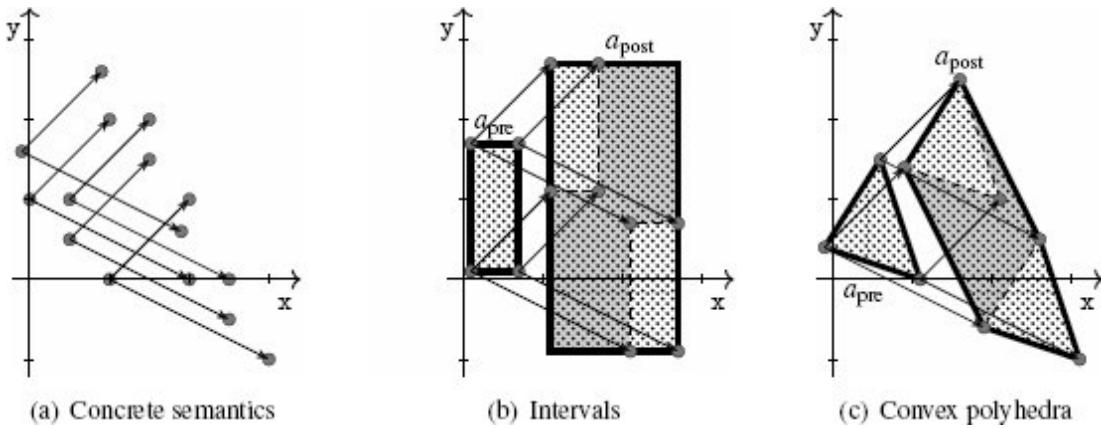


Figure 2.14

Abstraction of the result of a non-deterministic choice

Example 2.11 (Analysis of non-deterministic choice) In this example, we consider the very simple program below, and we show its analysis with both intervals and convex polyhedra:

`{translation(2,1)}or{translation(-2,-1)}`

Figure 2.14(b) shows the computation of an abstract post-condition in the intervals abstract domain, and figure 2.14(c) shows the computation of an abstract post-condition in the convex polyhedra abstract domain. These two cases are quite similar since each branch of the non-deterministic choice boils down to a geometric translation (which induces a translation of the shape of abstract elements), and the analysis should then return an over-approximation of the effects of both branches. In both domains, this operation incurs a significant loss of precision due to the approximation of the convex hull.

The above example shows another reason for the incompleteness of our analysis, as it cannot express precise disjunctive properties. This is a common issue in static analysis, and we present several solutions to this problem in section 5.1.

2.3.4 Abstraction of Non-Deterministic Iteration

Non-deterministic iteration is the last construction that we have to define the analysis for, and is also the most difficult to analyze since it can produce executions of any length and even infinite executions. Therefore, the analysis should compute in *finite time* an over-approximation for *infinitely many arbitrarily long executions*. Still, we observed in example 2.3 that we

can derive interesting properties about such programs with rather short informal proofs. Thus, we generalize this approach in this paragraph and design analysis algorithms that compute an over-approximation for the set of output states of a loop.

Note that the abstract post-condition produced as the analysis result describes only the final states of the terminating program executions. This result means that, if a concrete execution terminates, then this abstract post-condition holds. The result does not mean that the iteration will terminate with this abstract post-condition. This is because the halting problem cannot be computed exactly in finite time.

In the following, we consider the following program p that consists of a loop with body b :

$$p ::= \begin{cases} \text{iter}\{ } \\ \quad b \\ \quad \} \end{cases}$$

We can discriminate the executions of p depending on the number of iterations of the loop; indeed, an execution of program p executes b either zero times, one time, two times, three times, or so on. Thus, p is conceptually equivalent to the following (infinite) program:

$$\begin{aligned} &\{\} \\ &\text{or}\{b\} \\ &\text{or}\{b;b\} \\ &\text{or}\{b;b;b\} \\ &\text{or}\{b;b;b;b\} \\ &\vdots \end{aligned}$$

This program fully eliminates the loop and resorts only to the **or** construct, which can be analyzed as described in section 2.3.3, though it obviously cannot be completely written since it would be infinite. However, if we focus on the executions that spend at most k iterations in the loop, we can easily write a program without a loop that has exactly the same behaviors. For any integer k , we let b_k denote the program that iterates b k times (b_0 is

$\{\}$, b_1 is b , b_2 is $b;b$, and so on). Moreover, we write p_k for $\{b_0\} \text{or} \{b_1\} \text{or} \dots \text{or} \{b_{k-1}\} \text{or} \{b_k\}$. In short:

```

program p0 is {}
program p1 is {}or{b}
program p2 is {}or{b}or{b;b}
program p3 is {}or{b}or{b;b}or{b;b;b}
:

```

Then we observe the following equivalence, which relates these programs all together:

p_{k+1} is equivalent to $p_k \text{or} \{p_k; b\}$

Indeed, an execution of p_{k+1} either executes the loop at most k times (hence, it is an execution of p_k), or runs it $k + 1$ times exactly (and then it is an execution of $p_k; b$). Conversely, one can prove that an execution of $p_k \text{or} \{p_k; b\}$ is also an execution of p_{k+1} .

Therefore, the analysis of this sequence of programs can be computed recursively as follows:

```

analysis(pk+1, a) =
union(analysis(pk, a), analysis(b, analysis(pk, a)))

```

This approach corresponds to the analysis algorithm that inputs an abstract pre-condition a , stores it into a variable R , and iterates the operation:

$R \leftarrow \text{union}(R, \text{analysis}(b, R))$

Moreover, as shown above, any execution of p can be characterized by the number of times it iterates over the loop. Thus, any execution is ultimately covered by repeating this iterative abstract computation.

The following example illustrates this approach.

Example 2.12 (Abstract iteration) We consider the program below, which starts at a point located in a triangle and iterates a basic geometric translation a non-deterministically chosen number of times:

```

init({(x,y) | 0 ≤ y ≤ 2x and x ≤ 0.5});
iter{
    translation(1,0.5)
}

```

We assume that the analysis uses the convex polyhedra abstract domain. Then the set of states observed after initialization and before the **iter** statement is shown in figure 2.15(b). We show in figure 2.15(c), figure 2.15(d), and figure 2.15(e) the first three iterations of the analysis algorithm sketched above. The imprecision is inherent in the computation of over-approximations (in gray) of abstract elements as in the previous examples.

While this process does not terminate, we observe that repeating it forever would yield the result shown in figure 2.15(f), which also provides a sound approximation of all the possible output states of the program.

The iterative algorithm demonstrated in example 2.12 will actually always terminate if using the signs abstract domain. Indeed, this abstract domain has a finite number of abstract elements, and when the iterative algorithm computes $R \leftarrow \text{union}(R, \text{analysis}(b, R))$, the value of R will converge after finitely many steps: whenever it updates R , either the new value is the same as the previous one (in that case, so will be all the other subsequent values since they are computed using the same formula), or the new value denotes a *strictly* less precise property. Since the number of abstract properties is finite, the latter case will occur at most finitely many times. Therefore, at some point the value of R stabilizes, and then this value over-approximates the behaviors observed after *any* number of iterations. As a consequence, the termination of signs analysis is guaranteed.

However, we have not solved the issue of termination in the general case yet. The iteration technique used in example 2.12 will obviously not allow for a terminating analysis with the convex polyhedra abstraction or with the interval abstraction.

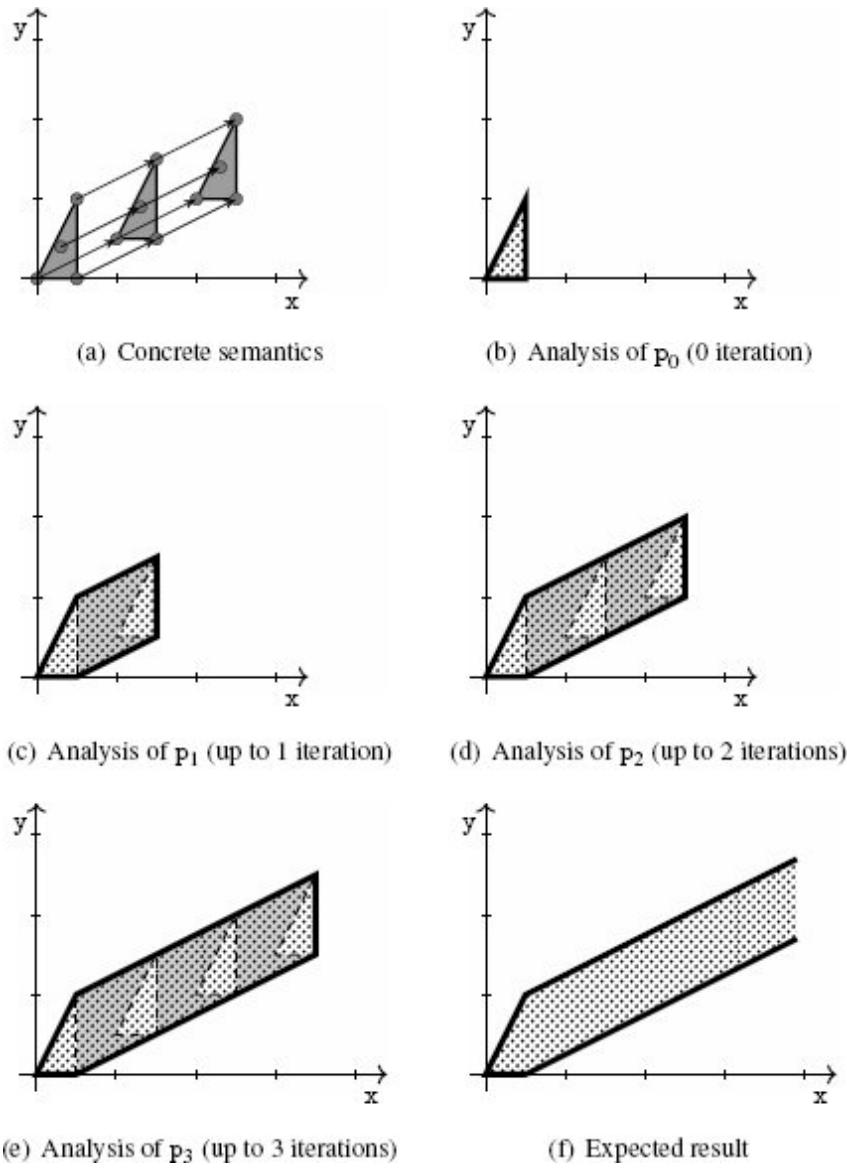


Figure 2.15

Abstract iteration

To ensure termination of the analysis, we need to enforce the convergence of abstract iterates, possibly at the price of a coarser result. Note that a common way to prove that an algorithm terminates involves finding a strictly positive value that decreases strictly over time toward a finitely reachable basis. Thus, a way to enforce the termination of the

analysis is to exhibit such a measure. Very often, non-termination is due to some loop indexes not being incremented properly, preventing such a decreasing measure to exist. Intuitively, this is the issue the iterative analysis algorithm we sketched above suffers from, as shown in example 2.12.

Another interesting observation is that abstract elements are made of finite sets of constraints. Therefore, another way to over-approximate abstract elements that arise in the abstract iteration would consist in forcing this number of constraints to decrease (possibly down to zero) until it stabilizes, thereby recovering termination.

Given the current constraint a_0 , suppose that analyzing one more iteration generates a_1 . To have an approximate constraint that subsumes both, we can let the analysis:

- keep all constraints of a_0 that are also satisfied in a_1 and
- discard all constraints of a_0 that are not satisfied in a_1 (hence to subsume a_1).

Applying this method to abstract iterates will produce a sequence of abstract elements with a positive, decreasing number of constraints until the sequence stabilizes. This method is an instance of a general technique called *widening*, which enforces the convergence of abstract iterates. We denote this operator by `widen`:

$$\text{operator } \text{widen} \quad \left\{ \begin{array}{l} \text{over-approximates unions} \\ \text{enforces convergence} \end{array} \right.$$

Stabilization holds when the concretization of the next iterate is included in that of the previous one. For all the abstract domains considered in this chapter, this inclusion can be decided in the abstract level simply by checking geometric inclusion. We thus let `inclusion` denote a function that inputs two abstract elements a_0, a_1 and returns **true** only when it can prove that $\gamma(a_0) \sqsubseteq \gamma(a_1)$:

operator `inclusion` returns **true** only when it succeeds checking
inclusion

As a conclusion, the following algorithm computes an abstract post-condition for the loop construction:

```

analysis(iter{p},a) = { R ← a;
repeat
    T ← R;
    R ← widen(R,analysis(p,R));
until inclusion(R,T)
return T;
}

```

This iteration technique will produce a sound result since it over-approximates the abstract elements produced by the sequence of iterates without widening, and its limit (reached after finitely many iterates) also over-approximates all the abstract elements produced by the sequence of iterates without widening and, thus, the states that the program may reach.

The following example illustrates its use in practice.

Example 2.13 (Abstract iteration with widening) We consider the same program as in example 2.12. Figure 2.15 shows the sequence of abstract iterates using the widening technique. This sequence converges after only two iterations and produces a (rather coarse) over-approximation of the reachable states of the program (shown in figure 2.15(a)). The most interesting point is the computation of the abstract element shown in figure 2.16(b) from the two triangles obtained in the first two iterations:

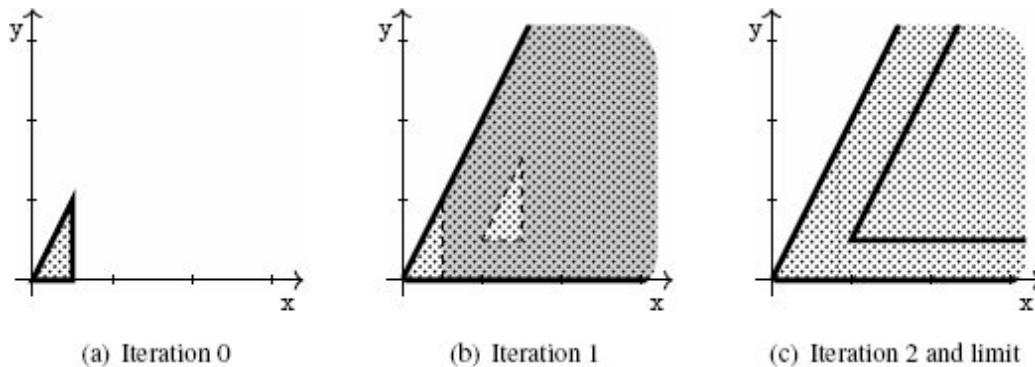


Figure 2.16

Abstract iteration with widening

- The constraints $0 \leq y$ and $y \leq 2x$ are stable as they are satisfied in the translated triangle; thus, they are preserved.
- The constraint $x \leq 0.5$ is not preserved; thus, it is discarded.

The result obtained in the example clearly shows that widening is another source of imprecision and, thus, of potential incompleteness. Indeed, to

ensure convergence in finite time, the analysis weakens the abstract elements more aggressively, adding many points that cannot be observed in any real program execution, as shown in figure 2.16(b).

Fortunately, we can implement many techniques to make the analysis of loops more precise. The example below demonstrates a classic such technique on the same code.

Example 2.14 (Loop unrolling) We observe that we can rewrite a program with a loop in different ways than the one used so far in this section. In particular, the program of example 2.12 is equivalent to the following program:

```

init({(x,y) | 0 ≤ y ≤ 2x and x ≤ 0.5});
{}or{
    translation(1,0.5)
}
iter{
    translation(1,0.5)
}

```

In essence, analyzing this second version instead has the following effect on the analysis. Indeed, for the first iteration, the **union** operator will be used, and for all subsequent iterations, **widen** will be used instead. When computing widening at iteration 2, all constraints are stable, but the constraint $x \leq 1.5$. This produces the result shown in figure 2.17(c). Thus, both the result of the first iteration (shown in figure 2.16(b)) and the widening output (shown in figure 2.17(c)) are much more precise than with the standard widening iteration technique presented in example 2.13.

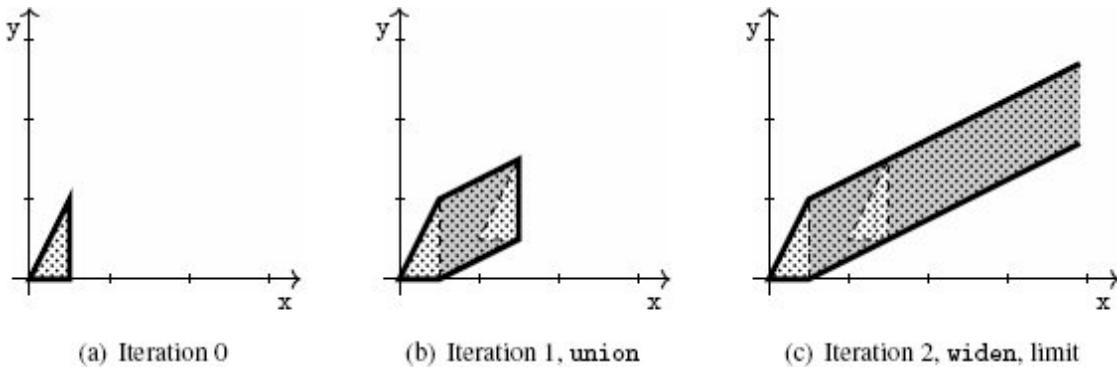


Figure 2.17

Abstract iteration with widening and unrolling

2.3.5 Verification of the Property of Interest

The analysis function that we have designed allows verifying the reachability property of interest introduced in section 2.1.

While the analysis function that we have shown so far returns only an over-approximation of the output states (and not of all the intermediate reachable states), it actually computes as intermediate results over-approximations for *all* the reachable states of the input program. Let us consider the case of a sequence $p_0;p_1$. The analysis then returns $\text{analysis}(p_1, \text{analysis}(p_0, a_{\text{pre}}))$. We observe that, after analyzing p_0 and before analyzing p_1 , the analysis holds an over-approximation of all the states that can be observed after executing p_0 and before executing p_1 (the abstract element $\text{analysis}(p_0, a_{\text{pre}})$). The same holds for each kind of instruction of our language.

As a consequence, the analysis can attempt to verify the property of interest by checking that the abstract elements computed at each step have an empty intersection with \mathfrak{D} , or, equivalently, are included in $\neg\mathfrak{D}$. This inclusion can be fully verified in the abstract level, using the same inclusion test we used for checking the termination of the sequences of abstract iterates.

We assume the analysis uses the abstract domain of convex polyhedra and illustrate successful and unsuccessful analyses in the two examples below.

Example 2.15 (Successful verification) *Figure 2.18(a) shows the over-approximation computed for the set of all the reachable states of the program of example 2.3. In this case, the over-approximation does not intersect \mathfrak{D} ; thus, the analysis proves the program correct. Again, this result is in line with the conclusion of example 2.7 that this program is correct.*

Example 2.16 (Unsuccessful verification) *Figure 2.18(b) shows the over-approximation computed for the set of all the reachable states of the program of example 2.1 (region filled with dots). Since this zone corresponds to the whole field and intersects the error zone \mathfrak{D} , the analysis cannot prove the property of interest for this program. This was to be expected. Example 2.2 has shown executions of this program that enter \mathfrak{D} . While the analysis rightfully flags this program as “potentially wrong,” it does not produce a proof that the program is definitely wrong (though we will see in section 5.5 that we can propose static analysis techniques that achieve this proof in certain cases).*

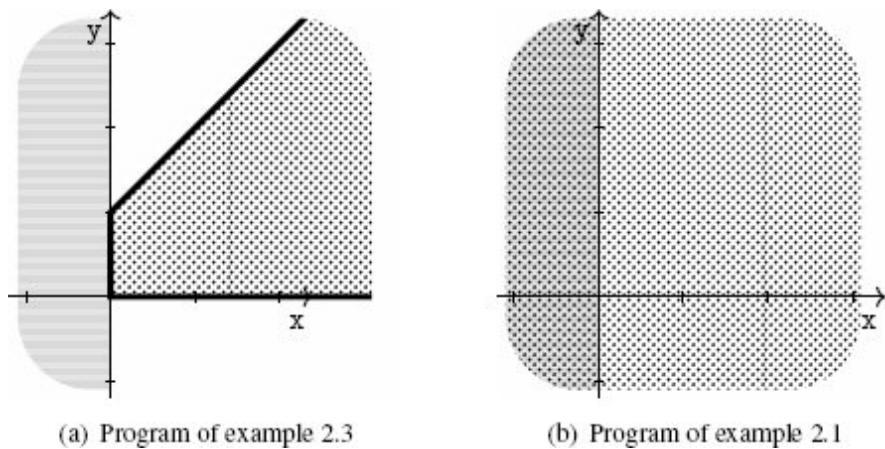


Figure 2.18

Abstractions of reachable states

2.4 A Computable Abstract Semantics: Transitional Style

In section 2.3, the `analysis` function has no explicit machinery to collect all intermediate, reachable states. In other words, it is extensionally defined, analogous to the denotational (or compositional) approach to the semantics. Its inductive definition over the syntactic structure of the program returns just a post-state of the input program from a pre-state. No collection of intermediate states is manifest in the definition. As discussed in section 2.3.5, however, a simple monitoring mechanism on top of `analysis` can collect all occurring intermediate states.

In this section, we introduce a different style of the analysis function. The new analysis function computes, from the outset, all occurring intermediate states. This formulation is analogous to an operational approach to the semantics.

This transitional style provides us with another convenient perspective that sheds new light on static analysis. In subsequent chapters, we will discuss how the compositional style is better suited for some problems, whereas the transitional style is a better fit for others. Therefore,

understanding both styles is beneficial to better grasp static analysis techniques in general.

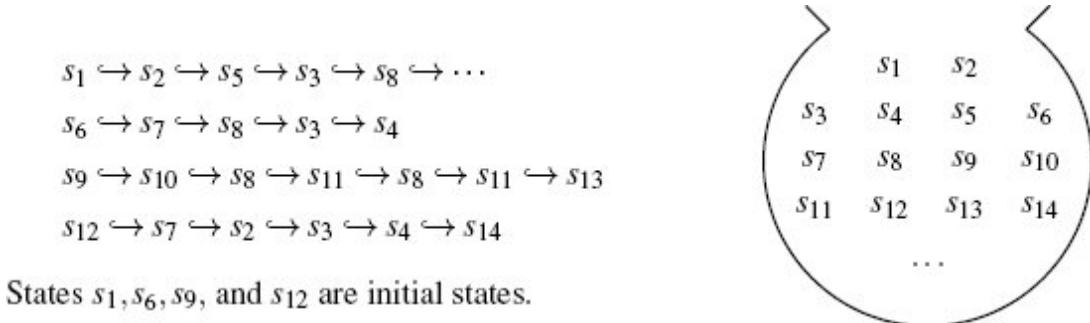


Figure 2.19

Transition sequences and the set of occurring states

2.4.1 Semantics as State Transitions

In the transitional style, we view an execution of a program as a sequence of transitions between states. This transition sequence exposes all the states that occur during the execution.

Let us consider the example language (section 2.1) of this chapter. In this language, a program execution moves a point in the two-dimensional space. In this case, a state can be defined as a pair comprising a statement label l and a point p in the two-dimensional space.

A single transition

$$(l, p) \leftrightarrow (l', p')$$

between states represents that the program at statement label l transforms the point p to p' and passes it to the next statement label l' for continuation.

One proper transition corresponds to a “single-step” execution of a basic statement. For a compound statement that consists of other statements, its execution consists of the transitions of its sub-statements.

An example of transition sequences for an example program is shown in the next section, after we define how we represent programs and what we mean by “statement labels.”

State Transitions and the Collection of all States Let our analysis goal be to collect all the states occurring in all possible transition sequences of the input program. Given such a set of all reachable states, we can check, for example, whether every reachable state remains in a safe zone of our interest.

Figure 2.19 illustrates transition sequences and the collection of states occurring in the sequences. Each node s_i is a state (l, p) : a pair comprising a statement label and a point set in the two-dimensional space that is to be transformed by the statement at the label. Here, we schematically show the transition sequences and occurring states. We will show in example 2.17 concrete examples of transition sequences.

Statement Labels and Execution Order We view a program as simply a collection of statements with a well defined execution order. We assign a unique label to each statement of the program. This label can be understood as the so-called *program counter* or *program point*. The execution order, between statements, the so-called *control flow*, is specified by a relation between the labels (from current program points to next program points).

Since our language has a non-deterministic choice and non-deterministic iterations, the execution order is non-deterministic too. Entering the or-statement $\{p\} \mathbf{or} \{p'\}$, the next statement to execute is either p or p' . Entering the iteration statement $\mathbf{iter}\{p\}$, the next statement to execute is either the loop body p or the next statement after the exit of the loop. The next statement of the loop body is again the iteration statement.

For example, consider an example program in figure 2.20. Each statement has a unique label. Figure 2.20(a) shows the program text with statement labels in circles. The statement corresponding to a label is circumscribed by a box with a light contour. Figure 2.20(b) shows a graphical representation of the program with its execution order as directed edges. Rectangular nodes are either basic statements or heads of compound statements. Numbered circle nodes are statement labels.

The non-deterministic function (or relation) for the execution order is defined as follows (as visible in the graph view of figure 2.20(b)):

| | | |
|-------------|-------------|--|
| next(0) = 1 | | |
| next(1) = 2 | next(1) = 5 | |
| next(2) = 3 | next(2) = 4 | |
| next(3) = 1 | next(4) = 1 | |

Note that, in general, for most modern languages the execution order (control flow) is not syntactically obvious. For example, when a language has a dynamic jump construct such as dynamic goto label, dynamic method dispatch, higher-order function call, or the raising of an exception whose target is computed only during execution, the exact execution order is not available before the static analysis. For such languages, determining the execution order should be a part of the static analysis under design or needs to be computed beforehand by another separate static analysis.

Chapter 4 presents a formal framework that covers such dynamic control-flow cases. Our example language in this chapter is one whose control-flow is obvious from the syntax.

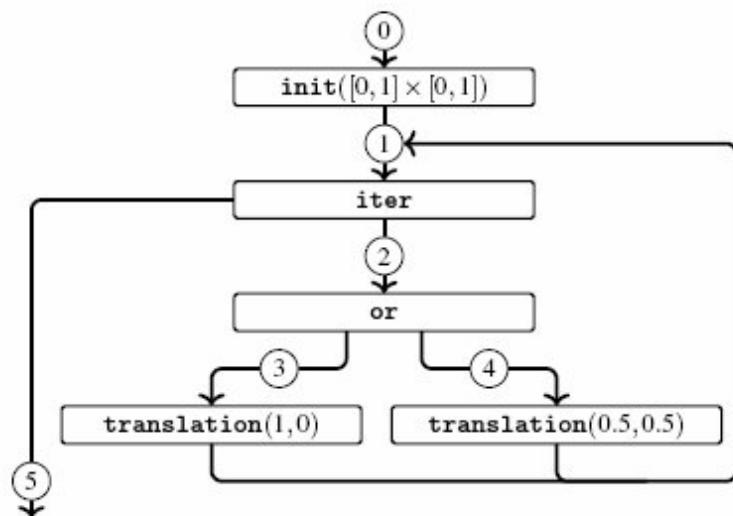
Example 2.17 (Transition sequences) *For the example program in figure 2.20, two examples of state transition (\hookrightarrow) sequences starting from statement 0 are as follows: recall that a state (l, p) is a pair of a statement label (l) and a point (p) just before being transformed by the corresponding statement. In the following, the left sequence is a transition sequence when the program terminates after two iterations; the right one is when the same program terminates after one iteration:*

```

(0) init([0,1] × [0,1]);
(1) iter{
    (2) {
        (3) translation(1,0);
    }or{
        (4) translation(0.5,0.5);
    }
}
(5)

```

(a) Text view, with labels



(b) Graph view, with labels

Figure 2.20

Example program with statement labels

| | | | | | |
|------------|---------------|------------|------------|---------------|------------|
| $(0, p_0)$ | \rightarrow | $(1, p_1)$ | $(0, p_0)$ | \rightarrow | $(1, p_1)$ |
| | \rightarrow | $(2, p_1)$ | | \rightarrow | $(2, p_1)$ |
| | \rightarrow | $(3, p_1)$ | | \rightarrow | $(4, p_1)$ |
| | \rightarrow | $(1, p_2)$ | | \rightarrow | $(1, p_3)$ |
| | \rightarrow | $(2, p_2)$ | | \rightarrow | $(5, p_3)$ |
| | \rightarrow | $(4, p_2)$ | | | |
| | \rightarrow | $(1, p_4)$ | | | |
| | \rightarrow | $(5, p_4)$ | | | |

where

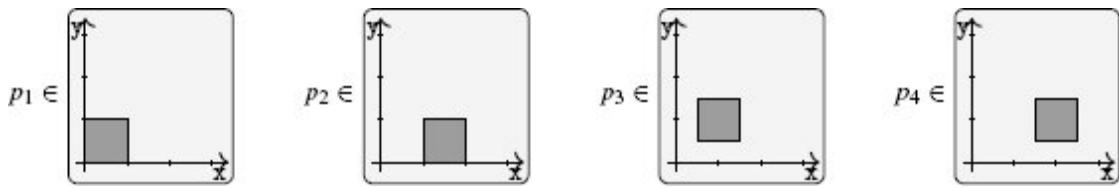


Figure 2.21 shows, on top of the graph view of the program, the right transition sequence.

2.4.2 Abstraction of States

Collecting the exact set of all the states that can occur during program executions (transition sequences) is in general either too costly or impossible in finite time. Indeed, due to loops, program executions may be arbitrarily long. Moreover, the set of initial states is also potentially infinite. The situation is worse for other conventional languages that receive inputs from outside. The number of possible inputs is usually combinatorially explosive or even infinite. That is, the number of transition sequences can be infinite too.

Hence, as discussed in section 2.3, the static computation of the set of all possible states cannot be exact in general. Our static computation may be only an approximation in an abstract world.

Now the question is what abstract world we are going to use. As an illustration among many candidates, let us use the following statement-wise abstract world:

For each statement (program point), an abstract element approximates the set of points that can occur at that program point during executions. The abstract elements for point sets are convex hull pre-conditions as used in section 2.3. In other words, an abstract state is a set of pairs of statement labels and abstract pre-conditions.

Figure 2.22 schematically shows the state abstraction we are using on top of the graphic view of a program. The areas in the two-dimensional plane

depict the set of points that can occur during executions.

2.4.3 Abstraction of State Transitions

The abstract state transition is defined over the abstract states of the preceding section. Note that an abstract state is a set of pairs of statement labels and abstract pre-conditions. An abstract transition transforms an abstract state into another abstract state.

Let $\text{Step}^\#$ be such an abstract state transition function. Given an abstract state X , $\text{Step}^\#(X)$ returns an abstract post-state. The $\text{Step}^\#$ function is defined by the one-step abstract transition operator $\leftrightarrow^\#$, lifted for a set (for an abstract state):

$$\text{Step}^\#(X) = \{x' \mid x \in X, x, \leftrightarrow^\# x'\}$$

The one-step abstract transition $x \leftrightarrow^\# x'$ is the same as the post-condition computations in section 2.3 except that the proper transition happens only for non-compound basic statements, and we reference the `next` function for the next label:

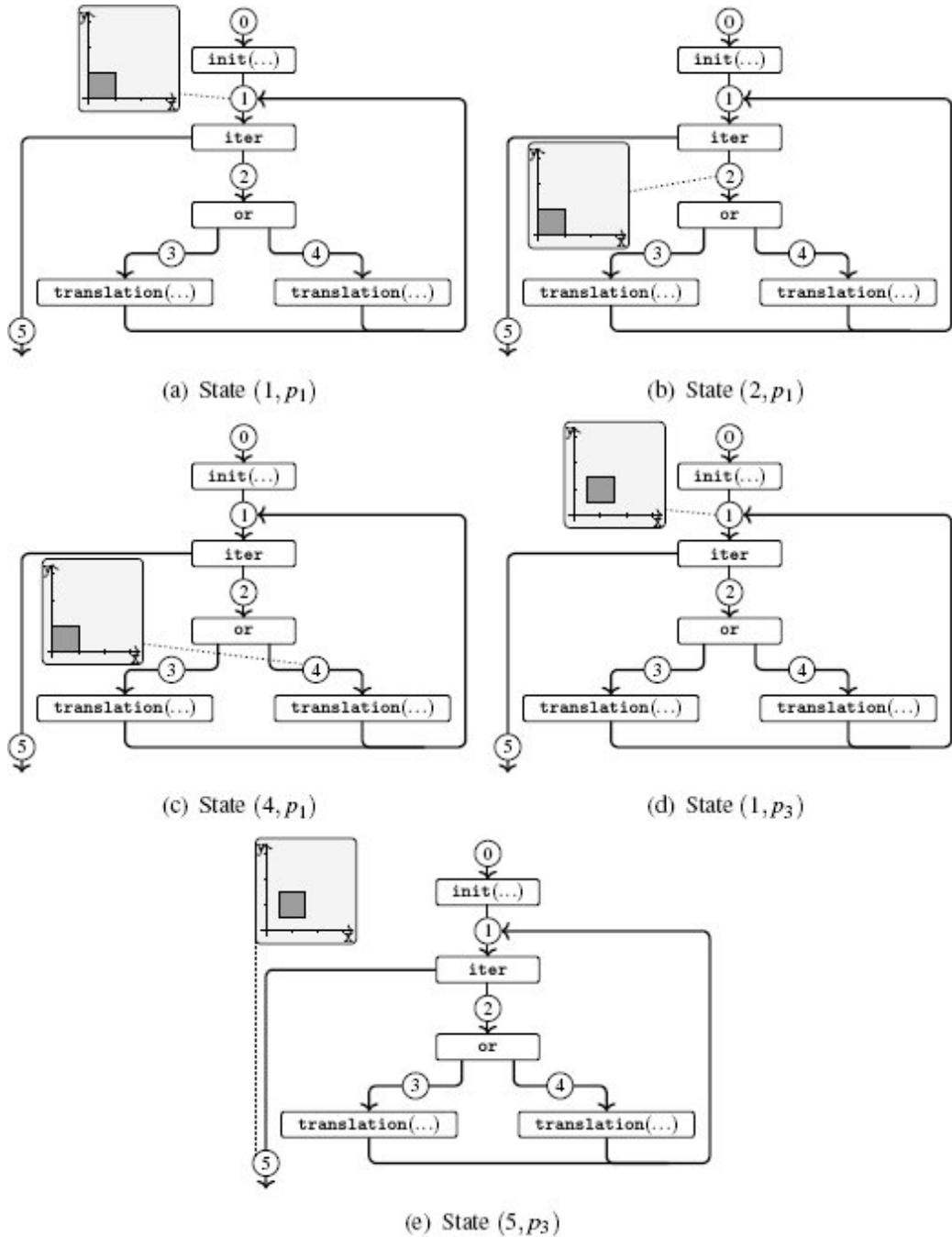
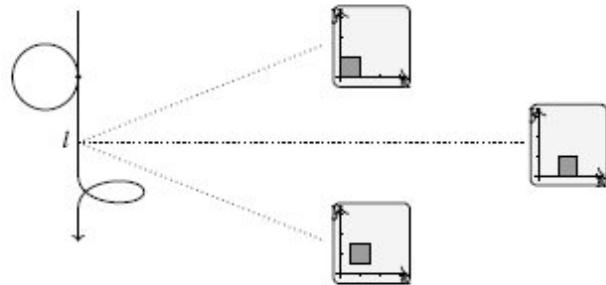


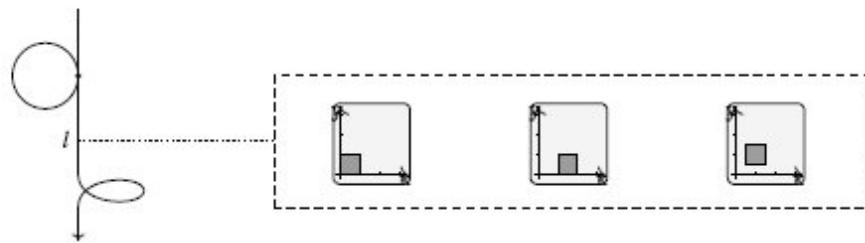
Figure 2.21

States on top of the graph view of the program. Each point p_i belongs to the rectangular area of the corresponding two-dimensional plane.

Collection of all states



Statement-wise collection:



Statement-wise abstraction:

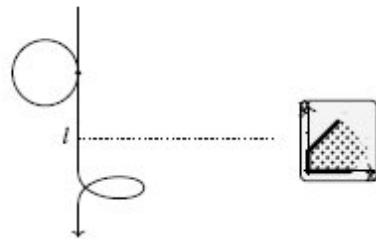


Figure 2.22

Statement-wise abstraction of all the possible states for statement label l

$$(\mathbf{or}_l, a_{\text{pre}}) \xrightarrow{\#} (\text{next}(l), a_{\text{pre}})$$

for an **or** statement at l

(figure 2.23(a));

$$(\mathbf{iter}_l, a_{\text{pre}}) \xrightarrow{\#} (\text{next}(l), a_{\text{pre}})$$

for an **iter** statement at l

(figure 2.23(b));

$$(p_l, a_{\text{pre}}) \xrightarrow{\#} (\text{next}(l), \text{analysis}(p_l, a_{\text{pre}}))$$

else, for a basic statement p_l at l

(figure 2.23(c)).

Note that the above *Step[#]* function is sound because the `analysis` function (section 2.3) is sound for basic statements (figure 2.13).

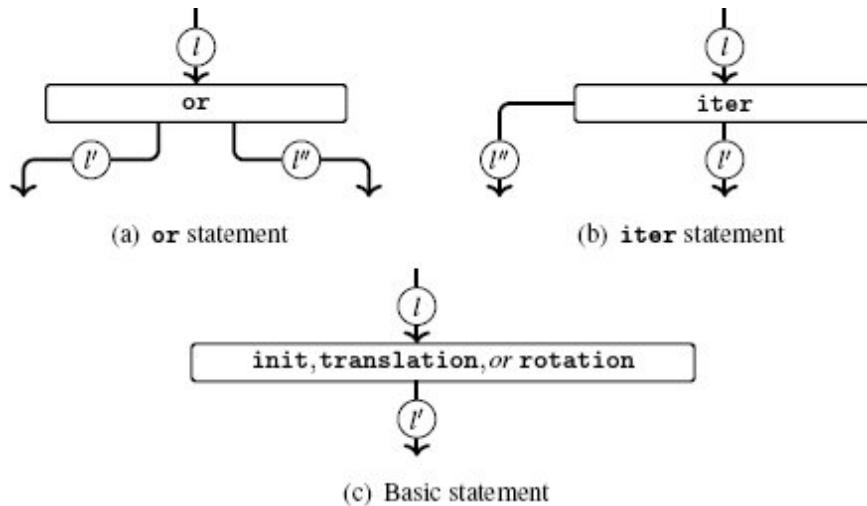


Figure 2.23

Next labels, depending on the statement of l , both l' and l'' , or l' only

2.4.4 Analysis by Global Iterations

The static analysis for collecting all abstract states should be sound. *Soundness* means that the set of concrete states implied by the analysis result over-approximates the reality.

Definition 2.7 (Sound analysis by abstract interpretation in transitional style) Let analysis_T be a static analysis function in transitional style that inputs a program and returns a set of abstract states. We say that analysis_T is sound if and only if the following condition holds:

If S is the set of states occurring in a transition sequence of p from initial state s_0 ,

then for any abstract element a such that $s_0 \in \gamma(a)$,

$$S \subseteq \gamma(\text{analysis}_T(p,a)).$$

For the input program p and an abstract state I that over-approximates all the possible initial states, the analysis result $\text{analysis}_T(p, I)$ is a set of pairs (l, a_{pre}) of statement label l and abstract pre-condition a_{pre} . The

soundness ensures that the abstract pre-conditions at label l over-approximate all the points that may occur at statement l during execution.

Such a sound analysis function $\text{analysis}_T(p, I)$ is composed of the sound abstract transition function $Step^\#$ in Section 2.4.3 as follows: letting $Step^{\#i}(I)$ denote the abstract post-states after i consecutive abstract transitions from I ,

$$\begin{aligned} Step^{\#0}(I) &= I \\ Step^{\#i+1}(I) &= Step^\#(Step^{\#i}(I)). \end{aligned}$$

This abstract state $Step^{\#i}(I)$ is sound: it subsumes all the states after i transitions from an initial state implied by I . This soundness of i consecutive applications of $Step^\#$ is clear because each step by $Step^\#$ over-approximates the results of a single-step transition. For our example language, the set I is $\{(0, \text{true})\}$, where the label 0 lies at the initial statement and its abstract pre-condition **true** implies all the points in the two-dimensional plane.

Then the analysis accumulates all the abstract states occurring at each step of the abstract transition from the initial abstract state I :

$$Step^{\#0}(I) \cup Step^{\#1}(I) \cup Step^{\#2}(I) \cup \dots$$

Now, a natural question is how to devise an algorithm that accumulates the above sequence. We remark that the above sequence is equivalent to what we illustrated in section 2.3.4 when we devised the analysis function for the **iter** statement, whose body is now $Step^\#$.

The analysis algorithm is to compute the limit ($\lim_{i \rightarrow \infty} C_i$) of the following sequence C_i :

$$C_i = Step^{\#0}(I) \cup Step^{\#1}(I) \cup \dots \cup Step^{\#i}(I).$$

Because the following equivalence holds:

$$C_{k+1} \text{ is equivalent to } C_k \cup Step^\#(C_k),$$

the analysis algorithm should simply start from I and iterate the operation

$$C \leftarrow C \cup Step^\#(C)$$

until stable.

Hence, the analysis algorithm for the input program p is a monolithic global iteration:

$$\text{analysis}_T(p, I) = \begin{cases} C \leftarrow \{I\} \\ \text{repeat} \\ \quad R \leftarrow C \\ \quad C \leftarrow \text{union}_T(C, \text{Step}^\sharp(C)) \\ \text{until } \text{inclusion}_T(C, R) \\ \text{return } R \end{cases}$$

The i -th iteration of the algorithm covers all states observed up to i execution steps of the input program.

The union_T and inclusion_T operators do the same as the union and inclusion operators (section 2.3.4), respectively, except that they are label-wise. That is, the $\text{inclusion}_T(C, R)$ returns **true** only when *at every statement label* the local point set implied from C is included in that from R . Similarly, the union_T summarizes *for each statement label* its local set of collected pre-conditions. The summarization is done by applying the union operator to reduce each local set of pre-conditions into a single pre-condition:

$$\text{union}_T(C, C') = \begin{cases} X \leftarrow \{\} \\ \text{for each label } l \text{ in } C \cup C' \\ \quad S_l \leftarrow \{a \mid (l, a) \in C\} \\ \quad S'_l \leftarrow \{a \mid (l, a) \in C'\} \\ \quad T_l \leftarrow \text{union}(S_l \cup S'_l) \\ \quad X \leftarrow X \cup \{(l, T_l)\} \\ \text{return } X \end{cases}$$

Example 2.18 (Abstract transitions) Consider the example program in figure 2.20. Suppose we use the convex polyhedra abstractions for point sets. The above analysis algorithm analysis_T stores the following C_i iterates into the variable C after i iterations. Remember that C_i covers up to i transitions of the input program:

$$\begin{aligned}
&\text{after 0 iteration, } C_0 \stackrel{\text{let}}{=} \{I\} \\
&\text{after 1 iteration, } C_1 \stackrel{\text{let}}{=} \text{union}_T(C_0, \{(1, a_1)\}) \\
&\text{after 2 iterations, } C_2 \stackrel{\text{let}}{=} \text{union}_T(C_1, \{(2, a_1), (5, a_1)\}) \\
&\text{after 3 iterations, } C_3 \stackrel{\text{let}}{=} \text{union}_T(C_2, \{(3, a_1), (4, a_1)\}) \\
&\text{after 4 iterations, } C_4 \stackrel{\text{let}}{=} \text{union}_T(C_3, \{(1, a_2), (1, a_3)\}) \\
&\vdots \quad \vdots
\end{aligned}$$

where

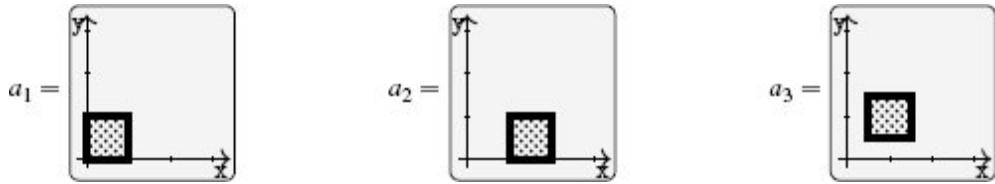
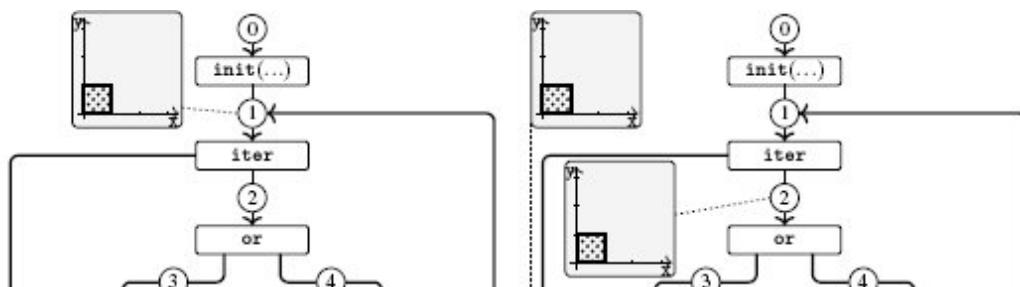


Figure 2.24 shows the snapshots of computing the iterates C_i on top of the graph view of the program:

- Figure 2.24(a), Figure 2.24(b), and Figure 2.24(c) show the pre-condition a_1 at statement labels 1, 2, 3, 4, and 5 until C_3 .
- Figure 2.24(d), Figure 2.24(e), and Figure 2.24(f) are snapshots of computing C_4 from C_3 .
- Figure 2.24(d) shows two new pre-conditions of statement 1 (post-conditions after statement 3 and 4) resulting from Step $^\#(C_3)$. They will be “united” with other pre-conditions at statement 1.
- Figure 2.24(e) shows the result of unioning a_2 and a_3 during $\text{union}\{a_1, a_2, a_3\}$ at statement 1.
- Figure 2.24(f) shows the final result of $\text{union}\{a_1, a_2, a_3\}$, union of the above and a_1 .

Analysis Algorithm with the Termination Guarantee We remark that the previous analysis_T algorithm does not guarantee termination. If a program has a loop, the analysis may iterate forever collecting new abstract pre-conditions. To guarantee the termination, we need to use the widening introduced to analyze the iteration statement in section 2.3.4.



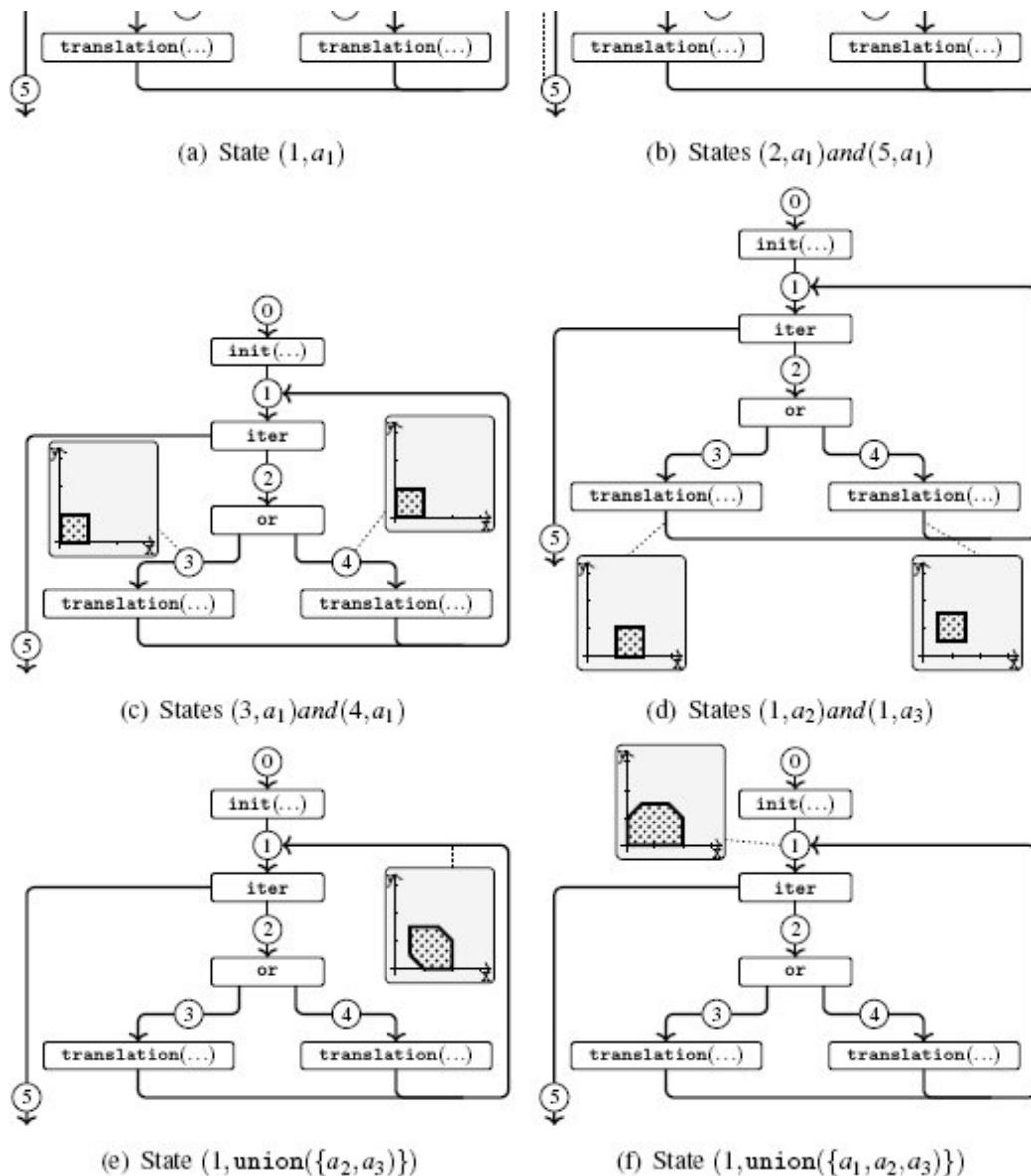


Figure 2.24

Abstract transition snapshots in the graph view of the program

A terminating analysis analysis_T should use a widening operation in place of the union_T operation:

$$\text{analysis}_T(p, I) = \begin{cases} C \leftarrow \{I\} \\ \text{repeat} \\ \quad R \leftarrow C \\ \quad C \leftarrow \text{widen}_T(C, \text{Step}^\sharp(C)) \\ \text{until } \text{inclusion}_T(C, R) \\ \text{return } R \end{cases}$$

The widen_T operator ensures the termination of the sequence of iterations. It makes sure the number of collected constraints for abstract pre-conditions will always decrease.

The widen_T function is identical to the union_T operation except that at the **iter** statements we use the widen operator in place of the union operator. This is because the **iter** statement is the only place where iteration happens during program execution. At other statements, we use the union operator as before:

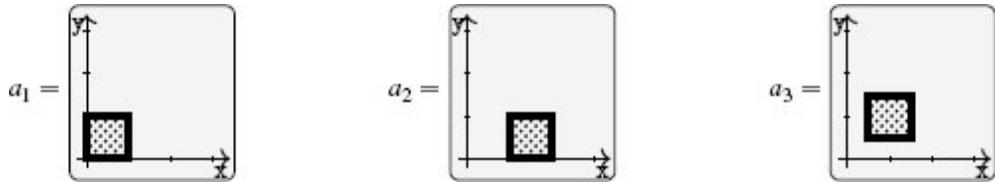
$$\text{widen}_T(C, C') = \begin{cases} X \leftarrow \{\} \\ \text{for each label } l \text{ in } C \cup C' \\ \quad S_l \leftarrow \{a \mid (l, a) \in C\} \\ \quad S'_l \leftarrow \{a \mid (l, a) \in C'\} \\ \quad T_l \leftarrow \text{if the statement at } l \text{ is } \textbf{iter} \\ \quad \quad \quad \text{then widen}(\text{union}(S_l), \text{union}(S'_l)) \\ \quad \quad \quad \text{else union}(S_l \cup S'_l) \\ \quad X \leftarrow X \cup \{(l, T_l)\} \\ \text{return } X \end{cases}$$

Note that, as opposed to the union operator, the widen operator is sensitive to the order of its arguments. The $\text{widen}(a, a')$ extrapolates a by a' as defined in section 2.3.4. When either argument is **false** (the abstract pre-condition for the empty set of points in the two-dimensional plane), the widen operation simply returns the other argument.

Example 2.19 (Abstract transitions with widening) Consider again the example program in figure 2.20, and suppose we use the convex polyhedra abstractions for sets of points. The above widening analysis algorithm analysis_T stores the following iterates C_i in the variable C after i iterations as before but using widen_T in place of union_T :

$$\begin{aligned}
&\text{after 0 iteration, } C_0 \stackrel{\text{let}}{=} \{I\} \\
&\text{after 1 iteration, } C_1 \stackrel{\text{let}}{=} \text{widen}_T(C_0, \{(1, a_1)\}) \\
&\text{after 2 iterations, } C_2 \stackrel{\text{let}}{=} \text{widen}_T(C_1, \{(2, a_1), (5, a_1)\}) \\
&\text{after 3 iterations, } C_3 \stackrel{\text{let}}{=} \text{widen}_T(C_2, \{(3, a_1), (4, a_1)\}) \\
&\text{after 4 iterations, } C_4 \stackrel{\text{let}}{=} \text{widen}_T(C_3, \{(1, a_2), (1, a_3)\}) \\
&\vdots \quad \vdots
\end{aligned}$$

where



The widening operation becomes effective at iteration 4 (C_4), when the algorithm brings the effect of the loop body back to the loop head (statement label 1). Figure 2.24(e) shows the abstract state produced when the two results from the **or** statement in the loop body are unioned ($\text{union}(\{a_2, a_3\})$). The algorithm brings this result to the loop head and widens it with the old precondition (a_1). In the algorithm, this widening operation

$$\text{widen}(a_1, \text{union}(\{a_2, a_3\}))$$

at the loop head happens during the computation of C_4 at iteration 4:

$$\text{widen}_T(C_3, \{(1, a_2), (1, a_3)\}).$$

The result corresponds to all the points such that $x \geq 0$ and $y \geq 0$, as shown in figure 2.25(b). It is a rather coarse over-approximation of the actual results, which are shown in figure 2.25(a).

The analysis accuracy can be improved by the “loop-unrolling” technique discussed in example 2.14 (section 2.3.4). This technique rewrites a loop “**iter** {b}” into “{}**or**{b};**iter** {b}” before the analysis. The analysis of the unrolled, first iteration (“{}**or**{b}”) will bring the unioned result to the subsequent loop head. For our example program the analysis result right after the unrolled first iteration is shown in Figure 2.24(f). Widening it at the subsequent loop head with the analysis result of the loop body will generate the result shown in figure 2.25(c). This result is still an over-approximation of the reality, yet it is more accurate than the result shown in figure 2.25(b).

2.5 Core Principles of a Static Analysis

The previous sections sketch the design of static analyses in the context of a simplistic graphical language. First, in section 2.1 we selected semantic properties of interest and formalized the semantics of programs, with respect to which these properties should be proved. Then in section 2.2 we showed how to define abstractions of the standard semantics of programs. Last, in

sections 2.3 and 2.4, we presented two static analyses for this graphical language. In fact, both analyses were derived from a presentation of the semantics of programs (one in compositional style and one in transitional style).

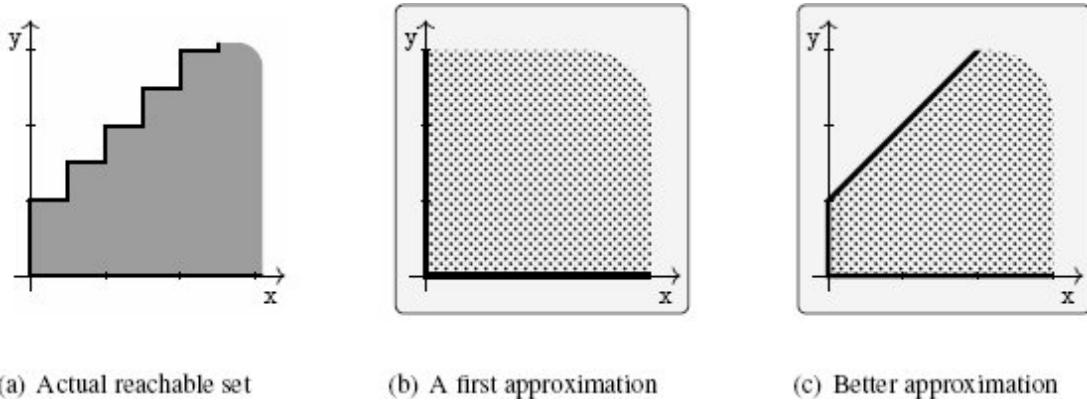


Figure 2.25

Static analysis results

This three-stage approach is actually general and has many fundamental and practical advantages, both for designing and for using static analysis tools.

Indeed, let us first recall the role of each stage:

1. **Selection of the semantics and properties of interest:** This stage is critical as it fixes the goal of the analysis. It describes the behaviors of programs and the properties that need to be verified. This description is often formal, even though we did it with prose in this chapter.
2. **Choice of the abstraction:** The abstraction describes the properties that are supposed to be manipulated by the analysis. These properties should be strong enough to express the properties of interest and all the invariants required to infer these properties.
3. **Derivation of the analysis algorithms from the semantics and from the abstraction:** The analysis algorithms follow from the choices made in the first two phases for the semantics and for the abstraction. In the two analyses presented in this chapter, we have

observed that the analysis closely follows the semantics. For instance, the compositional analysis (section 2.3) follows steps similar to those of a basic program interpreter, in the same order, but using abstract domain predicates instead of regular states.

From the static analysis point of view, this approach puts the choice of the reference semantics and property of interest at the forefront of the design process, as it should be, since this semantics and property define the actual goal of the analysis. It also addresses the selection of the predicates to use before the design of the algorithms to compute these predicates, although it does not preclude revising these choices after testing the analysis, as discussed at the end of this section.

As observed in sections 2.3 and 2.4, most of the choices related to the analysis algorithms are dictated by the abstraction and by the way programs get evaluated according to the concrete semantics. Therefore, this construction also allows justifying the soundness of the analysis step-by-step; indeed, whenever we defined the way a program construction should be handled by the analysis, we ensured that the analysis does not forget any program behavior, according to the abstraction. Thus, the mathematical proof of soundness follows the design of the analysis closely. We discuss this more in chapter 3.

Similarly, this approach also allows tying the analysis and the “standard” semantics of programs. It is actually possible to follow the same process when implementing a static analyzer, as we show in chapter 7.

Lastly, this methodology also simplifies the troubleshooting of the analysis when it falls short, either in terms of precision (ability to compute strong invariants and achieve the proof of the property of interest) or in terms of scalability (ability to cope with input programs that are large enough). In particular, let us consider the case where the analysis fails to prove the property of interest. Then after investigating the analysis results, the user can diagnose which step “went wrong”:

- The first point to check is that the base semantics allows expressing all the steps needed to prove the property of interest and that the abstraction preserves them; indeed, if the abstraction throws important information away, there is no hope that the analysis algorithms will infer predicates that the abstraction cannot capture and will recover from the loss of precision.

- When the semantics and abstraction are strong enough, the imprecisions stem from the analysis algorithms, and one needs to identify which abstract operation (for instance, the computation of the abstract post-condition for some basic operations in the language or the computation of an over-approximation for union) discards important information, which causes the analysis to fail.

When the analysis tool can be parameterized, the user can often remedy such issues by choosing settings carefully. We discuss this point in more depth in chapter 6.

¹ This chapter has been partially inspired by graphical descriptions of the notion of abstraction, such as the one presented in http://web.mit.edu/16.399/www/lecture_13-abstraction1/Cousot/MIT_2005_Course_13_4-1.pdf, even though this chapter focuses on different aspects of static analysis, transfer functions, and abstract iterations.

3 A General Static Analysis Framework Based on a Compositional Semantics

Goal of This Chapter In this chapter, we provide a formal (yet as lightweight as possible) introduction to the construction of a static analysis framework to compute program invariants by abstract interpretation. This framework is general and can be instantiated with different abstractions. We consider a basic imperative programming language that operates over purely numerical states, and we show how to construct a static analysis step-by-step, starting from the concrete semantics. The intuition conveyed in chapter 2 supports most of the contents of this chapter.

Recommended Reading: [S], [D], [U] We recommend this chapter to all readers since it defines the core concepts of static analysis and is fundamental to the understanding of most of the following chapters. Readers less interested in the foundations may skip some parts of the analysis design, whereas readers who would like to fully understand how static analyses achieve sound results may want to read the proofs supplied in appendix B. Moreover, readers interested in implementation may also combine the reading of this chapter with that of chapter 7 (in fact, both chapters may be read side by side, as chapter 7 considers exactly the same language as in this chapter).

Chapter Outline: Recipe for the Construction of an Abstract Interpreter The structure of this chapter follows the general recipe for the design of an abstract interpreter:

1. In section 3.1, we fix the target language and its concrete semantics.
2. In section 3.2, we select an abstraction that allows describing a set of properties that the analysis may reason about, and we fix their representation.
3. In section 3.3, we derive the abstract semantics of programs from their concrete semantics fixed in section 3.1 and from the abstraction chosen in section 3.2.

Finally, in section 3.4, we summarize the main assumptions and steps toward the design of the abstract interpreter.

3.1 Semantics

Before we can define a family of static analyses, we need to fix a language and its semantics.

3.1.1 A Simple Programming Language

To illustrate the concepts of static analysis, we do not require a language with a very large set of features. Thus, we fix a basic imperative language with a limited set of expressions, conditions, and statements.

The syntax is defined in the grammar shown in figure 3.1. We assume that a set of scalar values V (which define the constants in the language) and a finite set X of variables (defined by their name) are fixed. All variables have scalar type. We denote the set of Boolean values by $B = \{\text{true}, \text{false}\}$. Scalar expressions include constant expressions, variables, and binary operations (e.g., arithmetic operations) that are applied to pairs of expressions and evaluate into scalar values. Boolean expressions describe comparisons of variables with constants and evaluate them into Boolean values. Commands include a skip statement (which does nothing), sequences of commands, assignments, inputs of non-deterministic values (e.g., these values may be typed in by the user, or read on the disk or from the network, during the execution), conditional statements, and loop statements.

Even though this language is rather contrived, it allows expressing a large set of imperative programs. As such, it is enough to demonstrate the analysis of most basic constructions of programming languages, such as updates of scalar variables, conditions, and loops. The analysis of a language including more advanced programming constructions (functions, pointers, rich data types, etc.) is discussed in chapter 8.

3.1.2 Concrete Semantics

Following the methodology shown in chapter 2, we define the *semantics* of the programs that we intend to study before we formalize their static analysis. This semantics (also called *concrete semantics* in the rest of this chapter) formally defines the behavior of programs, and serves as a reference to design and prove the soundness of the static analysis that we study later in the

chapter. Therefore, it should be rich enough to express not only the target properties that we intend static analysis to compute but also all the logical facts that would be required to prove the property of interest. Indeed, a concrete semantics that fails to provide this would not allow designing and proving correct a static analysis that is sufficiently precise to establish the property of interest.

Many forms of semantics have been proposed, and some are more expressive than others. For instance, *trace semantics* describes program executions as sequences of program states, whereas *denotational semantics* describes only input-output relations, and ignores the intermediate steps. Some other forms of semantics only describe the sets of *reachable states*. Before we can select which form of semantics to use, we discuss the family of properties of interest.

| | |
|-------------------------------------|--|
| $n \in \mathbb{V}$ | scalar values |
| $x \in \mathbb{X}$ | program variables |
| $\odot ::= + - * \dots$ | binary operators |
| $\otimes ::= < \leq == \dots$ | comparison operators |
| $E ::=$ | scalar expressions |
| n | scalar constant |
| x | variable |
| $E \odot E$ | binary operation |
| $B ::=$ | Boolean expressions |
| $x \otimes n$ | comparison of a variable with a constant |
| $C ::=$ | commands |
| skip | command that “does nothing” |
| $C; C$ | sequence of commands |
| $x := E$ | assignment command |
| input(x) | command reading of a value |
| if(B){C}else{C} | conditional command |
| while(B){C} | loop command |
| $P ::= C$ | program |

Figure 3.1

Syntax of a simple imperative language

Family of Properties of Interest As in chapter 2, we focus on *reachability* properties, namely, properties that express program executions may reach only a predefined set of states R . Equivalently, such properties state that no execution should reach any state that is not in R , so the states that do not belong to R should be unreachable. For instance, the absence of run-time errors is a reachability property since it states that no execution should reach an error state. The verification of user assertions is another common case (in that case, the property expresses that no execution should reach an assertion point, yet not meet the assertion condition). More generally, set reachability properties form a subset of the class of safety properties defined in section

1.3.1. We address other classes of semantic properties (more general safety properties, liveness, and hyperproperties) in chapter 9.

A variant of reachability focuses on the final states of a program: it expresses that the final states of a program should satisfy a given *post-condition* (condition over final states), under the assumption that executions start from initial states that meet a given *pre-condition*. As an example of this family of properties, we can consider the verification that a program that should compute an absolute value always returns a non-negative number. To establish that a pre-condition entails a post-condition, we simply need a semantics that describes the relation between input states and final states; therefore, we shall use a semantics of that form.

The verification of reachability properties and of post-conditions are very strongly related. Clearly, post-conditions express a particular case of reachability property (i.e., that we should not reach a final state that does not meet the post-condition). On the other hand, semantic analyses that attempt at verifying post-conditions need to cover all reachable states and hence do the same work as required for proving more general reachability properties. Therefore, and since the focus of this chapter is on the definition of an abstract interpreter, we consider the computation of abstract post-conditions first and discuss the verification of general reachability properties in a second step.

An Input-Output Semantics In the following paragraphs, we set up an input-output semantics, which characterizes the effect of executing a program using a mathematical function that maps input states into corresponding output states. For instance, in the case of a program that computes an absolute value, this function will map a state where the argument is -5 into a state where the result is 5 . We observe that one input state may yield several output states, due to the non-deterministic execution of **input** commands: when running the program **input**(x) from any input state, we may observe infinitely many output states (one per value in V). Therefore, the actual form of the semantics of a command is a function that returns a set of output states. For the sake of compositionality, we also let it input a set of states. We obtain the following definition:

$$\text{program semantics} : \text{set of input states} \rightarrow \text{set of output states}$$

Such a semantics defines a very basic form of *denotational* semantics.

Intuitively, this semantics behaves like an *interpreter*. An interpreter is a program that inputs a program and an input state and runs the program so as to compute an output state. This definition is very close to that of the aforementioned semantics: the main difference is that an interpreter inputs a single state and returns a single state (that is partly chosen in a non-deterministic manner, in the case of the **input** command). Besides this point, the interpreter essentially implements the input-output semantics.

An important characteristic of the input-output semantics is that it is *compositional*, which means that the semantics of a command composed of several sub-commands (e.g., a sequence or a conditional command) can be defined simply by composing the semantics of the sub-commands. The analysis will inherit many characteristics from the semantics it is based on, including this one. This kind of semantics fits also very well the verification of post-conditions and can be easily instrumented to compute all reachable states.

Memory States In general a *program state* (or, for short, a *state*) should capture the configuration of the computer at a given point of the execution of a program. A state should thus describe the contents of the memory and the value of the “program counter” or next command to be executed. Therefore, it should comprise a *memory state* and a *control state* (the current location in the program).

In this chapter, we define an input-output semantics for programs, so control states can safely be ignored. Indeed, an input state (resp., output state) is fully determined by the memory state since its control state is known. As a consequence, in the following, a state will simply be defined as a memory state.

In our simple language, memory states can be described very simply. Indeed, our simple language features only a fixed set of variables, which all have the same type (it does not feature heap allocated regions, structures with multiple fields, and values of different sizes). Therefore, a memory state boils down to a function m from the (fixed, finite) set of variables X into the set of values V . Thus, the set of memory states M is defined by:

$$M = X \longrightarrow V$$

We write explicit definitions of functions that have a finite domain between braces of the form $\{\cdot\}$; for instance, if $X = \{x,y\}$, we write $\{x \mapsto 7, y \mapsto 3\}$

for the memory state that maps x into 7 and y into 3.

In the following paragraphs, we define the way each element of the language gets evaluated. Indeed, to describe the execution of commands, we need to first explain how scalar and Boolean expressions are evaluated.

Semantics of Scalar Expressions The evaluation of a scalar expression produces a scalar value. In the case of a constant n , the result is simply n . In the case of a variable x , the result is obtained by reading the content of variable x in the current memory state. Last, the result of the evaluation of an expression composed of an operator applied to two sub-expressions is obtained by evaluating the sub-expressions and applying to their results the mathematical function described by the operator. We let f_{\odot} denote the function associated to operator \odot .

The formalization of this semantics proceeds by induction over the syntax of expressions and is shown in figure 3.2. We let $\llbracket E \rrbracket(m)$ denote the semantics of expression E , evaluated in the memory state m , so that $\llbracket E \rrbracket$ is a function from memory states to scalar values.

Semantics of Boolean Expressions The case of Boolean expressions is very similar to the semantics of scalar expressions, with the only difference that it returns a Boolean value. We denote the semantics of a Boolean expression B by $\llbracket B \rrbracket$ and define it as a function that inputs a memory state and returns a Boolean value. Its definition is given in figure 3.3. We denote the function associated to a comparison operator \ominus by f_{\ominus} .

$$\begin{aligned}\llbracket E \rrbracket : M &\longrightarrow V \\ \llbracket n \rrbracket(m) &= n \\ \llbracket x \rrbracket(m) &= m(x) \\ \llbracket E_0 \odot E_1 \rrbracket(m) &= f_{\odot}(\llbracket E_0 \rrbracket(m), \llbracket E_1 \rrbracket(m))\end{aligned}$$

Figure 3.2

Semantics of scalar expressions

$$\llbracket B \rrbracket : \mathbb{M} \longrightarrow \mathbb{B}$$

$$\llbracket x \otimes n \rrbracket(m) = f_\otimes(m(x), n)$$

Figure 3.3

Semantics of Boolean expressions

Semantics of Commands The semantics of a command C is a function noted $\llbracket C \rrbracket_P$ that maps a set of input states into a set of output states, observed *after* the command. This choice entails that non-terminating executions are not observed at this point; indeed, if an execution of a command C does not terminate, it does not produce any output state. We denote the power set of memory states by $\wp(\mathbb{M})$, and we write M for an element of $\wp(\mathbb{M})$. The semantics $\llbracket C \rrbracket_P$ is shown in figure 3.4.

The semantics of the skip command is the identity function. The semantics of a sequence of commands is the composition of the semantics of each command. The evaluation of an assignment $x := E$; updates the value of x in the memory states with the result produced by the evaluation of E . The evaluation of **input**(x) replaces the value of variable x with any possible scalar value.

The evaluation of a conditional command is determined by the result of the condition. Since our semantics is defined over sets of input states, it should simply filter the states for which the condition evaluates to **true** and evaluate the corresponding branch, do the same for the states for which the condition evaluates to **false**, and return the union of the two results. The condition expression effectively filters out memory states; thus, for each Boolean expression B , we define a “filtering” function \mathcal{F}_B :

$$\mathcal{F}_B(M) = \{m \in M \mid \llbracket B \rrbracket(m) = \mathbf{true}\}$$

We write $\neg B$ for the negation of Boolean expression B . For instance $\neg(x \leq 3)$ is the Boolean expression $x > 3$. Thus, the set of states that enter the **true** (resp., **false**) branch is defined by $\mathcal{F}_B(M)$ (resp., $\mathcal{F}_{\neg B}(M)$).

The case of loops is more complex and interesting due to unbounded executions. To determine the set of output states produced by a loop

command, we can simply partition executions based on the number of iterations they spend inside the loop before they exit; thus, the set of output states is the infinite union of a family of sets M_0, M_1, M_2, \dots , where M_i denotes the states produced by program executions that went through the loop body exactly i times. Intuitively, each of these sets can be described using the same technique as if we were considering the output of a sequence of conditional commands (where the condition evaluates to **true** i times and to **false** for the last test). Thus, a state is in M_i if and only if it is in M_i defined as follows:

$$\begin{aligned}
\llbracket C \rrbracket_{\mathcal{P}} : \wp(\mathbb{M}) &\longrightarrow \wp(\mathbb{M}) \\
\llbracket \text{skip} \rrbracket_{\mathcal{P}}(M) &= M \\
\llbracket C_0; C_1 \rrbracket_{\mathcal{P}}(M) &= \llbracket C_1 \rrbracket_{\mathcal{P}}(\llbracket C_0 \rrbracket_{\mathcal{P}}(M)) \\
\llbracket x := E \rrbracket_{\mathcal{P}}(M) &= \{m[x \mapsto \llbracket E \rrbracket(m)] \mid m \in M\} \\
\llbracket \text{input}(x) \rrbracket_{\mathcal{P}}(M) &= \{m[x \mapsto n] \mid m \in M, n \in \mathbb{V}\} \\
\llbracket \text{if}(B)\{C_0\} \text{else}\{C_1\} \rrbracket_{\mathcal{P}}(M) &= \llbracket C_0 \rrbracket_{\mathcal{P}}(\mathcal{F}_B(M)) \cup \llbracket C_1 \rrbracket_{\mathcal{P}}(\mathcal{F}_{\neg B}(M)) \\
\llbracket \text{while}(B)\{C\} \rrbracket_{\mathcal{P}}(M) &= \mathcal{F}_{\neg B} \left(\bigcup_{i \geq 0} (\llbracket C \rrbracket_{\mathcal{P}} \circ \mathcal{F}_B)^i(M) \right)
\end{aligned}$$

Figure 3.4

Semantics of commands

$$M_i = \mathcal{F}_{\neg B} \left((\llbracket C \rrbracket_{\mathcal{P}} \circ \mathcal{F}_B)^i(M) \right)$$

As a result, the set of output states of the loop is computed by:

$$\bigcup_{i \geq 0} M_i = \bigcup_{i \geq 0} \mathcal{F}_{\neg B} \left((\llbracket C \rrbracket_{\mathcal{P}} \circ \mathcal{F}_B)^i(M) \right)$$

Moreover, \mathcal{F}_B commutes with the union; thus,

$$\bigcup_{i \geq 0} M_i = \mathcal{F}_{\neg B} \left(\bigcup_{i \geq 0} (\llbracket C \rrbracket_{\mathcal{P}} \circ \mathcal{F}_B)^i(M) \right)$$

Remark 3.1 (Alternate definition) As a side remark (which can be skipped in a first reading), the above definition of the semantics of loops can also be written using a least fixpoint, using Kleene's fixpoint theorem (theorem A.1) $\mathcal{F}_{\neg B}(\text{lfp}_M F)$, where $F : M' \mapsto M \cup \llbracket C \rrbracket_{\mathcal{P}} \circ \mathcal{F}_B(M')$. (The application of the theorem requires verifying the continuity of F .)

To conclude, the semantics of a command is defined by induction over the syntax, as shown in figure 3.4.

3.2 Abstractions

3.2.1 The Concept of Abstraction

In the following, we make the definition of abstract domains and concretization functions more formal than definitions 2.1 and 2.2.

Intuitions Gathered from the Previous Chapter In chapter 2, we noted that an abstract domain should provide:

- a set of abstract elements that stand for logical properties, to be used by the analysis;
- a compact and efficient data structure to represent abstract elements;
- a relation that fixes the meaning of each abstract element in concrete terms; and
- analysis algorithms, to carry out the computation of abstract post-conditions, union, widening, and so forth.

We also noted that logical comparison among abstract elements plays a crucial role to determine which abstract elements are better than others in terms of precision.

Additionally, chapter 2 presents several examples of abstractions based on signs, intervals, and convex polyhedra. These examples of abstractions are all meaningful for the language introduced in section 3.1; thus, we expect to be able to apply them to this language as well, and that the analyses set up this way also behave like the intuitive analysis of chapter 2.

Concrete Elements, Abstract Elements and Abstraction Relation In the following, we distinguish carefully the abstract domain that is used for the analysis of programs and the domain where the semantics of program is defined. We apply the “abstract” qualifier to the former and the “concrete” qualifier to the latter.

Definition 3.1 (Concrete domain) *We call concrete domain a set C used to describe concrete behaviors, with an order relation \subseteq that compares program behaviors in the logical point of view: if x, y are elements of the concrete domain, then $x \subseteq y$ holds whenever behavior x implies behavior y (i.e., x expresses a stronger property than y).*

The definition of order relations is recalled in appendix A.5. In this book, we will always consider concrete domains that are power sets, so that the order relation over the concrete domain will always be the inclusion relation (which is why we write \subseteq as for set inclusion).

Example 3.1 (Concrete domain) *For instance, to study sets of reachable states, we should choose $C = \wp(M)$ as the concrete domain.*

An abstract domain should also come with a way to compare elements, and it should provide some way to interpret them with respect to the concrete level. Intuitively, this notion of “meaning” should say when a concrete element c can be described by an abstract element a or, equivalently, when a concrete element c satisfies the logical properties expressed by an abstract element a .

In the following, we write $c \models a$ when this relation holds, that is, when a describes c .

Definition 3.2 (Abstract domain and abstraction relation) *We call abstract domain [26] a pair made of a set A and an ordering relation \sqsubseteq over that set.*

Given a concrete domain (C, \subseteq) , an abstraction is defined by an abstract domain (A, \sqsubseteq) and an abstraction relation $(\models) \subseteq C \times A$, such that,

- *for all $c \in C, a_0, a_1 \in A$, if $c \models a_0$ and $a_0 \sqsubseteq a_1$, then $c \models a_1$; and*
- *for all $c_0, c_1 \in C, a \in A$, if $c_0 \subseteq c_1$ and $c_1 \models a$, then $c_0 \models a$.*

The two properties that definition 3.2 requires \models to satisfy capture the fact that both order relations (in the concrete and in the abstract domains) should be compatible with the same interpretation as logical implication. For instance, let us consider the first one in detail: it asserts that if a concrete element c satisfies the property described by the abstract element a_0 , and if the abstract element a_1 expresses a weaker property than that expressed by a_0 , then c should also satisfy the abstract property a_1 . The second item asserts the same thing for the concrete ordering.

Example 3.2 (Abstraction) *We consider the concrete domain of example 3.1 and the abstraction that retain range information for each variable, following the principles set by definition 2.3 in chapter 2. We characterize \models based on a couple of examples. We assume that there are two variables x, y , and we consider the following elements of the concrete domain (which are sets of memory states):*

$$\begin{aligned} M_0 &= \{m \in \mathbb{M} \mid 0 \leq m(x) \leq m(y) \leq 8\} \\ M_1 &= \{m \in \mathbb{M} \mid 0 \leq m(x)\} \end{aligned}$$

Furthermore, we assume that the abstract element $M^\#$ over-approximates the value of x by the interval $[0, 10]$ and the value of y by interval $[0, 80]$. Then,

- $M_0 \models M^\#$ since any memory in M_0 satisfies the range constraints in $M^\#$; and
- $M_1 \not\models M^\#$ since there exist memory states in M_0 that cannot be described by $M^\#$, such as the memory state that maps x to 100 and y to 50.

A relation \models is not always the most convenient way to relate concrete properties and abstract elements, and it is often preferable to look at functions that

- map an abstract element into the concrete behaviors it describes, or
- turn concrete program behaviors into an abstract property that describes them accurately.

Therefore, in the following we study such functions and their basic properties.

Concretization Function The most common way to describe the abstraction relation \models proceeds by defining a function that maps each abstract element to the *largest* set of program behaviors that satisfy it.

Definition 3.3 (Concretization function) A concretization function (or, for short, concretization) is a function $\gamma : A \rightarrow C$ such that, for any abstract element a , $\gamma(a)$ satisfies a (i.e., $\gamma(a) \models a$) and $\gamma(a)$ is the maximum element of C that satisfies a .

A concretization function fully describes the abstraction relation, because of the following equivalence:

$$\forall c \in C, a \in A, \quad c \models a \Leftrightarrow c \subseteq \gamma(a)$$

A concretization function is also monotone. From the static analysis point of view, this remark is important since it means that an abstract element that is bigger for \sqsubseteq accounts for a larger set of concrete states thus is less precise.

Example 3.3 (Concretization function) We use the same notations as in example 3.2. Then the concretization of $M^\#$ is the set of all the memory states such that the value of x ranges between 0 and 10 and such that the value of y lies between 0 and 80. As expected, this set includes M_0 but not M_1 . We note there are memory states in $\gamma(M^\#)$ that are not in M_0 , such as the memory state m defined by $m(x) = 8, m(y) = 3$.

Abstraction Function and Galois Connection In some cases, we can also describe \models with a function that maps each concrete element into the *smallest* abstract element that describes it. More intuitively, this function maps each concrete element to its most precise abstract property.

Definition 3.4 (Abstraction function) Let c be a concrete element. We say that c has a best abstraction if and only if there exists an abstract element a such that (1) a is an abstraction of c and (2) any other abstraction of c is greater than a . If it exists, this element is unique and called the best abstraction of c .

An abstraction function (or, for short, abstraction) is a function $\alpha: C \rightarrow A$ that maps each concrete element to its best abstraction.

An abstraction function is essentially the dual of a concretization function. Thus, it also fully defines the underlying abstraction relation. An abstraction function is also monotone.

Example 3.4 (Abstraction function) We use the same notations as in examples 3.2 and 3.3. Then the best abstraction of M_0 describes both x and y with range $[0,8]$. It is smaller (thus more precise) than $M^\#$.

The existence of a best abstraction function is not guaranteed in general, and some important abstractions have no best abstraction function. As an example, we observed in chapter 2 that the abstract domain based on convex polyhedra does not have a best abstraction function (definition 2.5). Another, simpler example is provided later in example 3.6, and other cases will also be encountered in the rest of the book.

While we can design abstraction relations such that no concretization function can be defined, none will arise in this book. When an abstraction relation defines both a concretization function and an abstraction function, they are thus tightly related to each other. Thus, they form what we call a *Galois connection*.

Definition 3.5 (Galois connection) A Galois connection [26] is a pair made of a concretization function γ and an abstraction function α such that

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

We write such a pair as follows:

$$(C, \sqsubseteq) \xrightleftharpoons[\alpha]{\gamma} (A, \sqsubseteq)$$

Galois connections have numerous interesting algebraic properties, many of which are presented in [28]. We list below the main properties that one should know. Under the assumption that (C, \sqsubseteq) and (A, \sqsubseteq) form a Galois connection with abstraction function α and concretization function γ as defined above, then

- α and γ are monotone functions, which means that they map logically comparable inputs into logically comparable outputs;
- $\forall c \in C, c \sqsubseteq \gamma(\alpha(c))$ (or, more concisely, using the pointwise ordering of functions and the identity function id , $\text{id} \sqsubseteq \gamma \circ \alpha$), which means that applying the abstraction function and concretizing

the result back yield a less precise result (or, equivalently, a conservative approximation); and

- $\forall a \in A, \alpha(\gamma(a)) \sqsubseteq a$ (or, more concisely, $\alpha \circ \gamma \sqsubseteq \text{id}$), which means that concretizing an abstract element and then abstracting the result back refines the information available in the initial abstract element; this refinement is known as a *reduction* (it is studied more in detail in section 5.1.2).

The proofs for these properties are provided in appendix B.1.

Comparing Abstraction Formalizations In the case of a Galois connection, both the abstraction and the concretization function can be used interchangeably. In particular, since the abstraction function computes the best way to abstract any concrete behavior, it is very helpful to set the ideal analysis goal, namely, to compute the best abstraction of the semantics of the program being analyzed.

On the other hand, when no best abstraction function can be defined, some concrete behaviors do not have a best approximation in terms of precision. This means that, for some analysis operation, we may not be able to even specify the best result that could be produced; then the analysis will inevitably have to make choices among several possible answers that are logically incomparable.

3.2.2 Non-Relational Abstraction

In chapter 2, we introduced *non-relational* abstractions as abstractions that forget about all relations among program variables. In this section, we formalize the most common form of non-relational abstraction, where all variables are treated independently, with a similar abstraction of their values. Intuitively, this abstraction proceeds in two steps:

1. For each variable, it collects the values that this variable may take across a set of states.
2. Then it over-approximates each of these sets of values with one abstract element per variable; each of these abstract elements consists of numerical constraints over a single variable.

This second step relies on a *value abstraction*.

Definition 3.6 (Value abstraction) A value abstraction is an abstraction of $(\wp(V), \sqsubseteq)$.

Signs and intervals constraints (as used in chapter 2) define value abstractions.

Example 3.5 (Signs) The elements of the sign value abstract domain A_S are $[\geq 0]$, $[\leq 0]$, $[= 0]$, which describe the properties associated to their name; \top , which describes any set of values; and \perp , which describes only the empty set of values. Therefore, their concretizations are defined by

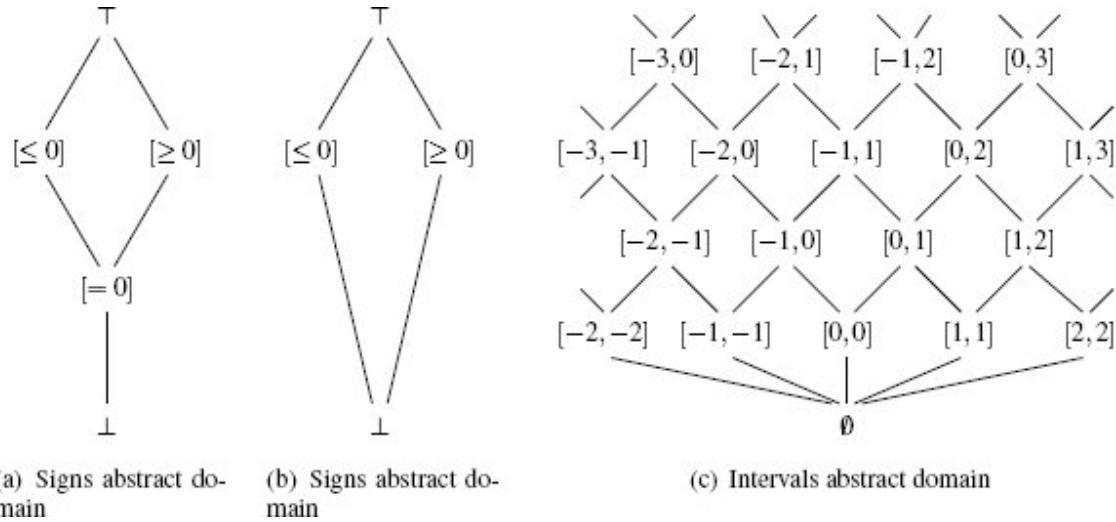


Figure 3.5

Value abstract domains

$$\gamma_S : \begin{array}{lcl} [\geq 0] & \mapsto & \{n \in \mathbb{V} \mid n \geq 0\} \\ [\leq 0] & \mapsto & \{n \in \mathbb{V} \mid n \leq 0\} \\ [= 0] & \mapsto & \{0\} \\ \top & \mapsto & \mathbb{V} \\ \perp & \mapsto & \emptyset \end{array}$$

This abstraction features a best abstraction function α_S : if V is a set of values, $\alpha_S(V)$ is the smallest element of the signs lattice that over-approximates it. Figure 3.5(a) shows the order relation in the abstract domain of signs using a Hasse diagram, which consists of a graph where nodes denote abstract elements and edges link neighbors in the ordering relation, with the convention that “smaller” abstract elements are lower than “bigger” abstract elements.

Example 3.6 (A variation on the lattice of signs, with no abstraction function) We can build many variations over the lattice of signs exposed in example 3.5 by adding elements to denote strict inequalities or to denote the predicate “not equal to zero,” or by removing some elements. For instance, figure 3.5(b) shows another abstract domain that retains only abstract elements \perp , $[\geq 0]$, $[\leq 0]$, and \top . This new abstraction defines the same concretization function as in example 3.5; however, it does not have a best abstraction function α_S . Indeed, let us consider the concrete set $\{0\}$; then $[\leq 0]$, $[\geq 0]$, and \top define valid abstractions of $\{0\}$. However, there exists no smaller abstract element that over-

approximates it since $[\leq 0]$ and $[\geq 0]$ are both incomparable (and smaller than \top). Thus, it would be impossible to define a best abstraction function α_S over $\{0\}$. The consequence in static analysis is that it is not possible in general to identify one element as the most precise (in other words, best sound) possible result. Provided the analysis designer and user are aware of this fact, it is not a serious limitation, however.

Example 3.7 (Intervals) The elements of the intervals value abstract domain [26] A_I are

- the element \perp , which represents the empty set of values, and
- the pairs (n_0, n_1) , where n_0 is either $-\infty$ or a value, n_1 is either $+\infty$ or a value, and $n_0 \leq n_1$.

For clarity, pairs will often be given using the classic interval notation (e.g., we will write $[8, +\infty)$ instead of $(8, +\infty)$). Therefore, their concretization is defined by

$$\gamma_{\mathcal{I}} : \begin{array}{rcl} \perp & \mapsto & \emptyset \\ [n_0, n_1] & \mapsto & \{n \in \mathbb{V} \mid n_0 \leq n \leq n_1\} \\ [n_0, +\infty) & \mapsto & \{n \in \mathbb{V} \mid n_0 \leq n\} \\ (-\infty, n_1] & \mapsto & \{n \in \mathbb{V} \mid n \leq n_1\} \\ (-\infty, +\infty) & \mapsto & \mathbb{V} \end{array}$$

The order relation \sqsubseteq_I over abstract elements is induced by the concretization and captures the interval inclusion. As in example 3.5, this abstraction relation also defines a best abstraction function α_I . The Hasse diagram of the abstract domain of intervals is shown in figure 3.5(c).

Example 3.8 (Congruences) While the instances of value abstractions presented so far all boil down to inequality constraints, we can cite relevant abstractions that do not follow this structure. A very useful example is the abstract domain of congruences [52], which describes sets of values using congruence relations. Abstract values are \perp and pairs of the form (n, p) where either $p = 0$ or $0 \leq n < p$. While \perp represents the empty set of values, a pair (n, p) stands for a set of values that are equal to n modulo p :

$$\gamma_{\mathcal{C}} : \begin{array}{rcl} \perp & \mapsto & \emptyset \\ (n, p) & \mapsto & \{n + kp \mid k \in \mathbb{Z}\} \end{array}$$

We let A_C denote the set of abstract elements, and let \sqsubseteq_C be the order relation defined by $a_0 \sqsubseteq_C a_1 \Leftrightarrow \gamma_C(a_0) \subseteq \gamma_C(a_1)$. This domain also has a best abstraction function α_C . Note that the set of all integers is described by the pair $(0, 1)$, and that a singleton $\{n\}$ is described by the pair $(n, 0)$.

As remarked above, a non-relational abstraction collects separately the values each variable may take and applies the value abstraction to each component.

Definition 3.7 (Non-relational abstraction) We assume that a value abstraction (A_V, \sqsubseteq_V) is given, with a concretization function $\gamma_V : A_V \rightarrow \wp(V)$, a least element \perp_V , and a greatest element \top_V . Then the non-relational abstraction based on it is defined by

- the set of abstract elements $A_N = X \rightarrow A_V$;
- the order relation \sqsubseteq_N defined by the pointwise extension of \sqsubseteq_V (which means that $M_0^\sharp \sqsubseteq_N M_1^\sharp$ if and only if $\forall x \in X, M_0^\sharp(x) \sqsubseteq_V M_1^\sharp(x)$); and

- the concretization function γ_N defined by

$$\gamma_N : M^\# \mapsto \{m \in M \mid \forall x \in X, m(x) \in \gamma_V(M^\#(x))\}$$

Intuitively, this abstraction treats each variable independently and applies the value abstraction to each variable separately from the others. The order relation is the pointwise extension of \sqsubseteq_V ; indeed, in the logical point of view, $M_0^\#$ describes a property that is stronger than that of $M_1^\#$ if and only if it does so for each variable. The machine representation of non-relational abstract values boils down to a data structure for tuples (with record types in small dimensions or arrays/functional arrays when the number of variables is high). When defining such an abstract element explicitly, we use the same notation $\{\cdot\}$ as before; for instance, if $X = \{x, y\}$, and if the A_V is the signs abstract domain, we write $\{x \mapsto [= 0], y \mapsto [\geq 0]\}$ for the non-relational abstract state that maps x into $[= 0]$ (i.e., it expresses that x is equal to zero) and y into $[\geq 0]$ (i.e., it expresses that x is positive).

The least element of the non-relational abstract domain is the function that maps each variable to the least element \perp_V of the value abstract domain:

$$\forall x \in X, \perp_N(x) = \perp_V$$

The greatest element \top_N can be defined similarly.

When the value abstraction also has an abstraction function α_V , the non-relational abstraction also has one that is defined as follows (note that it maps a set of states into a function from variables into elements of the value abstract domain):

$$\alpha_N : M \mapsto (x \in X) \mapsto \alpha_V(\{m(x) \mid m \in M\})$$

We remark that \perp_N is the best approximation of \emptyset .

Chapter 2 already introduced particular instances of this non-relational abstraction, in the case where X contains only two elements, so that a concrete memory is a point in the two-dimensional space, and an abstract element is a pair of value abstract elements (e.g., sign or interval predicate). Definition 3.7 extends this construction to the cases where the set of variables may have any size.

Example 3.9 (Non-relational abstraction) We assume that $X = \{x, y, z\}$, and we consider the memory states defined as follows:

$$\begin{aligned}
 m_0 : & \quad x \mapsto 25 \quad y \mapsto 7 \quad z \mapsto -12 \\
 m_1 : & \quad x \mapsto 28 \quad y \mapsto -7 \quad z \mapsto -11 \\
 m_2 : & \quad x \mapsto 20 \quad y \mapsto 0 \quad z \mapsto -10 \\
 m_3 : & \quad x \mapsto 35 \quad y \mapsto 8 \quad z \mapsto -9
 \end{aligned}$$

The best abstractions of $\{m_0, m_1, m_2, m_3\}$ can be defined as follows:

- With the signs abstraction:

$$M^\# : x \mapsto [\geq 0] \quad y \mapsto T \quad z \mapsto [\leq 0]$$

- With the intervals abstraction:

$$M^\# : x \mapsto [25, 35] \quad y \mapsto [-7, 8] \quad z \mapsto [-12, -9]$$

3.2.3 Relational Abstraction

We call *relational abstraction* any abstraction of sets of memory states that allows maintaining at least some constraints that bind several variables. Non-relational abstractions handle each variable separately from the others and thus cannot express precisely any constraints such as $x + y \leq 3$ or $x^2 - y \leq 0$. If we were trying to represent such constraints with non-relational abstractions, we would have to first approximate them in a very coarse manner, which would first drop any relationship between x and y .

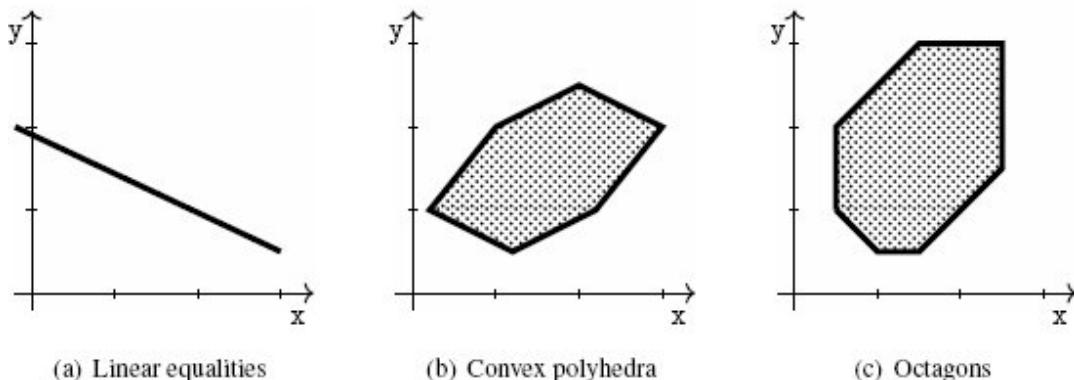


Figure 3.6

A few relational domains

Many relational abstract domains have been proposed in the literature. We give a few examples in the rest of this section, some of which are informally discussed in section 2.2.

We start with linear equalities [67].

Definition 3.8 (Linear equalities) *The elements of the abstract domain of linear equalities are the \perp value denoting the empty set and the conjunctions of linear equality constraints over the program variables to constrain sets of memory states.*

In the geometrical point of view, abstract elements boil down to affine spaces in V^N where N is the number of variables. For instance, in dimension 3 (i.e., if the program has three variables), this includes the empty set, points, lines, planes, and the whole space. The resulting abstraction features both a concretization and an abstraction function (the best abstraction of a set of points is the smallest enclosing affine space). The best abstraction of any set M of memory states is the smallest affine space that contains all the memories in M . An example is shown in figure 3.6(a).

As discussed in chapter 2, linear inequalities also provide a good basis to build an abstract domain.

Definition 3.9 (Convex polyhedra) *The elements of the abstract domain of linear inequalities (also called convex polyhedra abstract domain) [30] are the \perp value denoting the empty set and the conjunctions of linear inequality constraints over the program variables to constrain sets of memory states.*

In the geometrical point of view, abstract elements correspond to convex polyhedra of all dimensions in V^N , where N is the number of variables. An example is shown in figure 3.6(b).

The abstract domain of convex polyhedra defines a concretization function but features no best abstraction function, although certain concrete sets do have a best abstraction. The canonical example of a concrete set without a best abstraction is the disk (as discussed in section 2.2) since it is possible to refine any polyhedron enclosing a disk by adding one more face, thereby turning it into a strictly tighter enclosing polyhedron.

This example based on tighter and tighter abstractions of a disk also illustrates the main source of cost for this abstraction: even with a low number of variables, the number of inequalities (or faces in the geometrical point of view) is not bounded. In practice, it may grow exponentially with the number of variables.

To alleviate this issue, it is possible to restrict the set of inequality constraints that the abstract domain can express precisely. Due to the lesser expressiveness of the abstraction, the static analysis becomes less precise (it can infer fewer properties), but it may also benefit from a more compact representation and from faster algorithms; thus, the analysis may also become less costly. As an example, the following abstraction retains only constraints with at most two variables, and with coefficients $-1, 0$, and 1 .

Definition 3.10 (Octagons) *The elements of the abstract domain of octagons [82] are the \perp element denoting the empty set and the conjunctions of linear inequality constraints of the form $\pm x \pm y \leq c$ or $\pm x \leq c$.*

In the geometrical point of view, abstract elements correspond to “octagonal” shapes (e.g., in dimension 2, an abstract element corresponds to a convex polyhedron with at most eight faces that are either parallel to the axes or at a 45° angle). This abstraction has both abstraction and concretization functions. An example is shown in figure 3.6(c).

In general, the choice of an efficient computer representation for abstract domains that describe relational constraints is more difficult than in the case of non-relational abstract domains. Therefore, we refer the reader to other sources, e.g., [30, 82], and do not discuss this issue further in this book.

3.3 Computable Abstract Semantics

In this section, we formalize the static analysis algorithms for the basic language introduced in section 3.1. It follows similar principles as the analysis for a graphical language presented in chapter 2. We fully describe the analysis based on the non-relational abstract domain defined in section 3.2.2. We also detail the analysis modifications that would be required to use a relational domain (section 3.2.3) instead and show that these are minimal, in the same way as in chapter 2. The analysis takes the form of a mathematical function that inputs a program and an abstract pre-condition and computes an abstract post-condition that covers all the output states of the program when run from any state that satisfies the pre-condition. This setup is similar to that of section 2.3.

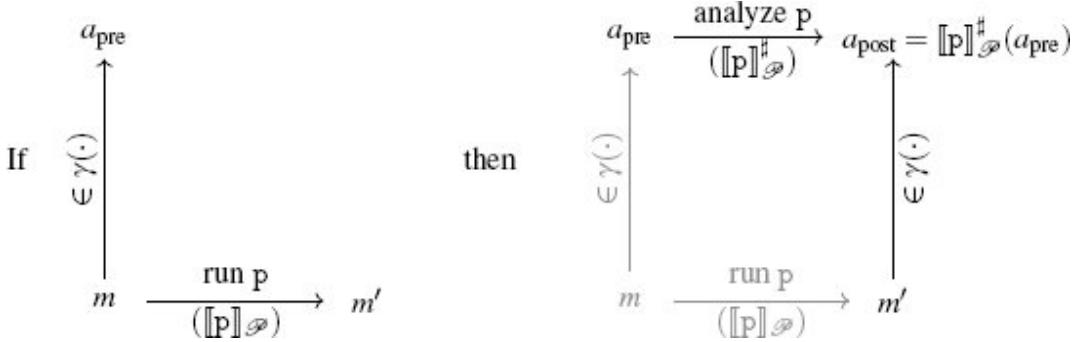


Figure 3.7

Sound static analysis

We let A denote the state abstract domain, and we write γ for the associated concretization function. We write A_V for the underlying value abstraction, and we write γ_V for its concretization. The design of the analysis is aimed at ensuring *soundness* in the sense of definition 2.6, which we recall in figure 3.7 in the present setup. In figure 3.7, we let $[\![p]\!]_{\phi}$ denote the static analysis function, which we also call *abstract semantics*. It states that, when (1) m is a concrete memory that can be described by the abstract pre-condition a_{pre} (i.e., $m \in \gamma(a_{pre})$) and (2) we can observe output memory state m' after running the program from m (i.e., $m' \in [\![p]\!]_{\phi}(\{m\})$), then m' can be described by the result of the analysis applied to p and a_{pre} (i.e., $m' \in \gamma([\![p]\!]_{\phi}^{\sharp}(a_{pre}))$).

While we use the concretization function γ , we could also use the best abstraction function α , if it exists. Then soundness statements are similar but expressed in terms of abstraction rather than concretization.

As in chapter 2, we are going to construct the definition of $[\![\cdot]\!]_{\phi}^{\sharp}$ by induction over the syntax. First, this will result in a definition of the analysis that is very close to the concrete semantics (likewise, it would also be very similar to an implementation shown in chapter 7). Moreover, this process allows for a straightforward step-by-step proof that the soundness property of figure 3.7 holds. Indeed, when considering any sort of command that includes some sub-commands (e.g., sequences, conditions, and loops), we will simply assume that the abstract semantics of the sub-commands has been defined and

build a definition of the abstract semantics of the command itself. Therefore, both the definition of the analysis and its proof of soundness proceed by induction over the syntax of programs; indeed, to analyze (resp., prove the analysis of) a program, we will simply rely on the analysis of (resp., on the proof of the analysis of) its components.

We start with a few easy cases before considering more complex commands.

Bottom Element For any command C , $\llbracket C \rrbracket_P(\emptyset) = \emptyset$ since the set of states reachable when running a program from an empty set of states is empty. Therefore,

$$\llbracket C \rrbracket_{\mathcal{P}}^{\sharp}(\perp) = \perp$$

Skip Commands The concrete semantics of the **skip** command returns its input unmodified; therefore, the following definition ensures soundness:

$$\llbracket \text{skip} \rrbracket_{\mathcal{P}}^{\sharp}(M^{\sharp}) = M^{\sharp}$$

Sequences of Commands The soundness property of figure 3.7 is stable under composition, and $\llbracket p_0; p_1 \rrbracket_P(M) = \llbracket p_1 \rrbracket_P(\llbracket p_0 \rrbracket_P(M))$; thus, the following definition ensures that we can prove soundness by induction over the syntax:

$$\llbracket C_0; C_1 \rrbracket_{\mathcal{P}}^{\sharp}(M^{\sharp}) = \llbracket C_1 \rrbracket_{\mathcal{P}}^{\sharp}(\llbracket C_0 \rrbracket_{\mathcal{P}}^{\sharp}(M^{\sharp}))$$

We can informally justify the soundness of this definition: to compute a sound post-condition for $C_0; C_1$, starting from a pre-condition M^{\sharp} , we can simply compute a sound post-condition for C_0 from this pre-condition M^{\sharp} and then use this post-condition as a pre-condition for C_1 and, finally, compute a sound post-condition for C_1 .

This principle generalizes to any composition of functions as follows.

Theorem 3.1 (Approximation of compositions) *Let $F_0, F_1 : \wp(M) \rightarrow \wp(M)$ be two monotone functions, and let $F_0^{\sharp}, F_1^{\sharp} : \mathbb{A} \rightarrow \mathbb{A}$ be two functions that over-approximate them, that is such that $F_0 \circ \gamma \subseteq \gamma \circ F_0^{\sharp}$ and $F_1 \circ \gamma \subseteq \gamma \circ F_1^{\sharp}$. Then $F_0 \circ F_1$ can be over-approximated by $F_0^{\sharp} \circ F_1^{\sharp}$.*

Indeed, if $M^{\sharp} \in \mathbb{A}$, then $F_1 \circ \gamma(M^{\sharp}) \subseteq \gamma \circ F_1^{\sharp}(M^{\sharp})$ (by the soundness assumption on F_1), so since F_0 is monotone, $F_0 \circ F_1 \circ \gamma(M^{\sharp}) \subseteq F_0 \circ \gamma \circ F_1^{\sharp}(M^{\sharp})$, and by the

soundness hypothesis on $F_0, F_0 \circ F_1 \circ \gamma(M^\sharp) \subseteq \gamma \circ F_0^\sharp \circ F_1^\sharp(M^\sharp)$. Theorem 3.1 is of fundamental interest since the concrete semantics heavily relies on the composition of functions, and this theorem means that we can decompose the over-approximation of a composition of operations into the composition of over-approximations of each operation.

We remark that the analysis functions for both skip commands and sequences of commands proceed in the same way as in chapter 2. Moreover, the definition of the analysis for these commands does not depend at all on the abstract domain.

3.3.1 Abstract Interpretation of Assignment

In the graphical language of chapter 2, state updates boil down to translation and rotation statements. We observed that the analysis of these statements boils down to the application of a similar transformation in the abstract level.

In the language considered in this chapter, updates to memory states are performed by assignment commands. However, the principles of chapter 2 still apply here: the analysis should update the abstract memory states so as to mimic any update that may occur in the concrete level.

To develop this idea, we will first present an algorithm for the abstract interpretation of expressions that characterizes the values that they may evaluate to, and then, we will show how abstract updates can be performed.

$$\begin{aligned} \llbracket E \rrbracket^\sharp : \mathbb{A} &\longrightarrow \mathbb{A}_\gamma \\ \llbracket n \rrbracket^\sharp(M^\sharp) &= \phi_\gamma(n) \\ \llbracket x \rrbracket^\sharp(M^\sharp) &= M^\sharp(x) \\ \llbracket E_0 \odot E_1 \rrbracket^\sharp(M^\sharp) &= f_\odot^\sharp(\llbracket E_0 \rrbracket^\sharp(M^\sharp), \llbracket E_1 \rrbracket^\sharp(M^\sharp)) \end{aligned}$$

Figure 3.8

Abstract semantics of expressions

Abstract Interpretation of Expressions The algorithm for the abstract interpretation of an expression should input an abstract pre-condition and return an abstraction of the values the expression may take. We denote the abstract interpretation of the expression E by $\llbracket E \rrbracket^\sharp$.

Since the semantics of scalar expressions is defined by induction over the syntax (figure 3.2), their abstract interpretation also proceeds by induction over the syntax. First, when the expression is a constant n , its abstract semantics should return any abstract element that over-approximates $\{n\}$. If the value abstraction has a best abstraction α_V , then $\alpha_V(\{n\})$ obviously provides such an over-approximation. Otherwise, we simply need to use a function $\phi_V : V \rightarrow A_V$ that returns an abstraction (that may not be the most precise one): this function should simply be such that $n \in \gamma_V(\phi_V(n))$ for any value n . Second, when the expression consists of a variable, the analysis should simply return the value abstraction associated to this variable in the non-relational abstract pre-condition. Last, to over-approximate the result of a binary operation $E_0 \odot E_1$ assuming over-approximations of the results of E_0 and E_1 , the analysis simply needs to apply a conservative approximation of the operator f_\odot in the non-relational lattice. Such an approximation should consist of an operator $f_\odot^\# : A_V \times A_V \rightarrow A_V$, such that

$$\text{for all } n_0^\#, n_1^\# \in A_V, \{f_\odot(n_0, n_1) \mid n_0 \in \gamma_V(n_0^\#), \text{ and } n_1 \in \gamma_V(n_1^\#)\} \subseteq \gamma_V(f_\odot^\#(n_0^\#, n_1^\#))$$

Intuitively, this operator should over-approximate the effect of the operation on concrete values. For instance, if we consider addition and assume that A_V is the domain of intervals, then $f_+^\#$ should compute an over-approximation for the addition of intervals:

$$\begin{aligned} f_+^\#([a, b], [a', b']) &= [a + a', b + b'] \\ f_+^\#([a, b], [a', +\infty)) &= [a + a', +\infty) \end{aligned}$$

Other arithmetic operations have similar counterparts, although sometimes more complicated (e.g., multiplication requires a number of case splits to handle positive and negative inputs).

The full definition of the abstract semantics of scalar expressions is shown in figure 3.8.

Example 3.10 (Abstract semantics of expressions) We assume that we use the interval abstract domain, that we consider $x+2*y-6$, and that $M^\#$ is defined by $M^\#(x) = [10,20]$ and $M^\#(y) = [8,9]$. For short, we denote operations over intervals just like the conventional arithmetic operations. Then,

$$\begin{aligned}
\llbracket x + 2 * y - 6 \rrbracket^{\#}(M^{\#}) &= f_{-}^{\#}(\llbracket x + 2 * y \rrbracket^{\#}(M^{\#}), \llbracket 6 \rrbracket^{\#}(M^{\#})) \\
&= f_{+}^{\#}(\llbracket x \rrbracket^{\#}(M^{\#}), \llbracket 2 * y \rrbracket^{\#}(M^{\#})) - [6, 6] \\
&= M^{\#}(x) + f_{*}^{\#}(\llbracket 2 \rrbracket^{\#}(M^{\#}), \llbracket y \rrbracket^{\#}(M^{\#})) - [6, 6] \\
&= [10, 20] + [2, 2] * [8, 9] - [6, 6] \\
&= [20, 32]
\end{aligned}$$

We can prove by induction over the structure of expressions that this semantics is *sound* (the proof is provided in appendix B.2.1) as follows.

Theorem 3.2 (Soundness of the abstract interpretation of expressions) *For all expressions E, for all non-relational abstract elements M[#], and for all memory states m such that m ∈ γ(M[#]),*

$$\llbracket E \rrbracket (m) \in \gamma_V(\llbracket E \rrbracket^{\#}(M^{\#}))$$

Analysis of Assignments We now define the analysis function for an assignment command $x := E$. We recall that $\llbracket x := E \rrbracket_P(M) = \{m[x \mapsto \llbracket E \rrbracket(m)] \mid m \in M\}$. Intuitively, an assignment is the composition of the evaluation of the expression into a value n and of the update of the variable x with this value n. By theorem 3.1, this composition can be over-approximated piece by piece. We showed in the previous paragraph how to compute an over-approximation of the right-hand side of the assignment. The abstract counterpart of a write into a concrete memory is a write in the abstract store. Therefore, the following definition provides a sound over-approximation for the concrete semantics of assignments:

$$\llbracket x := E \rrbracket_{\mathcal{P}}^{\#}(M^{\#}) = M^{\#}[x \mapsto \llbracket E \rrbracket^{\#}(M^{\#})]$$

We can also define a sound analysis function for the **input** statement; indeed, the only change is that a value chosen in a non-deterministic way is written into the variable, so in the abstract level, we should replace its value with \top_V :

$$\llbracket \text{input}(x) \rrbracket_{\mathcal{P}}^{\#}(M^{\#}) = M^{\#}[x \mapsto \top_V]$$

Example 3.11 (Analysis of an assignment command) *We consider the assignment $x := x + 2 * y - 6$ and the abstract pre-condition M[#] defined in example 3.10. Then the abstract post-condition is*

$$\llbracket x := x + 2 * y - 6 \rrbracket^{\#}(M^{\#}) = \{x \mapsto [20, 32], y \mapsto [8, 9]\}$$

Analysis of Assignments Using a Relational Abstract Domain The analysis presented in the previous paragraphs is intrinsically non-relational since it first evaluates the right-hand side expression in the value abstract domain (figure 3.8) and then updates the abstraction of the left-hand side variable. Such separate steps

prevent the inference of relations. As an example, let us consider assignment $x := y + 1$ with the abstract domain of convex polyhedra: then we expect the analysis to infer the post-condition $x \leq y + 1 \wedge x \geq y + 1$, which involves both the left-hand side variable x and the right-hand side expression $y + 1$.

For this reason, relational abstract domains implement specific algorithms for the analysis of assignments. A common way of analyzing $x := E$ in a relational domain proceeds as follows.

1. Add a temporary dimension x' that is meant to describe the value of the expression.
2. Represent as precisely as possible the constraint $x' = E$.
3. Project out dimension x' , and rename x' to x .

The following example shows this technique.

Example 3.12 (Relational analysis of an assignment) We use the abstract domain of convex polyhedra, and we assume the abstract pre-condition $2 \leq x \leq 3 \wedge 1 - x \leq y$. We consider the assignment $x := y + x + 2$. After introducing temporary x' , we can represent the assignment effect exactly:

$$2 \leq x \leq 3 \wedge 1 - x \leq y \wedge x' = y + x + 2$$

The last term allows rewriting all occurrences of x into $x' - y - 2$, so that we get the conjunction of constraints $2 \leq x' - y - 2 \leq 3 \wedge 3 - x' + y \leq y$. After projection and renaming, we finally get $4 \leq x - y \leq 5 \wedge 3 \leq x$.

3.3.2 Abstract Interpretation of Conditional Branching

To design a static analysis algorithm for conditional commands, we follow the same principle as in the previous paragraphs, and we over-approximate the definition of the concrete semantics step-by-step. The semantics of a command **if**(B) { C_0 } **else** { C_1 } is defined by

$$\llbracket \text{if}(B) \{C_0\} \text{else} \{C_1\} \rrbracket_{\mathcal{P}}(M) = \llbracket E_0 \rrbracket_{\mathcal{P}}(\mathcal{F}_B(M)) \cup \llbracket E_1 \rrbracket_{\mathcal{P}}(\mathcal{F}_{\neg B}(M))$$

This formula guides the design of the analysis. To construct the analysis function of this statement, we will first design an operation to over-approximate \mathcal{F}_B for any Boolean expression B (recall that the negation $\neg B$ is simply a Boolean expression computed from B by syntactic transformation, so that the definition of \mathcal{F}_B includes that of $\mathcal{F}_{\neg B}$); then we will use the abstract semantics of both branches (as part of the definition of the analysis by induction over the syntax), and finally, we will apply an over-approximation of the union of concrete sets. The non-deterministic choice

construction of chapter 2 does not have a condition and is thus simpler (the analysis of non-deterministic choice in section 2.3.3 does not require any abstract filtering operation).

Analysis of Conditions To analyze conditions, we first construct an abstraction of the concrete filtering function, which we denote by \mathcal{F}_B^\sharp . In the concrete level, $\mathcal{F}_B(M)$ returns the memory states in M such that the condition B evaluates to **true**. In the abstract level, \mathcal{F}_B^\sharp should thus input an abstract state and refine it so as to take into account that B should evaluate to **true**. Therefore, this operator should satisfy the following soundness condition:

$$\text{For all conditions } B, \text{ and for all abstract states } M^\sharp, \mathcal{F}_B(\gamma(M^\sharp)) \subseteq \gamma(\mathcal{F}_B^\sharp(M^\sharp))$$

Boolean expressions are of the form $x \otimes n$, where x is a variable, \otimes is a comparison operator, and n is a scalar value. Therefore, a sound and precise \mathcal{F}_B^\sharp simply adds novel constraints to the abstract state, as shown in the following cases (we show only a few representative cases).

- With the signs abstract domain $\{\perp, \top, [= 0], [\geq 0], [\leq 0]\}$:

$$\mathcal{F}_{x < 0}^\sharp(M^\sharp) = \begin{cases} (y \in \mathbb{X}) \mapsto \perp & \text{if } M^\sharp(x) = [\geq 0] \text{ or } [= 0] \text{ or } \perp \\ M^\sharp[x \mapsto [\leq 0]] & \text{if } M^\sharp(x) = [\leq 0] \text{ or } \top \end{cases}$$

The first case occurs when M^\sharp contains information that entails the condition will always evaluate to **false**, so that the set of memories for which the condition evaluates to **true** is empty and is described by the function that maps all variables to \perp . The second case occurs when the condition may evaluate to **true**, and the information about x gets refined so as to take this into account. Similar rules handle conditions with other comparison operators and constants.

- With the intervals abstract domain, and if $M^\sharp(x) = [a, b]$, then

$$\mathcal{F}_{x < n}^\sharp(M^\sharp) = \begin{cases} (y \in \mathbb{X}) \mapsto \perp & \text{if } a > n \\ M^\sharp[x \mapsto [a, n]] & \text{if } a \leq n \leq b \\ M^\sharp & \text{if } b \leq n \end{cases}$$

The first case occurs when M^\sharp contains information that entails the condition will always evaluate to **false**. The second case occurs when the abstract information can be refined by tightening the right

bound, and the third case occurs when all stores represented by $M^\#$ satisfy the condition $x < n$.

Example 3.13 (Analysis of a condition) We consider the code fragment below that computes the absolute value of $x - 7$ into variable y :

```

if(x > 7){
    y := x - 7
} else{
    y := 7 - x
}
```

We assume the abstract pre-condition $M^\#$ defined by $x \mapsto \top$, $y \mapsto \top$. Then, by the rules shown above (assuming the scalar values are of integer type)

$$\begin{aligned}\mathcal{F}_{x>7}^\sharp(M^\#) &= M^\#[x \mapsto [8, +\infty]] \\ \mathcal{F}_{x \leq 7}^\sharp(M^\#) &= M^\#[x \mapsto (-\infty, 7]]\end{aligned}$$

Although we do not provide the full definition of the condition analysis operator, we note that it should be proved sound (the proof is given in appendix B.2.2) as follows.

Theorem 3.3 (Soundness of the abstract interpretation of conditions) For all expressions B , for all non-relational abstract elements $M^\#$, and for all memory states m such that $m \in \gamma(M^\#)$,

$$\text{if } \llbracket B \rrbracket(m) = \text{true}, \quad \text{then} \quad m \in \gamma(\mathcal{F}_B^\sharp(M^\#))$$

In particular, the cases that we have shown above for the condition tests analysis in the signs and intervals abstract domain trivially meet the soundness condition of theorem 3.3.

Analysis of Flow Joins The concrete semantics computes the union of the results of both branches. Thus, the analysis should over-approximate unions of sets of concrete states. Therefore, the abstract join operator $\sqcup^\#$ should satisfy the following soundness property:

$$\gamma(M_0^\#) \cup \gamma(M_1^\#) \subseteq \gamma(M_0^\# \sqcup^\# M_1^\#)$$

In section 2.3.3, this was achieved by computing a weaker set of constraints (that define the convex hull in the geometrical point of view). Likewise, to define an abstract union operator in the non-relational abstract domain, we can simply

- define a join operator $\sqcup_\gamma^\#$ in the value abstract domain that satisfies a similar soundness condition (we remark that, both for signs and

for intervals, the least upper bound in the abstract domain, in the sense of figure 3.5, provides a sound choice for $\sqcup_{\gamma}^{\#}$; and

- apply operator $\sqcup_{\gamma}^{\#}$ in a pointwise manner:

$$\text{For all variables } x, (M_0^{\#} \sqcup^{\#} M_1^{\#})(x) = M_0^{\#}(x) \sqcup_{\gamma}^{\#} M_1^{\#}(x)$$

The definition of $\sqcup_{\gamma}^{\#}$ depends on the abstract domain, for instance, for the interval domain:

$$\begin{aligned} [a_0, b_0] \sqcup_{\gamma}^{\#} [a_1, b_1] &= [\min(a_0, a_1), \max(b_0, b_1)] \\ [a_0, b_0] \sqcup_{\gamma}^{\#} [a_1, +\infty) &= [\min(a_0, a_1), +\infty) \end{aligned}$$

The abstract union operator defined for the non-relational domain is sound in the following sense (the proof is given in appendix B.2.3).

Theorem 3.4 (Soundness of abstract join) *Let $M_0^{\#}$ and $M_1^{\#}$ be two abstract states. Then,*

$$\gamma(M_0^{\#}) \cup \gamma(M_1^{\#}) \subseteq \gamma(M_0^{\#} \sqcup^{\#} M_1^{\#})$$

We now show how this operator works on an example.

Example 3.14 (Analysis of flow joins) *Let us consider the following abstract states:*

$$\begin{aligned} M_0^{\#} &= \{x \mapsto [0, 3], y \mapsto [6, 7], z \mapsto [4, 8]\} \\ M_1^{\#} &= \{x \mapsto [5, 6], y \mapsto [0, 2], z \mapsto [6, 9]\} \end{aligned}$$

Then:

$$M_0^{\#} \sqcup^{\#} M_1^{\#} = \{x \mapsto [0, 6], y \mapsto [0, 7], z \mapsto [4, 9]\}$$

Analysis of a Conditional Command We can now put together the definitions of all the elements defined in the previous paragraphs and obtain a definition for the abstract semantics of conditional commands:

$$[\text{if}(B)\{C_0\}\text{else}\{C_1\}]_{\mathcal{P}}^{\#}(M^{\#}) = [C_0]_{\mathcal{P}}^{\#}(\mathcal{F}_B^{\#}(M^{\#})) \sqcup^{\#} [C_1]_{\mathcal{P}}^{\#}(\mathcal{F}_{\neg B}^{\#}(M^{\#}))$$

Intuitively, this analysis definition is very similar to that of section 2.3.3, except for the $\mathcal{F}_B^{\#}$ calls, which is not surprising since the language of chapter 2 featured only non-deterministic choice (no condition formula).

Example 3.15 (Analysis of a conditional command) *We demonstrate the analysis of the program of example 3.13, starting from the abstract pre-condition $M^{\#} = \{x \mapsto \top, y \mapsto \top\}$. Then the analysis proceeds as follows:*

1. First, the analysis of the **true** branch applies the filtering function and then computes a post-condition for the assignment command $y := x - 7$; it produces

$$\{x \mapsto [8, +\infty), y \mapsto [1, +\infty)\}$$

2. In the same way, the analysis of the **false** branch produces the abstract state

$$\{x \mapsto (-\infty, 7], y \mapsto [0, +\infty)\}$$

3. Last, the abstract join of these two abstract states yields

$$\{x \mapsto T, y \mapsto [0, +\infty)\}$$

Analysis of Conditional Commands with a Relational Abstract Domain So far, we have considered the analysis of conditional commands under the assumption that the abstract domain is non-relational. Switching to a relational abstract domain would let us still use the same method but with different algorithms for the analysis of tests and for the computation of abstract join.

As for non-relational domains, the analysis of a condition test with a relational domain boils down to the addition of one or several constraints to the abstract state. It is in general more precise since condition tests that involve several variables are more likely to be represented exactly; as an example, condition $x \leq y$ can be analyzed exactly with the abstract domain of convex polyhedra (by adding it to the abstract state), whereas analyzing this test with a non-relational abstract domain generally induces a loss of precision.

3.3.3 Abstract Interpretation of Loops

The last program construction left to analyze is the loop command. As in the case of the other commands, we rely on the concrete semantics in order to design the abstract semantics. In section 2.3.4, we proposed a loop static analysis algorithm that iterates the analysis of the loop body until stabilization. The concrete semantics shown in section 3.1.2 lends itself to the design of such an abstract semantics; therefore, we indeed expect the analysis of a loop command to proceed by iteration in the general case too. Another takeaway of section 2.3.4 is that the termination of the analysis requires special care, either using abstract domains that naturally guarantee termination or using a widening operator.

```

x := 0;
while(x ≤ 100){
    if(x ≥ 50){
        x := 10
    }else{
        x := x + 1
    }
}
(a) Incrementation

x := 0;
while(x ≥ 0){
    x := x + 1
    if(x ≥ 50){
        x := 10
    }else{
        x := x + 1
    }
}
(b) Incrementation with reset

```

Figure 3.9

Two programs containing a simple loop

Concrete Semantics of Loops To better design an analysis algorithm for loops, we first study their concrete semantics a bit more. In section 3.1.2, we set up the following semantics of loops:

$$[\![\text{while}(B)\{C\}]\!]_{\mathcal{P}}(M) = \mathcal{F}_{\neg B} \left(\bigcup_{i \geq 0} ([\![C]\!]_{\mathcal{P}} \circ \mathcal{F}_B)^i(M) \right)$$

We are defining the abstract semantics by induction over the syntax; therefore, we can assume that we can compute an over-approximation of $\llbracket C \rrbracket_P$. In previous paragraphs we have defined over-approximations for \mathcal{F}_B and $\mathcal{F}_{\neg B}$. For example, we have also observed when defining the analysis of sequences of commands that an over-approximation of function compositions can be obtained by composing over-approximations of each command.

Therefore, the problem we need to solve is to compute an over-approximation for an infinite union $\bigcup_{i \geq 0} F^i(M)$ under the assumption that the function $F^\#$ over-approximates the function F (in the sense that $F \circ \gamma(M^\#) \subseteq \gamma \circ F^\#(M^\#)$ for any abstract element $M^\#$), where $F = \llbracket C \rrbracket_P \circ \mathcal{F}_B$.

Example 3.16 (Analysis of programs with loops) In the following paragraphs, we use the example programs shown in figure 3.9 so as to demonstrate the computation of loop invariants. The program of figure 3.9(a) consists of a simple loop that increments variable x forever (recall we are assuming that

(all computations are done with mathematical numbers and not with machine integers, so that we are not dealing with wrap-around arithmetics). The program of figure 3.9(b) also consists of a loop that increments variable x , but one that also resets x to 10, whenever it reaches a certain value.

Sequences of Concrete and Abstract Iterates First, let us consider the executions that spend at most n iterations in the loop, for a fixed integer value n . Then the states they generate at the loop head are defined as

$$M_n = \bigcup_{i=0}^n F^i(M)$$

We can prove that this set of states can be computed by induction over n . Indeed, let us consider the first elements of the sequence $(M_k)_{k \in \mathbb{N}}$:

- $M_0 = M$;
- $M_1 = M \cup F(M) = M \cup F(M_0)$;
- $M_2 = M \cup F(M) \cup F(F(M)) = M \cup F(M \cup F(M))$ since F commutes with set union, so $M_2 = M \cup F(M_1)$;
- for any n greater than 2, we can show in the same manner than $M_{n+1} = M \cup F(M_n)$.

This implies that the sequence $(M_k)_{k \in \mathbb{N}}$ can equivalently be defined recursively as follows:

$$\begin{aligned} M_0 &= M \\ M_{k+1} &= M_k \cup F(M_k) \end{aligned}$$

This observation is of great interest for the purpose of designing an algorithm to analyze loops since it is very easy to compute an over-approximation of M_n using the techniques shown in the previous sections. Indeed, let us assume an element $M^\#$ of the abstract domain such that $M \subseteq \gamma(M^\#)$, and we define the sequence of abstract iterates $(M_k^\#)_{k \in \mathbb{N}}$ as follows:

$$\begin{aligned} M_0^\# &= M^\# \\ M_{k+1}^\# &= M_k^\# \sqcup^\# F^\#(M_k^\#) \end{aligned} \tag{3.1}$$

Then we can prove by induction that, for all integers n ,

$$M_n \subseteq \gamma(M_n^\#)$$

Example 3.17 (Abstract iterates) We assume that the analysis uses the abstract domain of intervals, and we show the abstract iterates for the two example programs shown in figure 3.9. In both cases, the analysis of the command $x := 0$ produces the abstract state $\{x \mapsto [0,0]\}$.

- In the case of the program of figure 3.9(a), we observe the following

$$\begin{aligned}
M_0^\# &= \{\mathbf{x} \mapsto [0, 0]\} \\
M_1^\# &= \{\mathbf{x} \mapsto [0, 1]\} \\
M_2^\# &= \{\mathbf{x} \mapsto [0, 2]\} \\
&\vdots = \vdots \\
M_n^\# &= \{\mathbf{x} \mapsto [0, n]\} \\
&\vdots = \vdots
\end{aligned}$$

- In the case of the program of figure 3.9(b), we obtain the following

$$\begin{aligned}
M_0^\# &= \{\mathbf{x} \mapsto [0, 0]\} \\
M_1^\# &= \{\mathbf{x} \mapsto [0, 1]\} \\
M_2^\# &= \{\mathbf{x} \mapsto [0, 2]\} \\
&\vdots = \vdots \\
M_{49}^\# &= \{\mathbf{x} \mapsto [0, 49]\} \\
M_{50}^\# &= \{\mathbf{x} \mapsto [0, 50]\} \\
M_{51}^\# &= \{\mathbf{x} \mapsto [0, 50]\} \\
M_{52}^\# &= \{\mathbf{x} \mapsto [0, 50]\} \\
&\vdots = \vdots
\end{aligned}$$

Convergence of Iterates The recursive formula (3.1) shows how to over-approximate any fixed number of iterates but does not settle the case of unbounded iteration and the termination problem, so we consider these issues now. We observe that the sequence $(\gamma(M_k^\#))_{k \in \mathbb{N}}$ is increasing, that is, that $\gamma(M_k^\#) \subseteq \gamma(M_{k+1}^\#)$, since $M_{k+1}^\# = M_k^\# \sqcup^\# F^\#(M_k^\#)$, and since $\sqcup^\#$ is a sound over-approximation of concrete unions (theorem 3.4). Intuitively, the elements of this sequence over-approximate larger and larger sets of concrete states, which is expected since the k th term describes all states observed in at most k iterations of the loop.

Let us assume that the abstract iteration stabilizes at some rank n , which means that $M_n^\# = M_{n+1}^\#$. A first consequence is that, for all ranks $k \geq n$, we also have $M_k^\# = M_n^\#$. This also entails that $M_k \subseteq \gamma(M_n^\#)$. Since this holds for all ranks $k \geq n$, we can also derive that

$$M_{\text{loop}} \subseteq \gamma(M_n^\#) \quad \text{where} \quad M_{\text{loop}} = \bigcup_{i \geq 0} M_i$$

Another interesting observation is that

$$M_{\text{loop}} = \bigcup_{i \geq 0} F^i(M) = \bigcup_{i \geq 0} M_i$$

Thus

$$M_{\text{loop}} = \bigcup_{i \geq 0} F^i(M) \subseteq \gamma(M_n^\sharp)$$

As a consequence, if the sequence of abstract iterates converges (which can be observed simply by checking that two consecutive iterates are equal), then its final value over-approximates *all* the concrete behaviors of the program **while**(B) $\{C\}$. This effectively provides an algorithm for analyzing loops, under the assumption that the sequence converges. However, the following example shows that this assumption is problematic.

Example 3.18 (Convergence of abstract iterates) We consider the programs studied in example 3.17:

`abs_iter(F^\sharp, M^\sharp)`

`R $\leftarrow M^\sharp;$`
`repeat`

`T $\leftarrow R;$`

`R $\leftarrow R \sqcup^\sharp F^\sharp(R);$`

`until R = T`

`return $M_{\text{lim}}^\sharp = T;$`

(a) Iteration with a finite height domain

`abs_iter(F^\sharp, M^\sharp)`

`R $\leftarrow M^\sharp;$`
`repeat`

`T $\leftarrow R;$`

`R $\leftarrow R \nabla F^\sharp(R);$`

`until R = T`

`return $M_{\text{lim}}^\sharp = T;$`

(b) Iteration with widening and a domain with possibly infinite height

Figure 3.10

Loop analysis algorithms

- In the case of the program of figure 3.9(a), since the ranges computed for x are always increasing, the sequence of abstract iterates does not converge.
- In the case of the program of figure 3.9(b), the ranges computed for x do stabilize but only after 51 iterations.

As we can see, neither of these two analyses is really satisfactory, because of either lack of termination or a high number of iterates required to converge.

Therefore, the following paragraphs formalize conditions that ensure that the sequence of abstract iterates converges or, equivalently, that $M_{n+1}^\sharp = M_n^\sharp$ for some bounded rank n .

Convergence in Finite Height Lattices In this paragraph, we assume that \sqsubseteq is such that $M_a^\sharp \sqsubseteq M_b^\sharp$ if and only if $\gamma(M_a^\sharp) \subseteq \gamma(M_b^\sharp)$ for all abstract states M_a^\sharp, M_b^\sharp . (this

condition holds for all the value abstract domains introduced in section 3.2). In particular, this means that the sequence of abstract iterates is increasing, that is, for all k , $M_k^\sharp \sqsubseteq M_{k+1}^\sharp$. Thus, a first case where convergence is ensured is when the structure of the abstract domain guarantees that the condition $M_k^\sharp \sqsubset M_{k+1}^\sharp$ (which means $M_k^\sharp \sqsubseteq M_{k+1}^\sharp$ and $M_k^\sharp \neq M_{k+1}^\sharp$) cannot hold infinitely many times. In particular, this new condition is realized when the abstract domain has *finite height*, that is, when the length of all the chains of the form $M_0^\sharp \sqsubset M_1^\sharp \sqsubset \dots \sqsubset M_k^\sharp$ is bounded by some fixed value h , called the *height of the abstract domain*. Indeed, let us assume that the abstract domain has finite height h ; then the sequence $M_0^\sharp, M_1^\sharp, \dots, M_h^\sharp, M_{h+1}^\sharp$ is increasing for \sqsubseteq but cannot be strictly increasing, so there exists a rank lower than h at which it becomes stable, and this iterate provides an over-approximation for the loop invariant. This bound is indeed the height of the Hasse diagram of the abstract domain, as in figure 3.5: we observe that the domain of signs is of height 3 and that the domain of intervals has infinite height.

We can then compute an abstract state M_{\lim}^\sharp that over-approximates M_{loop} within a finite number of iterations, using the algorithm shown in figure 3.10(a). We observe that this iterative algorithm is quite similar to the method used in section 2.3.4, even though we now consider a more realistic language. This iteration technique is actually general and can be utilized whenever analyzing programs that contain iterative constructions that resemble loops (e.g., it also adapts to recursive procedures).

Another observation is that the exit condition here boils down to a mere equality comparison. It is often possible to use \sqsubseteq instead and to let the analysis of the loop terminate when R is “included” in T , in the abstract level, that is, when the abstract iteration does not discover any new behavior.

Example 3.19 (Convergence of abstract iterates in the signs abstract domain) We demonstrate the analysis using the signs abstract domain in the cases of the two programs of example 3.16:

- In the case of the program of figure 3.9(a), we obtain the following iteration sequence:

$$\begin{aligned} M_0^\sharp &= \{x \mapsto [=0]\} \\ M_1^\sharp &= \{x \mapsto [\geq 0]\} \\ M_2^\sharp &= \{x \mapsto [\geq 0]\} \end{aligned}$$

We observe that the analysis terminates after only two iterations.

- In the case of the program of figure 3.9(b), we obtain the same iteration sequence, and the analysis of the loop also converges after only two iterations.

Widening Operators As we noted in the previous paragraph, the abstract domain of intervals has infinite chains. In particular, the chain below would be computed by the analysis of the program of figure 3.9(a), if we were using the above algorithm:

$$[0, 0] \sqsubset [0, 1] \sqsubset \cdots \sqsubset [0, n] \sqsubset [0, n + 1] \sqsubset \cdots$$

In example 3.17, the analysis of the program of figure 3.9(b) was found to terminate, although only after a very long sequence of abstract iterates. In both examples, the issue is that the abstract domain does not enforce a quick convergence of the iteration sequence.

Therefore, we now propose a second technique to bound the length of abstract iteration, using a notion of *widening* [26], as we did in section 2.3.4. Essentially, a widening operator is a binary operator that over-approximates concrete unions and also enforces termination of all sequences of iterates:

Definition 3.11 (Widening operator) A widening operator over an abstract domain A is a binary operator ∇ , such that,

1. for all abstract elements a_0, a_1 , we have

$$\gamma(a_0) \cup \gamma(a_1) \subseteq \gamma(a_0 \nabla a_1)$$

2. for all sequences $(a_n)_{n \in \mathbb{N}}$ of abstract elements, the sequence $(a'_n)_{n \in \mathbb{N}}$ defined below is ultimately stationary:

$$\begin{cases} a'_0 &= a_0 \\ a'_{n+1} &= a'_n \nabla a_n \end{cases}$$

Assuming such an operator ∇ for the non-relational domain, we can turn the sequence of abstract iterates into a terminating sequence, as follows.

Theorem 3.5 (Abstract iterates with widening) We assume that ∇ is a widening operator over the non-relational abstract A domain and that $F^\#$ is a function from A to itself. Then the algorithm shown in figure 3.10(b) terminates, and returns an abstract element $M_{\lim}^\#$.

Moreover, if we assume that $F : M \rightarrow M$ is continuous and is such that $F \circ \gamma \subseteq \gamma \circ F^\#$ for the pointwise inclusion (which means that, for all abstract element $M_0^\#$, $F \circ \gamma(M_0^\#) \subseteq \gamma \circ F^\#(M_0^\#)$), then

$$\bigcup_{i \geq 0} F^i(\gamma(M^\#)) \subseteq \gamma(M_{\lim}^\#)$$

We observe that this result derives from the continuity property mentioned in remark 3.1.

This theorem guarantees not only the termination of the loop analysis but also its soundness; indeed, it shows that M_{\lim}^{\sharp} over-approximates the concrete semantics of the loop.

We now discuss the construction of a widening operator for the abstract domain of intervals. In section 2.3.4, we noted that such an operator could, in some cases, be built by dropping constraints that are not stable. This can be achieved very easily with intervals by simply removing unstable bounds. For instance, the widening of two intervals with the same left bound boils down to

$$[n, p] \nabla_{\mathcal{V}} [n, q] = \begin{cases} [n, p] & \text{if } p \geq q \\ [n, +\infty) & \text{if } p < q \end{cases}$$

The case of unequal left bounds is symmetric.

Example 3.20 (Widening operator for the abstract domain of intervals) We now study the analysis of the examples of example 3.16. In both cases, we obtain the following iteration sequence:

$$\begin{aligned} M_0^{\sharp} &= \{x \mapsto [0, 0]\} \\ M_1^{\sharp} &= \{x \mapsto [0, +\infty)\} \\ M_2^{\sharp} &= \{x \mapsto [0, +\infty)\} \end{aligned}$$

The convergence is now very fast. However, the result is far from precise, in the case of the example of figure 3.9(b), since the analysis fails to stabilize any finite upper bound on x . While this appears like a serious loss of precision, several common techniques allow actually obtaining much more precise bounds, and we study some of them in section 5.2.

Analysis of Loops We can now produce an analysis function for loops that inputs an abstract pre-condition and returns an abstract post-condition:

$$[\![\text{while}(B)\{C\}]\!]_{\mathcal{P}}^{\sharp}(M^{\sharp}) = \mathcal{F}_{\neg B}^{\sharp}(\text{abs_iter}([\![C]\!]_{\mathcal{P}}^{\sharp} \circ \mathcal{F}_B^{\sharp}, M^{\sharp}))$$

This formula follows directly from the definition of the concrete semantics of loops and from the loop condition over-approximation method described in theorem 3.5.

Analysis of Loops with a Relational Abstract Domain The analysis of a loop statement with a relational domain such as convex polyhedra proceeds the same way as with a non-relational domain and requires only an abstract join or widening operator specific to the abstraction being used. We note that linear equalities (definition 3.8) define a finite height lattice. Thus, they necessitate no widening. Convex polyhedra (definition 3.9) and octagons (definition 3.10) are of infinite height and require a widening operator.

Another View on the Analysis of Loops This subsection offers an alternative definition of the concrete semantics of loops and sheds a different light on their analysis. As such, the material in this subsection may be skipped in a first reading. On the other hand, it provides useful information for readers interested in the design and implementation of static analysis tools.

First, let us recall the concrete semantics of a loop statement:

$$[\![\text{while}(B)\{C\}]\!]_{\mathcal{P}}(M) = \mathcal{F}_{\neg B} \left(\bigcup_{i \geq 0} ([\![C]\!]_{\mathcal{P}} \circ \mathcal{F}_B)^i(M) \right) = \mathcal{F}_{\neg B}(M_{\text{loop}})$$

where M_{loop} represent the set of all the memory states that can be observed “at loop head”:

$$\begin{aligned} M_{\text{loop}} &= \bigcup_{i \geq 0} ([\![C]\!]_{\mathcal{P}} \circ \mathcal{F}_B)^i(M) = M \cup \left(\bigcup_{i > 0} ([\![C]\!]_{\mathcal{P}} \circ \mathcal{F}_B)^i(M) \right) \\ &= M \cup [\![C]\!]_{\mathcal{P}} \circ \mathcal{F}_B \left(\bigcup_{i \geq 0} ([\![C]\!]_{\mathcal{P}} \circ \mathcal{F}_B)^i(M) \right) \end{aligned}$$

This definition actually corresponds to the following characterization of M_{loop} :

- First, it is such that $M \cup [\![C]\!]_{\mathcal{P}} \circ \mathcal{F}_B(M_{\text{loop}}) = M_{\text{loop}}$, which means it is a *fixpoint* for the function $G : X \mapsto M \cup [\![C]\!]_{\mathcal{P}} \circ \mathcal{F}_B(X)$.
- Second, it is the smallest set of memory states that satisfies this equality, which means it the *least fixpoint* for that function that contains M .

These two results follow from the algebraic properties of the function G (essentially, it commutes with arbitrary unions over non-empty families of sets of memory states) and from the Kleene fixpoint theorem (theorem A.1, in appendix A). We let **Ifp** G denote the least fixpoint of G .

The consequences of this remark are twofold.

First, the concrete semantics of a loop can be expressed using the least fixpoint of a function that calculates the effect of either starting from a memory state in M or running one iteration of the loop:

$$[\![\text{while}(B)\{C\}]\!]_{\mathcal{P}}(M) = \mathcal{F}_{\neg B}(\text{Ifp } G) \quad \text{where} \quad G : X \mapsto M \cup [\![C]\!]_{\mathcal{P}} \circ \mathcal{F}_B(X)$$

Second, the abstract semantics of a loop relies on the over-approximation of a concrete least fixpoint and can benefit from many techniques to compute

over-approximations of least fixpoints. We have already seen two such techniques:

- When the abstract lattice has finite height, we have proposed an iteration technique that relies on abstract union,
- In the case where the abstract lattice may have infinite height, we have proposed another technique that relies on a widening operator in order to enforce convergence.

$$\begin{aligned}
\llbracket n \rrbracket^\sharp(M^\sharp) &= \phi_Y(n) \\
\llbracket x \rrbracket^\sharp(M^\sharp) &= M^\sharp(x) \\
\llbracket E_0 \odot E_1 \rrbracket^\sharp(M^\sharp) &= f_\odot^\sharp(\llbracket E_0 \rrbracket^\sharp(M^\sharp), \llbracket E_1 \rrbracket^\sharp(M^\sharp)) \\
\llbracket C \rrbracket_\mathcal{P}^\sharp(\perp) &= \perp \\
\llbracket \text{skip} \rrbracket_\mathcal{P}^\sharp(M^\sharp) &= M^\sharp \\
\llbracket C_0; C_1 \rrbracket_\mathcal{P}^\sharp(M^\sharp) &= \llbracket C_1 \rrbracket_\mathcal{P}^\sharp(\llbracket C_0 \rrbracket_\mathcal{P}^\sharp(M^\sharp)) \\
\llbracket x := E \rrbracket_\mathcal{P}^\sharp(M^\sharp) &= M^\sharp[x \mapsto \llbracket E \rrbracket^\sharp(M^\sharp)] \\
\llbracket \text{input}(x) \rrbracket_\mathcal{P}^\sharp(M^\sharp) &= M^\sharp[x \mapsto \top_Y] \\
\llbracket \text{if}(B)\{C_0\} \text{else}\{C_1\} \rrbracket_\mathcal{P}^\sharp(M^\sharp) &= \llbracket C_0 \rrbracket_\mathcal{P}^\sharp(\mathcal{F}_B^\sharp(M^\sharp)) \sqcup^\sharp \llbracket C_1 \rrbracket_\mathcal{P}^\sharp(\mathcal{F}_{\neg B}^\sharp(M^\sharp)) \\
\llbracket \text{while}(B)\{C\} \rrbracket_\mathcal{P}^\sharp(M^\sharp) &= \mathcal{F}_{\neg B}^\sharp(\text{abs_iter}(\llbracket C \rrbracket_\mathcal{P}^\sharp \circ \mathcal{F}_B^\sharp, M^\sharp))
\end{aligned}$$

Figure 3.11

Abstract semantics, with a non-relational abstract domain

Compared to the state of the art in static analysis, these techniques are fairly basic, though, and we present several improvements in section 5.2.

3.3.4 Putting Everything Together

We can now summarize the analysis definition in figure 3.11, and discuss the properties of this analysis. This definition assumes several elements that we have defined in the course of this chapter:

- The function ϕ_Y for the abstraction of constants and the abstract counterpart f_\odot^\sharp of the operator \odot are specific to the underlying value abstraction.
- The abstract condition \mathcal{F}_B^\sharp is also specific to the value abstraction.

- The abstract union operation $\sqcup^\#$ is defined based on the value abstract union $\sqcup_\gamma^\#$.
- The general abstract iteration function `abs_iter` for the analysis of loops relies on the widening operator ∇ .

We have designed this abstract semantics step-by-step, in a way that ensures that it is sound, that is, that it never leaves out any concrete behavior of the program being analyzed, as follows.

Theorem 3.6 (Soundness) *For all commands C and all abstract states $M^\#$, the computation of $\llbracket C \rrbracket_{\mathcal{P}}^\#(M^\#)$ terminates, and*

$$\llbracket C \rrbracket_{\mathcal{P}}(\gamma(M^\#)) \subseteq \gamma(\llbracket C \rrbracket_{\mathcal{P}}^\#(M^\#))$$

The proof of this theorem proceeds by induction over the syntax of commands: for each kind of command, we ensured that the definition of its abstract semantics would lead to a sound result, assuming that its components had a sound abstract semantics. We provide the full proof of correctness of the analysis in appendix B.2.5.

Computing an Over-Approximation of All Reachable States The analysis of figure 3.11 can easily be extended with a global accumulator per program location so as to compute an over-approximation of *all* reachable states (and not just output states).

Using the Results of the Analysis to Prove the Properties of Interest This soundness theorem establishes that the analysis computes an over-approximation of the states that the program may produce. This over-approximation may be used to check reachability properties. For instance, if one intends to verify that all the states that may be observed when running a program C from a state in $\gamma(M^\#)$ belong to a set M , one may simply run the analyzer and attempt to verify that its output $\gamma(\llbracket C \rrbracket_{\mathcal{P}}^\#(M^\#))$ is included in M . We observe that this method is sound (when it succeeds, it is guaranteed that $\llbracket C \rrbracket_{\mathcal{P}}(\gamma(M^\#)) \subseteq M$) but incomplete: when the inclusion $\gamma(\llbracket C \rrbracket_{\mathcal{P}}^\#(M^\#)) \subseteq M$ does not hold, no information can be derived; in fact, either there may exist an execution that violates the property of interest or the analysis may just be imprecise. In general, when this inclusion does not hold, static analysis tools emit reports that they failed to prove the property of interest, which are generally called *alarms*, and users

need to further inspect the analysis results to decide whether the alarm is true (the property is indeed violated) or false (the program actually satisfies it). This process is called *triage* and is studied in detail in section 6.3.

The analysis function $\llbracket C \rrbracket_{\wp}^{\sharp}$ is not monotone in general. In some cases, it is monotone, but in other cases it is not. In fact our framework does not assume anything about the monotonicity or non-monotonicity of the analysis function. Therefore, replacing the abstract pre-condition M^{\sharp} with a more precise one does not entail that $\gamma(\llbracket C \rrbracket_{\wp}^{\sharp}(M^{\sharp}))$ also becomes more precise. In fact, the consequence of the lack of monotonicity of all the abstract operations also means that replacing one abstract operation with a more precise one would not necessarily lead to an analysis that is more precise overall.

Adapting the Analysis to Use a Different Abstraction We remark that the use of a relational domain actually changes little the analysis of statements as defined in figure 3.11. The analysis of assignments in figure 3.11 is intrinsically non-relational since it evaluates expressions in the value domain. Switching to a relational abstract domain would require using a relational algorithm to compute post-conditions for assignments and for **input** commands. However, the overall structure of the analysis would not need to be modified. In fact, many static analysis tools are parametric in the choice of the abstract domain, which means that the user may decide which abstract domain should be used, for instance, using some launch option. More generally, the practical considerations related to the setting of the analysis are discussed in section 6.2.

We also observe that the whole analysis design could also have been guided by a best abstraction function α instead of γ . In that case, the soundness theorem would resemble

$$\alpha(\llbracket C \rrbracket_{\wp}(M)) \sqsubseteq \llbracket C \rrbracket_{\wp}^{\sharp}(\alpha(M))$$

3.4 The Design of an Abstract Interpreter

To conclude this section, we summarize the process we have followed in this chapter, so as to construct our abstract semantics. At this point, we can also comment further on the discussion related to the structure of a static analyzer. We have already discussed it in section 2.5, in the context of an informal introduction to the analysis of a graphical language (chapter 2).

- First, we fix a concrete semantics that describes the behaviors of programs in a faithful and formal manner. This step is crucial because it later serves as the foundation of the analysis, since it specifies the properties the analysis should compute and is the reference compared to which soundness is to be proved.
- Second, we fix an abstraction that defines the set of logical predicates that the analysis may use, and their computer representation. This also guides the definition of the analysis algorithms. In this section we use a non-relational abstraction that can be parameterized by the choice of the value abstraction (signs, intervals, or other). In general, the choice of the abstraction should be guided by the property of interest and by all the logical facts that need to be tracked in order to establish the property of interest.
- Last, we define the abstract semantics. This step is guided by the semantics and the abstraction to a large extent, although there are often several choices left for the definition of abstract operators. As an example, ϕ_V , f^\sharp and $\sqcup^\#$ are simply required to be sound. In certain cases, we may prefer intentionally choosing imprecise operators, for the sake of efficiency, faster convergence, or other practical reasons.

These three steps describe a standard and general process to construct a static analysis:

1. fix the reference concrete semantics,
2. select the abstraction, and
3. derive analysis algorithms.

This methodology applies not only to imperative languages, such as the one considered in this chapter, but also to other programming paradigms, such as logic programming languages [59], mobile systems [41], or hardware designs [60, 105].

This division of the analysis design into independent steps is of great interest not only for the construction of a static analysis tool but also when a static analysis needs to be improved. When an analysis is not precise enough, one should identify which properties it fails to compute and improve either the concrete semantics (step 1), the abstraction (step 2), or the analysis algorithms (step 3), depending on where the lack of precision stems from. Very often the imprecisions stem from too coarse an abstraction that fails to

express properties of interest, and in that case, step 2 should be revisited. Another common source of imprecision is when the analysis algorithms (e.g., widening) return overly approximate results for the sake of cost, and then step 3 should be improved. It may also happen that the concrete semantics is too coarse to express the properties of interest, and then step 1 should be reconsidered. More generally, this structured design of abstract interpreters makes the search for the origin of lack of completeness more systematic [46].

1. Concrete semantics:

$$[\![C]\!]_{\mathcal{P}} : \wp(\mathbb{M}) \longrightarrow \wp(\mathbb{M})$$

The actual definition relies on the following:

- The definition of the set of memory states \mathbb{M}
- Concrete operations f_{\odot} for each operator in the language
- Filter functions \mathcal{F}_B (for tests)
- Set union \cup (for conditions)
- Infinite set union $\cup/\text{least fixpoint}$ (for loops)

2. Abstraction:

$$\begin{aligned} \mathbb{A} &= (\mathbb{X} \rightarrow \mathbb{A}_{\gamma}) \\ \gamma &: \mathbb{A} \longrightarrow \wp(\mathbb{M}) \end{aligned}$$

The actual definition relies on the value abstraction \mathbb{A}_{γ} , and its concretization function $\gamma_{\gamma} : \mathbb{A}_{\gamma} \rightarrow \wp(\mathbb{V})$.

3. Abstract semantics:

$$[\![C]\!]^{\sharp}_{\mathcal{P}} : \mathbb{A} \longrightarrow \mathbb{A}$$

The actual definition relies on the following:

- An abstract function f_{\odot}^{\sharp} for each concrete operator f_{\odot} such that f_{\odot}^{\sharp} computes a sound approximation of f_{\odot}
- An abstract filter function \mathcal{F}_B^{\sharp} that is sound with respect to \mathcal{F}_B (for tests)
- An abstract join operator \sqcup^{\sharp} that over-approximates set union \cup (for the analysis of control flow joins)
- An over-approximation of concrete fixpoints, based on a widening operator if \mathbb{A} has infinite height (for the analysis of loops)

Figure 3.12

An abstract interpretation framework, based on a non-relational abstract domain

Figure 3.12 summarizes the framework used in this chapter and the objects that need to be defined at each of these three steps. Moreover, the informal presentation of chapter 2 also follows these steps, albeit for a rather contrived programming language. Therefore, they define a general process to design abstract semantics, so as to build static analyses.

4 A General Static Analysis Framework Based on a Transitional Semantics

Goal of This Chapter In this chapter, we provide a formal introduction to static analysis by abstract interpretation in the transitional style. This framework is general and can be instantiated for different languages and different abstractions. We show a step-by-step recipe for constructing a sound static analysis in this framework. Following the recipe will result in a sound static analysis. This soundness guarantee is summarized in theorems whose proofs are in appendix B. We assume that the readers are already familiar with key concepts of static analysis in abstract interpretation. The intuition conveyed in chapter 2 supports most of the contents of this framework. Understanding the concept of abstraction and being familiar with its formal notions of section 3.2.1 are necessary.

Recommended Reading: [S], [D], [U] We recommend this chapter to all readers since it defines the core concepts of static analysis and is fundamental to the understanding of most of the following chapters in the book. Readers less interested in the foundations may skip some parts of the analysis design, whereas readers who would like to fully understand how static analyses achieve sound results may want to read the proofs supplied in appendix B. Moreover, readers interested in implementation may also read this chapter together with chapter 7.

Chapter Outline: Recipe for the Construction of an Abstract Interpreter in a Transitional-Style Semantics We present a general framework of designing a sound static analysis by abstract interpretation in transitional style. The presentation of this framework is not bound to a particular programming language. We present the framework solely in the semantic level, without referring to the syntax of a specific target programming language. A transition-style semantics allows this parameterization.

1. In section 4.1, we define semantics as state transitions and show such semantics for an example program snippet. We then present a recipe for defining the concrete state-transition semantics.
2. In section 4.2, we present a recipe for defining an abstract state-transition semantics that is a sound over-approximation of a concrete semantics as defined in section 4.1.
3. In section 4.3, we present analysis algorithms that compute abstract state-transition semantics as defined in section 4.2.
4. In section 4.4, we fix a simple imperative language and illustrate a use of the recipes of sections 4.1 and 4.2 in defining a correct analysis.

4.1 Semantics as State Transitions

We use a transitional-style approach to define semantics. In this style we define concrete and abstract semantics in the small-step operational semantics.

This style of semantics is handy for languages whose compositional semantics (also known as *denotational semantics*) is not obvious. For example, if the target programming language has dynamic jumps (e.g., function calls, local gotos, jump labels as values, non-local gotos, function pointers, functions as values, dynamic method dispatches, or exception raises), then defining its compositional semantics becomes a burden. With gotos, programs may loop with an arbitrary portion of the program, not tamed to a particular construct such as the while-loop. Defining the compositional semantics for such language feature needs an advanced knowledge in programming language semantics. Transitional semantics (one style of *operational semantics*), on the other hand, is free from the need to be compositional and is relatively easy to define, once we understand how programs operate.

The transitional style is also a good fit for the proof of the reachability property. For this property, the static analysis goal is to over-approximate the set of reachable states of the input program. This set is obvious in the transitional style because the semantics explicitly exposes all the intermediate states of program executions.

4.1.1 Concrete Semantics

We start from the concrete semantics. The concrete semantics of a programming language defines the run-time behaviors of its programs. Informally, the concrete semantics of a language is what programmers have in mind about the run-time behaviors of their programs when they program.

The transitional-style semantics of a program is defined as the set of all possible sequences of state transitions from the initial states. We write a concrete state transition as

$$s \leftrightarrow s'$$

A sequence

$$s_0 \leftrightarrow s_1 \leftrightarrow s_2 \leftrightarrow \dots$$

of state transitions is the chain that links the transitions

$$s_0 \leftrightarrow s_1, s_1 \leftrightarrow s_2, \dots.$$

A state $s \in S$ of the program is a pair (l,m) of a program label l and the machine state m at that program label during execution. The program label denotes the part of the program that is to be executed next. The machine state is usually the memory state that contains the effect of the program's hitherto execution and a data for the program's continuation. For the example language of chapter 2, a machine state is a point in the two-dimensional space because program's execution step transforms a point to another point. For conventional imperative languages with local blocks and function calls, the machine state would consist of a memory (a table from locations to storable values), an environment (a table from program variables to locations), and a continuation (a stack of return contexts where a return context is a program label and an environment to resume at the return of a function).

One step of the state transition relation,

$$(l,m) \xrightarrow{} (l',m'),$$

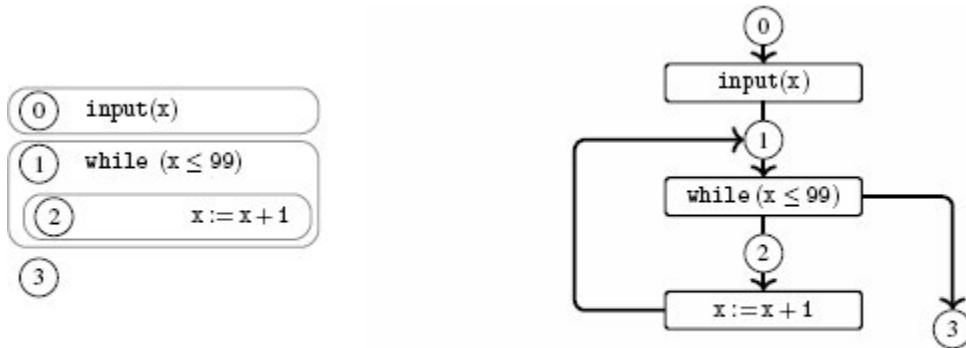
is defined by the language construct of the program part pointed to by l . The next memory state m' is the result of executing the program part at l by one step.

For simple languages, the next label l' of the program (called *control flow*) is determined by the program syntax. In case the control flow is not determined solely by the syntax but is determined by the program execution (e.g., goto target as values, function pointers, functions as values, or dynamic method dispatches), the next program label l' is an evaluation result from the current program label l and the current machine state m .

Example 4.1 (Concrete transition sequence) Consider the following program:

```
input(x);
while (x ≤ 99)
    {x := x + 1}
```

The labeled representations of this program in text and graph are, respectively,



Let the initial state be the empty memory \emptyset . Transition sequences for some integer inputs are as follows:

For input 100: $(0, \emptyset) \rightarrow (1, x \mapsto 100) \rightarrow (3, x \mapsto 100)$.

For input 99: $(0, \emptyset) \rightarrow (1, x \mapsto 99) \rightarrow (2, x \mapsto 99) \rightarrow (1, x \mapsto 100) \rightarrow (3, x \mapsto 100)$.

For input 0: $(0, \emptyset) \rightarrow (1, x \mapsto 0) \rightarrow (2, x \mapsto 0) \rightarrow (1, x \mapsto 1) \rightarrow \dots \rightarrow (3, x \mapsto 100)$.

A transition sequence for a program can be infinitely long if the program has non-terminating executions. The number of transition sequences can be infinite too if the initial states can be infinitely many.

Set of Reachable States We restrict our analysis interest to computing the set of reachable states, the set of all states that can occur in the transition sequences of the input program.

Example 4.2 (Reachable states) For the program in example 4.1, let us assume that the possible inputs are only 0, 99, and 100. Then the set of all reachable states is the set of states occurring in the three transition sequences:

$$\begin{aligned}
& \{(0, \emptyset), (1, x \mapsto 100), (3, x \mapsto 100)\} \\
\cup & \{(0, \emptyset), (1, x \mapsto 99), (2, x \mapsto 99), (1, x \mapsto 100), (3, x \mapsto 100)\} \\
\cup & \{(0, \emptyset), (1, x \mapsto 0), (2, x \mapsto 0), (1, x \mapsto 1), \dots, (2, x \mapsto 99), (1, x \mapsto 100), (3, x \mapsto 100)\}
\end{aligned}$$

Given a program, the set of all its reachable states is intuitive in the operational sense. Starting from the set of all initial states of the program, we keep adding next states to the set. The next states are those produced by the application of the single-step transition \hookrightarrow to each state in the current set. We keep adding next states until no more addition is possible. The final set is the set of all reachable states.

We will define this set of reachable states in mathematical terms. This mathematical formalization is a necessary step in our framework because it will later be a reference in proving the soundness of a designed static analysis. We will see that the mathematical concept called *the least fixpoint of a monotonic function* exactly defines the reachable set.

Given a program, let I be the set of its initial states, and let $Step$ be the powerset-lifted version of \hookrightarrow :

$$\begin{aligned}
Step : \wp(\mathbb{S}) &\rightarrow \wp(\mathbb{S}) \\
Step(X) &= \{s' \mid s \hookrightarrow s', s \in X\}
\end{aligned}$$

Note that the set of states that can occur right after i transitions from the set I of initial states is

$$Step^i(I),$$

where

$$\begin{aligned}
Step^0(X) &= X, \\
Step^{i+1}(X) &= Step(Step^i(X)).
\end{aligned}$$

Example 4.3 ($Step^i$ operation) For the program in example 4.1, assuming the set $I = \{(0, \emptyset)\}$ of initial states, and assuming that the possible inputs are 0, 99, and 100, we have the following:

$$\begin{aligned}
Step^0(I) &= I \\
Step^1(I) &= \{(1, x \mapsto 100), (1, x \mapsto 99), (1, x \mapsto 0)\} \\
Step^2(I) &= \{(3, x \mapsto 100), (2, x \mapsto 99), (2, x \mapsto 0)\} \\
Step^3(I) &= \{(1, x \mapsto 100), (1, x \mapsto 1)\} \\
Step^4(I) &= \{(3, x \mapsto 100), (2, x \mapsto 1)\} \\
Step^5(I) &= \{(1, x \mapsto 2)\} \\
Step^6(I) &= \{(2, x \mapsto 2)\} \\
Step^7(I) &= \{(1, x \mapsto 3)\} \\
&\vdots
\end{aligned}$$

Thus, the accumulated set of all reachable states of a program is the collection of $Step^i(I)$ for all $i \geq 0$:

$$I \cup Step^1(I) \cup Step^2(I) \cup \dots \quad (4.1)$$

We can define this set inductively as follows. Let C_i be the accumulated set $I \cup Step^1(I) \cup \dots \cup Step^i(I)$ of reachable states in 0-to- i transition steps. Then C_i can be inductively defined as

$$\begin{aligned}
C_0 &= I, \\
C_{i+1} &= I \cup Step(C_i).
\end{aligned}$$

The base C_0 is the initial set I . As of the inductive case, note that $Step(C_i)$ generates states occurring after one more step from C_i , that is, in 1-to-($i+1$) steps of transitions. Hence the set C_{i+1} of states in 0-to-($i+1$) steps of transitions is $I \cup Step(C_i)$.

The accumulated set (4.1) of all reachable states is the limit of the sequence $(C_i)_{i \in \mathbb{N}}$, a set C such that accumulating further by $I \cup Step(C)$ remains the same as C . That is, the limit is the least solution of the following equation:

$$X = I \cup Step(X)$$

Such limit corresponds, in mathematics, to the one called *least fixpoint* of the continuous monotonic function F :

$$\begin{aligned}
F : \wp(\mathbb{S}) &\rightarrow \wp(\mathbb{S}) \\
F(X) &= I \cup Step(X),
\end{aligned}$$

written as

lfp F .

The least fixpoint **lfp** F of F is constructive as follows.

Theorem 4.1 (Least fixpoint) *The least fixpoint **lfp** F of $F(X) = I \cup Step(X)$ is*

$$\bigcup_{i \geq 0} F^i(\emptyset),$$

where $F^0(X) = X$ and $F^{n+1}(X) = F(F^n(X))$.

The above theorem, which is a version of the Kleene fixpoint theorem (theorem A.1), holds because the function $F : \wp(S) \rightarrow \wp(S)$ is *continuous* over the powerset $\wp(S)$ with the set-inclusion order. (The reader may refer to appendix A.5 for the definition of continuous functions. Intuitively, any function that can be exactly implemented as a computer program is continuous.)

Definition 4.1 (Concrete semantics, the set of reachable states) *Given a program, let S be the set of states, and let \hookrightarrow be the one-step transition relation from a state to a state. Let I be the set of its initial states, and let $Step$ be the powerset-lifted version of \hookrightarrow : one-step transition relation over states:*

$$\begin{aligned} Step : \wp(S) &\rightarrow \wp(S) \\ Step(X) &= \{s' \mid s \hookrightarrow s', s \in X\}. \end{aligned}$$

Let

$$F(X) = I \cup Step(X).$$

Then the concrete semantics of the program, the set of all reachable states from I , is defined as the least fixpoint **lfp** F of F .

4.1.2 Recipe for Defining a Concrete Transitional Semantics

When building a static analysis for programs written in a programming language L , the first step is to define its concrete semantics. The concrete semantics is the basis for later steps toward a static analysis.

1. For the target programming language, define the set of states between which a single-step transition relation \hookrightarrow is to be defined. Let us name this set S .
2. Define the $s \hookrightarrow s'$ relation between states s and $s' \in S$, and let $Step$ be its natural powerset-lifted version:

$$\begin{aligned} Step : \wp(\mathbb{S}) &\rightarrow \wp(\mathbb{S}) \\ Step(X) &= \{s' \mid s \hookrightarrow s', s \in X\} \end{aligned}$$

3. Given a program of the language with its set $I \subseteq S$ of initial states, let

$$\begin{aligned} F : \wp(\mathbb{S}) &\rightarrow \wp(\mathbb{S}) \\ F(X) &= I \cup Step(X). \end{aligned}$$

The concrete semantics, defined as the set of all the reachable states of the program, is the least fixpoint of the continuous function F :

$$\text{lfp } F = \bigcup_{i \geq 0} F^i(\emptyset)$$

The concrete semantics is not what we implement as a static analyzer. Implementing this concrete semantics is rather equivalent to implementing an interpreter that actually runs the programs of the target language.

The next steps toward a static analysis consist of defining an abstract version of the concrete semantics and checking its soundness. These steps will reference the concrete semantics.

The concrete semantics as a mathematical object (as the least fixpoint) provides the foundation upon which static analysis design and its soundness check are conveniently formalized and proven. We will see that our intuitions as sketched in chapter 2 about sound static analysis and its algorithm have correspondences in mathematics.

Before we continue, we define two terms.

Definition 4.2 (Semantic domain and semantic function) *We assume the concrete semantics of a program by the least fixpoint of a function $F : \wp(S) \rightarrow \wp(S)$. Then we call the function F concrete semantic function, and we call the space $\wp(S)$ over this semantic function is defined concrete semantic domain or simply concrete domain, whose partial order is the subset order.*

4.2 Abstract Semantics as Abstract State Transitions

Given a concrete semantics, we now focus on the design of an abstract version that is finitely computable.

An abstract semantic function $F^\#$ will have the same structure as the concrete semantic function F :

$$\begin{array}{ll} F : \wp(\mathbb{S}) \rightarrow \wp(\mathbb{S}) & F^\# : \mathbb{S}^\# \rightarrow \mathbb{S}^\# \\ F(X) = I \cup Step(X) & F^\#(X^\#) = I^\# \cup^\# Step^\#(X^\#), \end{array}$$

where $I^\#$, $U^\#$, and $Step^\#$ are the abstract versions of, respectively, I , U , and $Step$. The concrete semantics of the target language consists of two parts: a semantic domain $\wp(S)$ and a semantic function over the domain $F : \wp(S) \rightarrow \wp(S)$. An abstract semantics also consists of two parts, an abstract domain $S^\#$ and an abstract semantic function $F^\# : S^\# \rightarrow S^\#$ over it. The whole purpose of this abstract version is to derive a finitely computable yet sound semantics for any program of the target language.

The forthcoming framework will guide us toward the definition of such an abstract domain $S^\#$ and the abstract semantic function $F^\#$ such that the abstract semantics of the input program is always finitely computable and is an over-approximation of the concrete semantics $\text{Ifp } F$.

4.2.1 Abstraction of the Semantic Domain

A concrete semantic domain D is the powerset of concrete states

$$\begin{aligned} D &= \wp(S), \\ S &= L \times M. \end{aligned}$$

The set S of concrete states is defined as the Cartesian product (set of pairs) of L (program labels) and M (machine states). The concrete semantics of a program, the set of all the reachable states of the program, is an element of the concrete domain D . Given a program, the L set is defined to be its finite and fixed set of labels.

Program-Label-wise Reachability In this chapter we consider a class of abstractions that come from one particular analysis goal: program-label-wise reachability. We are interested in the reachable set for each program label. For each program label we want to know the set of memories that can occur at that label during executions. Such analysis is sometimes called *flow sensitive*, because program labels are in this case assigned along the control flow of programs.

In this case, we can view the abstraction in two steps (figure 4.1). We first partition the set of states by the program labels of the states,

$$\begin{array}{ccc} \text{from collection of all states} & \xrightarrow{\quad \text{to} \quad} & \text{label-wise collection,} \\ \wp(L \times M) & \xrightarrow{\text{abstraction}} & L \rightarrow \wp(M) \end{array}$$

and then we abstract the local set of memories collected at each label into a single abstract memory,

$$\begin{array}{ccc} \text{from label-wise collection} & \text{to} & \text{label-wise abstraction.} \\ \mathbb{L} \rightarrow \wp(\mathbb{M}) & \xrightarrow{\text{abstraction}} & \mathbb{L} \rightarrow \mathbb{M}^\sharp. \end{array}$$

An element of this abstract domain is a table from each program label to an abstract memory.

The program labels are usually syntactic elements, such as those assigned to every statement and/or expression of the program (as in example 4.1 or section 2.4). For any input program we thus assume that its label set \mathbb{L} is finite and fixed before the analysis.

For this class of abstractions, what remains is to design the space \mathbb{M}^\sharp of abstract memories. The \mathbb{M}^\sharp is an abstraction of the powerset $\wp(\mathbb{M})$. This abstraction is the parameter and can be defined in many ways depending on the target properties to compute by static analysis.

Abstract Domain by Galois Connection We first design an abstract domain, a space over which the abstract semantics of programs can be finitely computable.

In this chapter, an abstract domain is a partial order that has a least element called *bottom* (written \perp) and such that each totally ordered subset (such a subset is called a *chain*) has a least upper bound. Such structure is called *CPO* (complete partial order) (see appendix A.5).

$$\begin{array}{ll}
\rho(\mathbb{L} \times \mathbb{M}) \ni & \text{collection of} \\
& \text{all states} \\
& \left\{ \begin{array}{ll} (0, m_0), (0, m'_0), \dots, & \text{at } 0 \\ (1, m_1), (1, m'_1), \dots, & \text{at } 1 \\ \vdots & \\ (n, m_n), (n, m'_n), \dots, & \text{at } n \end{array} \right. \\
\\
\mathbb{L} \rightarrow \rho(\mathbb{M}) \ni & \text{label-wise} \\
& \text{collection} \\
& \left\{ \begin{array}{l} (0, \{m_0, m'_0, \dots\}) \\ (1, \{m_1, m'_1, \dots\}) \\ \vdots \\ (n, \{m_n, m'_n, \dots\}) \end{array} \right. \\
\\
\mathbb{L} \rightarrow \mathbb{M}^\sharp \ni & \text{label-wise} \\
& \text{abstraction} \\
& \left\{ \begin{array}{l} (0, M_0^\sharp) \\ (1, M_1^\sharp) \\ \vdots \\ (n, M_n^\sharp) \end{array} \right. \\
\end{array}$$

Figure 4.1

Label-wise abstraction of states. Each M_l^\sharp over-approximates the set $\{m_l, m'_l, \dots\}$ of memories collected at label l .

Note that different structures other than CPO can also be used. As in chapter 3, abstract domains as \sqcup -semilattices also work. The generality of CPO and that of \sqcup -semilattice are not comparable.

An abstract domain needs to preserve the partial order of the concrete domain in the sense that the partial order in the abstract domain has a corresponding partial order in the concrete domain. This concept is captured by the Galois connection between the two domains. Note that solutions other than the Galois connection would work too (e.g., section 3.3). In this Chapter we choose to use the Galois connection approach.

We design an abstract domain as a CPO that is Galois connected (definition 3.5 in section 3.2.1) with the concrete domain

$$(\rho(\mathbb{L} \times \mathbb{M}), \subseteq) \xrightleftharpoons[\alpha]{\gamma} (\mathbb{L} \rightarrow \mathbb{M}^\sharp, \subseteq).$$

The abstraction function α defines how each element (a set of states) in the concrete domain is abstracted into an element in the abstract domain. The adjoined concretization function γ defines the set of concrete states implied by each abstract state. The partial order \sqsubseteq is the label-wise order:

$$a^\# \sqsubseteq b^\# \text{ iff } \forall l \in L : a^\#(l) \sqsubseteq_M b^\#(l),$$

where \sqsubseteq_M is the partial order of $M^\#$.

The above Galois connection (abstraction) can be understood as the composition of two Galois connections:

$$\begin{array}{c} (\wp(L \times M), \subseteq) \\ \xrightleftharpoons[\alpha_0]{\gamma_0} (\mathbb{L} \rightarrow \wp(M), \sqsubseteq) \quad (\sqsubseteq \text{ is the label-wise } \subseteq) \\ \xrightleftharpoons[\alpha_1]{\gamma_1} (\mathbb{L} \rightarrow M^\#, \sqsubseteq) \quad (\sqsubseteq \text{ is the label-wise } \sqsubseteq_M) \end{array}$$

The first abstraction α_0 partitions the set of states by the labels of each state and collects the memories for each label:

$$\alpha_0 \left\{ \begin{array}{l} (0, m_0), (0, m'_0), \dots, \\ (1, m_1), (1, m'_1), \dots, \\ \vdots \\ (n, m_n), (n, m'_n), \dots \end{array} \right\} = \left\{ \begin{array}{l} (0, \{m_0, m'_0, \dots\}), \\ (1, \{m_1, m'_1, \dots\}), \\ \vdots \\ (n, \{m_n, m'_n, \dots\}) \end{array} \right\}$$

This partitioning is a Galois connection.

What remains is to find a Galois connection pair α_1 and γ_1 for the second abstraction, which is a label-wise abstraction of the collected memory sets:

$$\alpha_1 \left\{ \begin{array}{l} (0, \{m_0, m'_0, \dots\}), \\ (1, \{m_1, m'_1, \dots\}), \\ \vdots \\ (n, \{m_n, m'_n, \dots\}) \end{array} \right\} = \left\{ \begin{array}{l} (0, M_0^\#), \\ (1, M_1^\#), \\ \vdots \\ (n, M_n^\#) \end{array} \right\}$$

Thus, defining this second Galois connection pair boils down to defining a Galois connection pair between the powerset of memories and an abstract memory domain:

$$(\wp(\mathbb{M}), \subseteq) \xrightleftharpoons[\alpha_M]{\gamma_M} (\mathbb{M}^\sharp, \sqsubseteq_M)$$

Notations Before we continue, we briefly define the notations used in the rest of this chapter.

- An element of $A \rightarrow B$, which is a map from A to B , is interchangeably an element in $\wp(A \times B)$. For example, an element in $L \rightarrow M^\sharp$ of the abstract states is interchangeably an element in $\wp(L \times M^\sharp)$, a set (so-called *graph*) of pairs of labels and abstract memories. Note that in the above examples of this subsection we already used this graph notation to represent the abstract state function.
- A relation $f \subseteq A \times B$ is interchangeably a function $f \in A \rightarrow \wp(B)$ defined as

$$f(a) = \{b \mid (a, b) \in f\}.$$

For example, the concrete one-step transition relation $\hookrightarrow \subseteq S \times S$ is interchangeably a function $\hookrightarrow \in S \rightarrow \wp(S)$.

- For function $f : A \rightarrow B$, we write $\wp(f)$ for its powerset version, defined as

$$\begin{aligned}\wp(f) &: \wp(A) \rightarrow \wp(B) \\ \wp(f)(X) &= \{f(x) \mid x \in X\}.\end{aligned}$$

- For function $f : A \rightarrow \wp(B)$, we write $\check{\wp}(f)$ as a shorthand for $\cup \circ \wp(f)$:

$$\begin{aligned}\check{\wp}(f) &: \wp(A) \rightarrow \wp(B) \\ \check{\wp}(f)(X) &= \bigcup \{f(x) \mid x \in X\}\end{aligned}$$

For example, powerset-lifted function $Step : \wp(S) \rightarrow \wp(S)$ of relation \hookrightarrow ,

$$Step(X) = \{s' \mid s \hookrightarrow s', s \in X\},$$

is equivalently, by regarding \hookrightarrow as a function of $S \rightarrow \wp(S)$,

$$Step = \check{\wp}(\hookrightarrow).$$

- For functions $f : A \rightarrow B$ and $g : A' \rightarrow B'$, we write (f, g) for

$$(f, g) : A \times A' \rightarrow B \times B'$$

$$(f, g)(a, a') = (f(a), g(a')).$$

4.2.2 Abstraction of Semantic Functions

The abstract semantic function $F^\#$ over the abstract state is defined as follows.

Given a concrete semantic function F ,

$$\begin{aligned} S &= L \times M \\ F : \wp(S) &\rightarrow \wp(S) \\ F(X) &= I \cup Step(X), \end{aligned}$$

where

$$\begin{aligned} Step &= \check{\wp}(\hookrightarrow) \quad (\text{relation } \hookrightarrow \text{ as a function}), \\ \hookrightarrow &\subseteq (L \times M) \times (L \times M), \end{aligned}$$

its abstract version is defined to be

$$\begin{aligned} S^\# &= L \rightarrow M^\# \\ F^\# : S^\# &\rightarrow S^\# \\ F^\#(X^\#) &= \alpha(I) \cup^\# Step^\#(X^\#), \end{aligned}$$

where

$$\begin{aligned} Step^\# &= \wp(id, \cup_M^\#) \circ \pi \circ \check{\wp}(\hookrightarrow^\#) \quad (\text{relation } \hookrightarrow^\# \text{ as a function}), \\ \hookrightarrow^\# &\subseteq (L \times M^\#) \times (L \times M^\#). \end{aligned} \tag{4.2}$$

The $Step^\#$ function is the one-step transition function over the abstract states $L \rightarrow M^\#$. The reason for its definition (4.2) is as follows:

- $\check{\wp}(\hookrightarrow^\#)$: To an abstract state in $L \rightarrow M^\#$ as a set $\subseteq L \times M^\#$, it applies the abstract transition function $\hookrightarrow^\#$ to each element and collects the results, returning a set $\subseteq L \times M^\#$.
- $\pi \circ \check{\wp}(\hookrightarrow^\#)$: The operator π partitions the result $\subseteq L \times M^\#$ of $\check{\wp}(\hookrightarrow^\#)$ by the labels in L , returning a set $\subseteq L \times \wp(M^\#)$, where each label has only one pair, a set representation of an element in $L \rightarrow \wp(M^\#)$.

- $\wp(\text{id}, \cup_M^\#) \circ \pi \circ \check{\wp}(\rightarrow^\#)$: To the result $\subseteq L \times \wp(M^\#)$ of $\pi \circ \check{\wp}(\rightarrow^\#)$, applying $\wp(\text{id}, \cup_M^\#)$ returning a set $\subseteq L \times M^\#$, in effect an abstract state $\in L \rightarrow M^\#$, so that each label is to be paired with a single abstract memory. This single abstract memory is a sound approximation $\cup_M^\#$ of the set of abstract memories at each label.

Example 4.4 (One-step transition Step $^\#$) Suppose the program has two labels, l_1 and l_2 . That is, $L = \{l_1, l_2\}$. Given an abstract state $\{(l_1, M_1^\#), (l_2, M_2^\#)\}$, Step $^\#$ first applies $\check{\wp}(\rightarrow^\#)$ to it:

$$\rightarrow^\#(l_1, M_1^\#) \cup \rightarrow^\#(l_2, M_2^\#)$$

Suppose that $\rightarrow^\#(l_1, M_1^\#)$ returns $\{(l_1, M'_1^\#), (l_2, M''_1^\#)\}$ and that $\rightarrow^\#(l_2, M_2^\#)$ returns $\{(l_1, M'_2^\#)\}$. Then the result is

$$\{(l_1, M'_1^\#), (l_2, M''_1^\#), (l_1, M'_2^\#)\}.$$

The subsequent application of the operator π partitions the result by labels into

$$\{(l_1, \{M'_1^\#, M'_2^\#\}), (l_2, \{M''_1^\#\})\}.$$

The final organization operation $\wp(\text{id}, \cup_M^\#)$ returns the post abstract state $\in L \rightarrow M^\#$:

$$\{(l_1, M'_1^\# \cup_M^\# M'_2^\#), (l_2, M''_1^\#)\}$$

Conditions for Sound $\hookrightarrow^\#$, $\cup^\#$, and $\cup_M^\#$ The abstract one-step transition relation $\rightarrow^\#$ must satisfy, as a function,

$$\check{\wp}(\rightarrow) \circ \gamma \subseteq \gamma \circ \check{\wp}(\rightarrow^\#).$$

Figure 4.2 depicts the above condition in a diagram. The condition is natural: the abstract one-step transition relation $\rightarrow^\#$ must cover the cases of the corresponding concrete one-step relation \rightarrow in the concrete semantics. Note that abstract state $X^\# \in L \rightarrow M^\#$ is considered a set $\in L \times M^\#$ as the argument for $\check{\wp}(\rightarrow^\#)$.

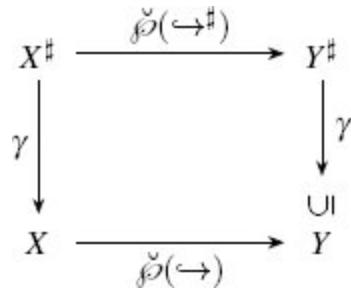


Figure 4.2

Sound one-step abstract transition \hookrightarrow^\sharp

In the same vein, the condition for the abstract union operators \cup^\sharp and \cup_M^\sharp to be sound is

$$\cup \circ (\gamma_-, \gamma_-) \subseteq \gamma_- \circ \cup_-^\sharp.$$

All the above postulates (concrete semantic domains, concrete semantic function F , Galois connected abstract domains, abstract semantic function F^\sharp , sound \hookrightarrow^\sharp , sound \cup^\sharp , and sound \cup_M^\sharp) contribute to the correctness of the resulting abstract semantics.

4.2.3 Recipe for Defining an Abstract Transition Semantics

In summary, the recipe of achieving a sound static analysis based on the transitional semantics is as follows. Such static analysis over-approximates the concrete semantics of the input programs.

1. Define M to be the set of memory states that can occur during program executions. Let L be the finite and fixed set of labels of a given program.
2. Define a concrete semantics as the $\text{lfp } F$, where

$$\begin{aligned}
 \text{concrete domain} \quad \wp(S) &= \wp(L \times M), \\
 \text{concrete semantic function} \quad F : \wp(S) &\rightarrow \wp(S) \\
 F(X) &= I \cup Step(X), \\
 Step &= \check{\wp}(\hookrightarrow), \\
 \hookrightarrow &\subseteq (L \times M) \times (L \times M).
 \end{aligned}$$

The \hookrightarrow is the one-step transition relation over $L \times M$.

3. Define its abstract domain and abstract semantic function as follows:

$$\begin{aligned}
\text{abstract domain} \quad S^\# &= L \rightarrow M^\# \\
\text{abstract semantic function} \quad F^\# : S^\# &\rightarrow S^\# \\
F^\#(X^\#) &= \alpha(I) \cup^\# Step^\#(X^\#) \\
Step^\# &= \check{\rho}(id, \cup_M^\#) \circ \pi \circ \check{\rho}(\hookrightarrow^\#) \\
\hookrightarrow^\# &\subseteq (L \times M^\#) \times (L \times M^\#)
\end{aligned}$$

The $\hookrightarrow^\#$ is the one-step abstract transition relation over $L \times M^\#$.

Function π partitions a set $\subseteq L \times M^\#$ by the labels in L , returning an element in $L \rightarrow \rho(M^\#)$ represented as a set $\subseteq L \times \rho(M^\#)$.

4. Check the abstract domains $S^\#$ and $M^\#$ are CPOs and form a Galois connection, respectively, with $\rho(S)$ and $\rho(M)$:

$$(\rho(S), \subseteq) \xleftarrow[\alpha]{\gamma} (S^\#, \sqsubseteq) \quad \text{and} \quad (\rho(M), \subseteq) \xleftarrow[\alpha_M]{\gamma_M} (M^\#, \sqsubseteq_M),$$

where the partial order \sqsubseteq of $S^\#$ is label-wise \sqsubseteq_M :

$$a^\# \sqsubseteq b^\# \quad \text{iff} \quad \forall l \in L : a^\#(l) \sqsubseteq_M b^\#(l)$$

5. Check the abstract one-step transition $\hookrightarrow^\#$ and abstract unions $\cup^\#$ and $\cup_M^\#$ satisfy

$$\begin{aligned}
\check{\rho}(\hookrightarrow) \circ \gamma &\subseteq \gamma \circ \check{\rho}(\hookrightarrow^\#), \\
\cup \circ (\gamma_-, \gamma_-) &\subseteq \gamma_- \circ \cup_-^\#.
\end{aligned}$$

6. Then sound static analysis can be defined as follows:

- (a) (theorem 4.2) If $S^\#$ is of finite height (every chain is finite) and $F^\#$ is monotone or extensive, then

$$\bigsqcup_{i \geq 0} F^{\#i}(\perp)$$

is finitely computable and over-approximates the concrete semantics $\mathbf{lfp}F$.

(b) (theorem 4.3) Otherwise, find a widening operator ∇ (definition 3.11); then the following chain $X_0 \sqsubseteq X_1 \sqsubseteq \dots$

$$X_0 = \perp \quad X_{i+1} = X_i \nabla F^\#(X_i)$$

is finite and its last element over-approximates the concrete semantics $\text{lfp } F$.

Theorem 4.2 (Sound static analysis by $F^\#$) Given a program, let F and $F^\#$ be defined as in section 4.2.3. If $S^\#$ is of finite height (every chain $S^\#$ is finite) and if $F^\#$ is monotone or extensive, then

$$\bigsqcup_{i \geq 0} F^\#^i(\perp)$$

is finitely computable and over-approximates $\text{lfp } F$:

$$\text{lfp } F \subseteq \gamma(\bigsqcup_{i \geq 0} F^\#^i(\perp)) \quad \text{or, equivalently,} \quad \alpha(\text{lfp } F) \sqsubseteq \bigsqcup_{i \geq 0} F^\#^i(\perp)$$

The proof of the above theorem is in appendix B.3.1.

In case the abstract domain may have an infinite chain or the abstract semantic function $F^\#$ can be neither monotone nor extensive, by means of a special operator called *widening operator* we can still finitely compute an upper approximation of the concrete semantics of programs, as follows.

Theorem 4.3 (Sound static analysis by $F^\#$ and widening operator ∇) Given a program, let F and $F^\#$ be defined as in section 4.2.3. Let ∇ be a widening operator as defined in definition 3.11. Then the following chain $Y_0 \sqsubseteq Y_1 \sqsubseteq \dots$

$$Y_0 = \perp \quad Y_{i+1} = Y_i \nabla F^\dagger(Y_i)$$

is finite, and its last element Y_{\lim} over-approximates $\text{lfp } F$:

$$\text{lfp } F \subseteq \gamma(Y_{\lim}) \quad \text{or, equivalently,} \quad \alpha(\text{lfp } F) \sqsubseteq Y_{\lim}$$

The proof of the above theorem is in appendix B.3.2.

Note that if a widening operator is used with a monotone or extensive $F^\#$, the second condition for a widening operator in definition 3.11 can be relaxed from “for all sequences $(a_n)_{n \in \mathbb{N}}$ ” to “for all chains $a_0 \sqsubseteq a_1 \sqsubseteq \dots$ ”.

Using the Results of the Analysis Like theorem 3.6 for compositional-style abstract semantics, the above two soundness theorems formalize what the

analysis achieves: the analysis computes an over-approximation of *all* the states that the program may reach during its executions.

The same discussion we had after theorem 3.6 regarding the use of a sound analysis in practice applies here too. If the over-approximate reachable set does not intersect with the set of error states, then we are sure that the program will not generate an error state. Otherwise, we cannot conclude anything. A non-empty intersection may be due to an imprecision of the analysis, or to the fact that the program can indeed generate an error state. Upon this unsettled case, the analysis has to generate *alarms* so that users should inspect the analysis results so as to decide whether the alarms are true or not. This so-called *triage* process is discussed in detail in section 6.3.

4.3 Analysis Algorithms Based on Global Iterations

4.3.1 Basic Algorithms

Once we have designed an abstract semantics function $F^\#$ as in the recipe (section 4.2.3), the analysis implementation is straightforward from theorems 4.2 and 4.3.

Algorithm from Theorem 4.2 If the abstract domain $S^\#$ is of finite height and if $F^\#$ is monotone or extensive, the increasing chain

$$\perp \sqsubseteq (F^\#)^1(\perp) \sqsubseteq (F^\#)^2(\perp) \sqsubseteq \dots$$

is finite, and its biggest element is equal to

$$\bigsqcup_{i \geq 0} F^{\#i}(\perp).$$

Hence, the analysis algorithm is a simple loop, shown in figure 4.3.

```

C ← ⊥
repeat
    R ← C
    C ← F#(C)
until C ⊑ R
return R

```

Figure 4.3

Algorithm without widening

As a side note, the algorithms in the gentle introduction of section 2.4.4 can be rephrased as the following algorithm that computes an upper bound of $\sqcup_{i \geq 0} F^{\#^i}(\perp)$. Instead of starting the iteration from $I^{\#}$ as in section 2.4.4 the $I^{\#}$ is always included when we compute the body $F^{\#}$:

```

C ← ⊥
repeat
    R ← C
    C ← C ∪# F#(C)
until C ⊑ R
return R

```

Note that this algorithm computes a chain

$$Y_0 = \perp \quad Y_{i+1} = Y_i \cup^{\#} F^{\#}(Y_i)$$

such that every element Y_i is an upper bound of its corresponding element X_i of the first algorithm,

$$X_0 = \perp \quad X_{i+1} = F^{\#}(X_i).$$

Algorithm from Theorem 4.3 If the abstract domain $S^{\#}$ is of infinite height, or if $F^{\#}$ is neither monotonic nor extensive, we have to use a widening operator ∇ . Even when the abstract domain $S^{\#}$ is of finite height, we can accelerate the analysis steps by using the widening operator.

Given a program, the analysis algorithm with ∇ is shown in figure 4.4. The algorithm computes a finite increasing chain as ensured in theorem 4.3.

4.3.2 Worklist Algorithm

The basic iteration algorithms of section 4.3.1 have a room for speedup. Let us reconsider the widening version with the operation $F^\#(C)$ being inlined in order to expose its

```

C ← ⊥
repeat
    R ← C
    C ← C  $\nabla F^\#(C)$ 
until C ⊑ R
return R

```

Figure 4.4

Algorithm with ∇

performance bottleneck:

```

C ← ⊥
repeat
    R ← C
    C ← C  $\nabla \underbrace{(\rho(\text{id}, \cup_M^\#) \circ \pi \circ \check{\rho}(\rightarrow^\#))(C)}_{F^\#}$ 
until C ⊑ R
return R

```

Note that, at each iteration, computing

$$\check{\rho}(\rightarrow^\#)(C)$$

applies the abstract transition operation $\rightarrow^\#$ to the state at every label in the program. The table C has as many entries as the number of labels in the program. When every statement of the program is uniquely labeled, a one-million-statement program has one million labels.

We can speed up the performance by reducing the program labels to visit at each iteration. We keep a worklist, a set of labels for which the transition needs to be applied because its input memories were changed. For each iteration, the transition is applied only for those labels in the worklist. The worklist for the next iteration comes to consist of the labels whose input memories are changed in the previous iteration.

This worklist version is in figure 4.5. As for notation, for the table C from all labels in the program to their abstract memories, $C|_{\text{WorkList}}$ is the same as the C table but restricted only for labels in WorkList .

This worklist algorithm calls for improvements regarding two specific points:

- Note that collecting the new worklist for next iteration

$$\text{WorkList} \leftarrow \{l \mid C(l) \not\subseteq R(l), l \in L\}$$

rescans all the labels of the program. We can avoid this scanning of all the labels. Right after each application $\hookrightarrow^\#$ to $(l, C(l))$, if the result state $(l', M^\#)$ is changed ($M^\# \not\subseteq C(l')$), we add l' to the new worklist.

```

 $C : \mathbb{L} \rightarrow M^\sharp$ 
 $F^\sharp : (\mathbb{L} \rightarrow M^\sharp) \rightarrow (\mathbb{L} \rightarrow M^\sharp)$ 
 $\text{WorkList} : \wp(\mathbb{L})$ 

 $\text{WorkList} \leftarrow \mathbb{L}$ 
 $C \leftarrow \perp$ 
repeat
   $R \leftarrow C$ 
   $C \leftarrow C \nabla F^\sharp(C|_{\text{WorkList}})$  (* only for WorkList *)
   $\text{WorkList} \leftarrow \{l \mid C(l) \not\subseteq R(l), l \in \mathbb{L}\}$  (* next WorkList *)
until  $\text{WorkList} = \emptyset$ 
return  $R$ 

```

Figure 4.5

Analysis algorithm with worklist and widening

- Generally, the widening operator deteriorates the precision of resulting abstract elements; thus, they should be applied only as necessity. Naive implementation of the widening operation $C \nabla F^\sharp(C|_{\text{WorkList}})$ would be a squandering, label-wise widening:

for every $(l, M^\sharp) \in F^\sharp(C|_{\text{WorkList}})$ we widen $C(l)$ by M^\sharp

The precision would be better if we apply the widening operation only when the l is the target of a cycling control flow (e.g., the l -labeled statement is a **while** statement or a target statement of a cycling **goto**) [13]. For other labels, we apply the upper bound operation U^\sharp instead.

4.4 Use Example of the Framework

4.4.1 Simple Imperative Language

We consider a simple imperative language (figure 4.6) that is almost identical to the one in chapter 3 on the compositional-style framework. One difference, in order to expose the merit of the transitional style, is the goto statement whose target label is not fixed in the program text but to be computed during execution by its argument expression. Labels in a program are integers that are uniquely assigned to every statement of the program; we assume that the programmers know the labels for each statement of their programs. The expressions compute values without mutating the memory. An expression E computes an integer or a program label. A Boolean expression B computes a Boolean value.

| | |
|------------------------|--------------------------------------|
| $x \in \mathbb{X}$ | program variables |
| $C ::=$ | statements |
| skip | no-op statement |
| $C ; C$ | sequence of statements |
| $x := E$ | assignment |
| input(x) | read an integer input |
| if(B){C}else{C} | condition statement |
| while(B){C} | loop statement |
| goto E | goto with dynamically computed label |
| $E ::=$ | expression |
| n | integer |
| x | variable |
| $E + E$ | addition |
| $B ::=$ | Boolean expression |
| true false | |
| $E < E$ | comparison |
| $E = E$ | equality |
| $P ::= C$ | program |

Figure 4.6

Syntax of a simple imperative language

Program Labels and Execution Order Given a program, each of its statements has a unique label as a natural number. For example, figure 4.7 shows an example program where a unique label is associated to each statement.

Except for **goto** E , the execution order (or *control flow*) between the statements in a program is clear from the program syntax. The function $\langle\!\langle C, l' \rangle\!\rangle$ defined below collects all the execution orders available from syntax between the statement labels in C . The label l' is the next label to continue after executing C . Thus, given a program p and label l_{end} for the end label of the program, $\langle\!\langle p, l_{\text{end}} \rangle\!\rangle$ collects the function graphs of **next**, **nextTrue**, and **nextFalse**. We write **label**(C) for the label of statement C .

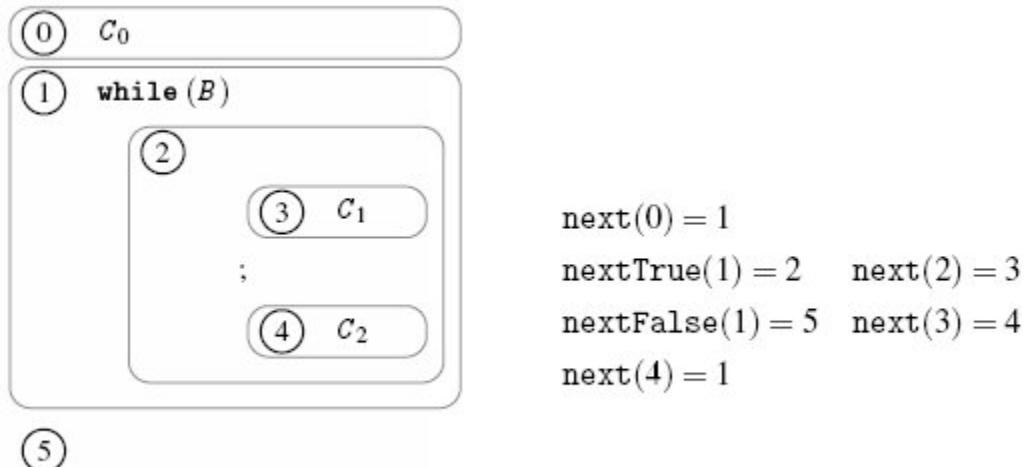


Figure 4.7

Example program with statement labels and execution order (all labels are statically known)

```

 $\langle\langle C, l' \rangle\rangle = \text{case } C \text{ of} \quad (* \text{ let } l \text{ be label}(C) *)$ 
    skip : {next}(l) = l'
    x := E : {next}(l) = l'
    input(x) : {next}(l) = l'
     $C_1; C_2$  : {next}(l) = label( $C_1$ )  $\cup$   $\langle\langle C_1, \text{label}(C_2) \rangle\rangle \cup \langle\langle C_2, l' \rangle\rangle$ 
    if(B){C}_1 \text{else}{C}_2 : {nextTrue}(l) = label( $C_1$ ), {nextFalse}(l) = label( $C_2$ )
                                 $\cup \langle\langle C_1, l' \rangle\rangle \cup \langle\langle C_2, l' \rangle\rangle$ 
    while(B){C} : {nextTrue}(l) = label( $C$ ), {nextFalse}(l) = l'  $\cup \langle\langle C, l \rangle\rangle$ 
    goto E : {}      (* to be determined at run-time by evaluating E *)

```

4.4.2 Concrete State Transition Semantics

Given a program p , let X be the finite set of its variables. Each statement of the program is uniquely labeled, and we assume that the **next**, **nextTrue**, and **nextFalse** functions are syntactically computed beforehand by $\langle\langle p, l_{\text{end}} \rangle\rangle$, except for the goto case.

Then the concrete semantics of the program, for the set I of input states, is the least fixpoint

lfp F

of the continuous function

$$\begin{aligned} F : \wp(\mathbb{S}) &\rightarrow \wp(\mathbb{S}) \\ F(X) &= I \cup \text{Step}(X) \\ \text{Step}(X) &= \check{\wp}(\rightarrow). \end{aligned}$$

The set of states is the set of label-and-memory pairs

$$S = L \times M,$$

where

$$\begin{aligned} \text{memories } M &= X \rightarrow V, \\ \text{values } V &= Z \cup L. \end{aligned}$$

The state transition relation $(l, m) \hookrightarrow (l', m')$ is defined as follows. The transition relation is defined by case analysis on statement labeled by l :

| | |
|---|--|
| skip | $: (l, m) \xrightarrow{} (\text{next}(l), m)$ |
| input(x) | $: (l, m) \xrightarrow{} (\text{next}(l), \text{update}_x(m, z))$ for an input integer z |
| $x := E$ | $: (l, m) \xrightarrow{} (\text{next}(l), \text{update}_x(m, \text{eval}_E(m)))$ |
| $C_1; C_2$ | $: (l, m) \xrightarrow{} (\text{next}(l), m)$ |
| if(B)$\{C_1\}$else$\{C_2\}$ | $: (l, m) \xrightarrow{} (\text{nextTrue}(l), \text{filter}_B(m))$ |
| | $: (l, m) \xrightarrow{} (\text{nextFalse}(l), \text{filter}_{\neg B}(m))$ |
| while(B)$\{C\}$ | $: (l, m) \xrightarrow{} (\text{nextTrue}(l), \text{filter}_B(m))$ |
| | $: (l, m) \xrightarrow{} (\text{nextFalse}(l), \text{filter}_{\neg B}(m))$ |
| goto E | $: (l, m) \xrightarrow{} (\text{eval}_E(m), m)$ |

The memory update operation $\text{update}_x(m, v)$ returns a new memory that is the same as m except that its image for x is v . The expression-evaluation operation $\text{eval}_E(m)$ returns a value of expression E given memory m . The $\text{filter}_B(m)$ (resp., $\text{filter}_{\neg B}(m)$) operation returns m if the value of Boolean expression E for m is true (resp., false). Otherwise, no corresponding transition relation happens.

4.4.3 Abstract State

An abstract domain $M^\#$ is a CPO such that

$$(\wp(M), \subseteq) \xrightleftharpoons[\alpha_M]{\cong} (M^\#, \sqsubseteq_M).$$

We define an abstract memory $M^\#$ for a set of memories as a single map from program variables to abstract values:

$$M^\# \in M^\# = X \rightarrow V^\#,$$

where $V^\#$ is an abstract domain that is a CPO such that

$$(\wp(V), \subseteq) \xrightleftharpoons[\alpha_V]{\cong} (V^\#, \sqsubseteq_V).$$

We design $V^\#$ as

$$V^\# = \mathbb{Z}^\# \times L^\#$$

where $\mathbb{Z}^\#$ is a CPO that is Galois connected with $\wp(\mathbb{Z})$, and $L^\#$ is the powerset $\wp(L)$ of labels.

Note that all abstract domains are Galois connected CPOs. Because $\mathbb{Z}^\#$ and $L^\#$ are Galois connected CPOs, so are the compound domains built from them: $V^\#$, the component-wise-ordered pairs of $\mathbb{Z}^\#$ and $L^\#$; $M^\#$, the point-wise-ordered vectors of $V^\#$; and $S^\#$, the point-wise-ordered vectors of $M^\#$.

4.4.4 Abstract State Transition Semantics

For an abstract memory $M^\#$, we define the abstract state transition relation $(l, M^\#) \xrightarrow{\#} (l', M'^\#)$ as follows.

Case: The l -labeled statement of

$$\begin{aligned}
 \text{skip} &: (l, M^\#) \xrightarrow{\#} (\text{next}(l), M^\#) \\
 \text{input}(x) &: (l, M^\#) \xrightarrow{\#} (\text{next}(l), \text{update}_x^\#(M^\#, \alpha(\mathbb{Z}))) \\
 x := E &: (l, M^\#) \xrightarrow{\#} (\text{next}(l), \text{update}_x^\#(M^\#, \text{eval}_E^\#(M^\#))) \\
 C_1; C_2 &: (l, M^\#) \xrightarrow{\#} (\text{next}(l), M^\#) \\
 \text{if}(B)\{C_1\} \text{else}\{C_2\} &: (l, M^\#) \xrightarrow{\#} (\text{nextTrue}(l), \text{filter}_B^\#(M^\#)) \\
 &\quad : (l, M^\#) \xrightarrow{\#} (\text{nextFalse}(l), \text{filter}_{\neg B}^\#(M^\#)) \\
 \text{while}(B)\{C\} &: (l, M^\#) \xrightarrow{\#} (\text{nextTrue}(l), \text{filter}_B^\#(M^\#)) \\
 &\quad : (l, M^\#) \xrightarrow{\#} (\text{nextFalse}(l), \text{filter}_{\neg B}^\#(M^\#)) \\
 \text{goto } E &: (l, M^\#) \xrightarrow{\#} (l', M^\#) \quad \text{for } l' \in L \text{ of } (z^\#, L) = \text{eval}_E^\#(M^\#)
 \end{aligned}$$

Let $F^\#$ be defined as the framework:

$$\begin{aligned}
 F^\# : \mathbb{S}^\# &\rightarrow \mathbb{S}^\# \\
 F^\#(S^\#) &= \alpha(I) \cup^\# \text{Step}^\#(S^\#) \\
 \text{Step}^\# &= \wp(\text{id}, \cup_M^\#) \circ \pi \circ \check{\wp}(\rightarrow^\#)
 \end{aligned}$$

If $\text{Step}^\#$ and $\cup^\#$ are sound abstractions of, respectively, Step and \cup , as required by the framework

$$\begin{aligned}
 \check{\wp}(\rightarrow) \circ \gamma &\subseteq \gamma \circ \check{\wp}(\rightarrow^\#), \\
 \cup \circ (\gamma_-, \gamma_-) &\subseteq \gamma_- \circ \cup_-^\#,
 \end{aligned}$$

then we can use $F^\#$ to soundly approximate the concrete semantics $\mathbf{lfp}F$ (theorems 4.2 and 4.3), and the approximation is finitely computable as in the algorithms in section 4.3.

Defining Sound $\hookrightarrow^\#$ Note that the abstract transition relation $\hookrightarrow^\#$ is the same as \hookrightarrow except that it uses the abstract correspondents for semantic operators: operator $eval^\#_E$ for $eval_E$, $update^\#_X$ for $update_X$, $filter^\#_B$ for $filter_B$, and $filter^\#_{\neg B}$ for $filter_{\neg B}$.

If each of the abstract semantic operators is a sound abstraction of its concrete correspondent, then $\hookrightarrow^\#$ is a sound abstraction of \hookrightarrow , as follows.

Theorem 4.4 (Soundness of $\hookrightarrow^\#$) Consider the concrete one-step transition relation of section 4.4.2 and the abstract transition relation of section 4.4.4. If the semantic operators satisfy the soundness properties

$$\begin{aligned}\wp(eval_E) \circ \gamma_M &\subseteq \gamma_N \circ eval_E^\#, \\ \wp(update_X) \circ \times \circ (\gamma_M, \gamma_N) &\subseteq \gamma_M \circ update_X^\#, \\ \wp(filter_B) \circ \gamma_M &\subseteq \gamma_M \circ filter_B^\#, \\ \wp(filter_{\neg B}) \circ \gamma_M &\subseteq \gamma_M \circ filter_{\neg B}^\#, \end{aligned}$$

then $\check{\wp}(\hookrightarrow) \circ \gamma \sqsubseteq \gamma \circ \check{\wp}(\hookrightarrow^\#)$. (The \times is the Cartesian product operator of two sets.)

The proof is included in appendix B.3.

Defining Sound $U^\#$ and $\cup_M^\#$ For a sound $U^\#$, one candidate is the least-upper-bound operator \sqcup if $S^\#$ is closed by \sqcup , because

$$\begin{aligned}(\gamma \circ \sqcup)(a^\#, b^\#) &= \gamma(a^\# \sqcup b^\#) \supseteq \gamma(a^\#) \cup \gamma(b^\#) \quad \text{by the monotonicity of } \gamma \\ &= (\cup \circ (\gamma, \gamma))(a^\#, b^\#).\end{aligned}$$

By the same reason, if $M^\#$ is closed by the least-upper-bound operator \sqcup_M , then we can use \sqcup_M for a sound $\cup_M^\#$.

Adapting the Analysis for a Different Abstraction Soundness theorem 4.4 also suggests a way to adapt the analysis for a different abstraction. We follow the recipe of section 4.2.3, and for a different analysis we need to use different abstract domains and semantic operators. The new analysis is sound if the abstract transition relation $\hookrightarrow^\#$ is the same as \hookrightarrow except that it uses the abstract correspondents for semantic operators, and that each

abstract semantic operator $f^\# : A^\# \rightarrow B^\#$ satisfies the pattern $\rho(f) \circ \gamma_A \subseteq \gamma_B \circ f^\#$.

In fact, many static analysis tools are parametric in the choice of the abstract domains and semantic operators. The practical considerations related to the parameter setting of the analysis are discussed in section 6.2.

5 Advanced Static Analysis Techniques

Goal of This Chapter This chapter completes chapters 3 and 4 with some advanced concepts that play an important role in program analysis. A very large number of advanced analysis techniques have been introduced, and it is not possible to describe or even summarize them all in a single book; therefore, we focus on the most important ones. The techniques described in this chapter allow building more general, efficient, and precise static analyzers.

Recommended Reading: [S], [D] (2nd reading), [U] (2nd reading) Some readers may want to skip this chapter during the first reading and come back to it when they need the techniques it introduces. In particular, readers motivated by the practical use of static analysis tools should read more practical material in chapter 6 and refer to this chapter later when the need to understand specific techniques arises.

Chapter Outline We will consider three important areas in static analysis:

- **Abstract domains:** The choice of the abstraction is one of the most crucial steps in the design of a static analysis since it fixes the set of logical predicates that the analysis may use. In chapter 3 we introduced only a few basic instances and mainly focused on the use of non-relational domains; thus, we present in section 5.1 more advanced ways to construct powerful abstract domains. In this chapter, we study only general techniques to construct abstract domain; abstraction for specific features is discussed in chapter 8.
- **Iteration techniques:** Static analyzers rely on iterative techniques to compute program invariants, and chapter 3 presented only the most basic ways to achieve this. Yet, these iterative techniques have a great influence on both the precision and the cost of the analysis. The design space for widening operators and algorithms to solve the iterative equations is large, and a design choice may greatly contribute to

improving a given analysis. Therefore, we discuss such refinements in section 5.2.

- **Scalability techniques:** Even when efficient iteration techniques are used, static analysis tools may have significant room for efficiency improvement. In particular, when a static analyzer computes a fixpoint as a table from program labels to abstract memory states (as in chapter 4), the memory consumption during the fixpoint iterations is often the bottleneck for the scalability. In section 5.3, we discuss *sparse analysis* techniques to exploit the sparsity of abstract semantics to reduce the time and space consumptions of the induced analyses. In section 5.4, we also discuss *modular analysis* techniques for modular analysis, which allow analyzing program fragments (e.g., procedures) separately and combining the modular results into the final analysis result.
- **Analysis direction:** In chapter 3, we showed *forward* abstract interpretation, which computes abstract post-conditions from abstract pre-conditions, which is the most common static analysis pattern, in order to infer effects from causes. Another useful pattern computes abstract pre-conditions from abstract post-conditions, so as to derive the causes of some observable effects. We study this approach in section 5.5.

In this chapter, we stick (as much as possible) to the notations fixed in chapters 3 and 4.

5.1 Construction of Abstract Domains

The selection of an adequate abstract domain is at the foundation of the design of any static analysis tool, as it fixes the logical predicates that the analysis may use. In section 3.2, we presented a few common numerical abstract domains. However, real-world static analyses require far more sophisticated abstract domains that are able to express more complex properties. The goal of this section is to study a few techniques to construct such abstract domains.

We show in section 5.1.1 how to describe states with Boolean variables as well as numerical variables. In sections 5.1.2 and 5.1.3, we discuss how to construct domains to express composite logical properties, such as conjunctions or disjunctions of basic properties. Finally, we provide guidelines for the construction of abstract domains in section 5.1.4.

However, note that we cannot cover the abstraction of all possible data types in this section. In particular, we defer complex data structures to section 8.3.

5.1.1 Abstraction of Boolean-Numerical Properties

In chapters 3 and 4, we considered the analysis of languages where all variables have the same type and are all scalars. This is unrealistic, as real programming languages support many data types, including Booleans, strings, pointers, and so on. We now discuss the abstraction of *heterogeneous states*, with both variables of numerical type and variables of Boolean type. We consider other more complex types in chapter 8.

We let x_0, x_1, \dots denote numerical variables, and we denote the Boolean variables by b_0, b_1, \dots . We show an example program in figure 5.1(a) that is rather contrived but will help demonstrate two Boolean abstractions in the following paragraphs. Figure 5.1(b) shows the set of states that can be observed at the end of this program. We remark that in this set the variables x_0 and x_1 have the same sign and that both b_0 and b_1 store the value **true** if and only if x_0 is negative.

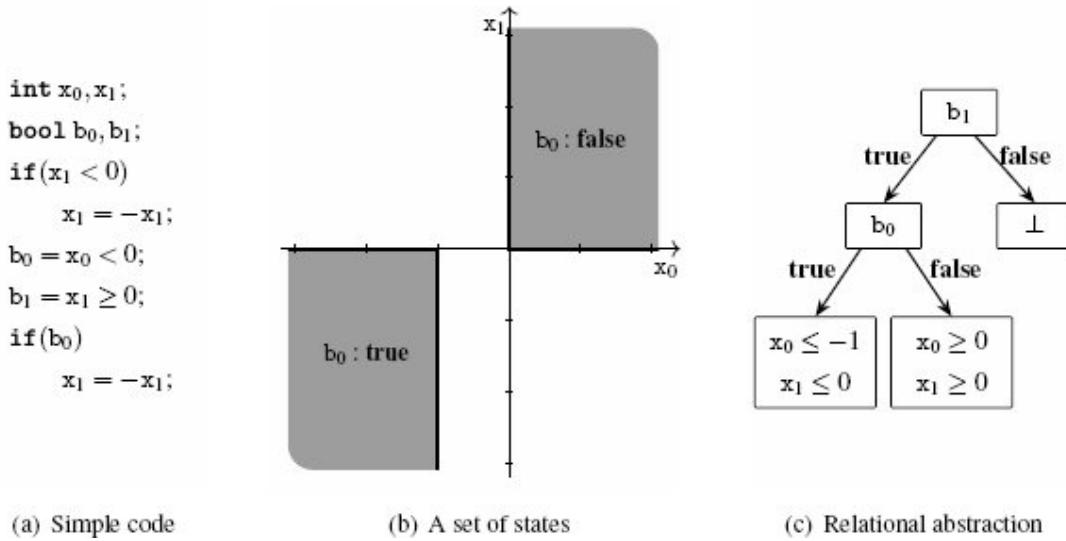


Figure 5.1

Boolean-numerical program and abstraction

Non-Relational Abstraction As we showed in section 3.2.2, the principle of non-relational abstraction is to describe the value of a variable separately from the other variables. To extend this technique to variables of Boolean type, we simply have to specify an abstraction for sets of Boolean values. A common such abstraction is provided by the lattice $A_B = \{\perp, \text{true}, \text{false}, \top\}$, with concretization function γ_B defined by

$$\gamma_B(\perp) = \emptyset \quad \gamma_B(\text{true}) = \{\text{true}\} \quad \gamma_B(\text{false}) = \{\text{false}\} \quad \gamma_B(\top) = \{\text{true}, \text{false}\}$$

Extending the analyses of chapters 3 and 4 to this domain is fairly straightforward and results in the computation of the following invariants.

Example 5.1 (Non-relational Boolean abstraction) Assuming that numerical variables are abstracted using intervals (figure 3.5(c)) and that Boolean variables are abstracted using the above lattice, the compositional analysis of the program of figure 5.1(a) produces the following invariants:

$$b_0 \mapsto \top \quad b_1 \mapsto \text{true} \quad x_0 \mapsto \top \quad x_1 \mapsto \top$$

This abstraction is obviously not precise and captures no relation between b_0, x_0 , and x_1 .

Relational Abstraction To remedy the imprecision uncovered in example 5.1, we need an abstract domain that captures relations between Boolean and numerical variables. In this program, Boolean variables store the result of numerical comparisons and affect control flow later on. Intuitively, the relation between Boolean and numerical variables can be expressed with basic decision trees of the following form, where C_0 and C_1 are basic numerical constraints:

$$\begin{array}{ll} \text{if } b \text{ contains true} & \text{then numerical condition } C_0 \text{ holds} \\ \text{otherwise} & \text{then numerical condition } C_1 \text{ holds} \end{array}$$

Such predicates can be encoded as *decision trees* over program variables [12, 53, 54] with techniques similar to *binary decision diagrams* (BDDs), except that the leaves store numerical abstract constraints (e.g., C_0 and C_1 in the above example) instead of Boolean values **true** and **false**.

Example 5.2 (Relational Boolean abstraction) The abstract decision tree shown in figure 5.1(c) characterizes precisely the set of reachable states shown in figure 5.1(b) of the program of figure 5.1(a): when b_0 is equal to **true**, then x_0 is negative, and when b_0 is equal to **false**, then x_0 is positive or null. Nonetheless, the analysis algorithms required to achieve these results are more complex than those shown in chapter 3, as the domain is relational.

5.1.2 Describing Conjunctive Properties

Verifying a program often requires using *conjunctive* predicates, that is, logical predicates that correspond to the conjunction of several basic predicates. We

consider here the construction of abstract domains that can handle such cases.

Construction of a Domain for Conjunctions More precisely, we assume that two abstract domains A_0 and A_1 describing sets of concrete states are defined together with their concretization functions $\gamma_0 : A_0 \rightarrow \wp(M)$ and $\gamma_1 : A_1 \rightarrow \wp(M)$, and we define an abstract domain that describes logical predicates that correspond to the conjunction of two predicates, respectively, from A_0 and A_1 . Since the most intuitive way to define a logical predicate $P_0 \wedge P_1$ (where we denote logical conjunction by \wedge) is to build a pair of P_0 and P_1 , this domain of conjunctive properties boils down to a *product* abstract domain, as follows.

Definition 5.1 (Product domain) *The product abstraction is defined by*

- *the abstract domain $A_\times = A_0 \times A_1$ that collects the pairs made of an element of A_0 and of an element of A_1 ; and*
- *the concretization function γ_\times defined as*

$$\forall (a_0, a_1) \in A_\times, \gamma_\times(a_0, a_1) = \gamma_0(a_0) \cap \gamma_1(a_1)$$

This definition matches the intuition of the conjunctive abstractions since an abstract value (a_0, a_1) describes the set of concrete states that satisfy both the property denoted by a_0 and the property denoted by a_1 . Furthermore, we note that it generalizes seamlessly to the conjunction of more than two properties. The machine representation of product abstract domains usually relies on record types. In fact, we have already manipulated conjunctive properties, hence products, in previous chapters even though we did not stress this fact at that point.

Example 5.3 (Non-relational domain as a product domain) *Let us assume that concrete states are stores with two variables x, y (thus, a concrete state is essentially a point in the two-dimensional space, as in chapter 2). We consider a non-relational abstraction introduced in definition 3.7, that is, intervals. Then an abstract state defines an interval for x and an interval for y . This corresponds to an instance of a product abstraction, where A_0 characterizes the values that x may take, and A_1 does the same for y .*

Obviously, product abstractions are also very interesting when the two domains characterize the same variables with distinct kinds of predicates.

Example 5.4 (Product of two numerical abstract domains) *Let us assume that a concrete state is a store with a single variable i (i.e., a concrete state is characterized by a single value, and the non-relational abstraction coincides with the value abstraction). Moreover, we denote the abstract domain of intervals (example 3.7) by A_0 , and we write A_1 for the abstract domain of congruences (example 3.8). Then the product domain $A_0 \times A_1$ lets an abstract value pack both a range constraint and a congruence constraint. For instance, abstract state $([0,100],(0,2))$ describes sets of even values that are between 0 and 100. Low-level programs often manipulate pointers, with range and alignment constraints, and this product abstraction provides the right information to characterize such values.*

Product Domain, Precision, and Reduced Product The basic product abstraction of definition 5.1 is not sufficient to effectively let both sides of the conjunction cooperate, that is, to exchange information so as to derive stronger properties. To show this, we use the same setup as in example 5.4, and consider the combination of the interval constraint [1,3] and of the congruence constraint (0,2) that describes even numbers; then the real concretization is {2}, so that the most precise abstract information is defined by interval [2,2] and by congruence (0,2) (which describes constant 2). This information refinement operation is called a *reduction* [27]. It proceeds by merging equivalent abstract information into their best representation, as follows.

Definition 5.2 (Reduced product) *With the same notations as in definition 5.1, the reduced product [27] of abstract domains A_0 and A_1 is defined by the set A_{\bowtie} of equivalence classes of $A_0 \times A_1$ for the relation \equiv , defined by*

$$(a_0, a_1) \equiv (a'_0, a'_1) \iff \gamma_x(a_0, a_1) = \gamma_x(a'_0, a'_1)$$

The concretization function of the reduced product is defined by γ_x on the equivalence classes of \equiv .

Before we comment on the application of this definition in static analysis, we consider how it works out on our two examples of product domains.

Example 5.5 (Coalescent product and non-relational abstraction) *We consider the issue of reduction in the setup of example 5.3. For simplicity, we assume that $X = \{x, y\}$ and that the abstract domain of intervals is used (an abstract value maps each variable to an interval). We write M_{\perp}^{\sharp} for the abstract element defined by $M_{\perp}^{\sharp}(x) = M_{\perp}^{\sharp}(y) = \perp$. First, we search which abstract values need reduction.*

When an abstract state maps both x and y to non-empty intervals, it describes exactly the set of concrete states that satisfy these range constraints; thus, there exists no other abstract state in the same equivalence relation for \equiv , and it needs no reduction.

Let us now assume that M^{\sharp} is an element of the non-relational abstract domain that maps x into the empty interval \perp . Then we observe that $\gamma_{\mathcal{N}}(M^{\sharp}) = \emptyset = \gamma_{\mathcal{N}}(M_{\perp}^{\sharp})$. Therefore, M^{\sharp} and M_{\perp}^{\sharp} belong to the same equivalence class and can all be represented by a single abstract element. Obviously, the most simple choice is M_{\perp}^{\sharp} . More generally, when M^{\sharp} maps any variable into \perp , it should be reduced into M_{\perp}^{\sharp} . This principle generalizes to any number of variables (or numerical dimensions) and to any underlying value abstract domain. We call the corresponding abstraction the coalescent product. It is defined by the following set of elements:

- *The least element M_{\perp}^{\sharp} is the function that maps any dimension into the bottom value of the underlying value abstraction.*
- *The elements $M^{\sharp} : X \rightarrow A_V$ are such that, for all $x \in X$, $M^{\sharp}(x) \neq \perp$ (i.e., no dimension is mapped to the bottom value of the underlying domain).*

In fact, the analysis of section 3.3 uses such a coalescent product. Indeed, we observed that the best approximation of the empty set of stores in a non-

relational domain is the \perp element that maps each variable to the infimum element of the value domain, and we let $[\mathcal{C}]_{\wp}^{\sharp}(\perp) = \perp$ for any command \mathcal{C} .

Last, we sketch the reduction in the case of the product of intervals and congruences.

Example 5.6 (Reduced product of intervals and congruences) We use the same setup as in example 5.4. We consider an interval $[a,b]$ and a congruence (n, p) , and we discuss when this information can be improved. One obvious case is when either bound of $[a,b]$ does not satisfy the congruence constraint. For instance, if the interval is $[1,5]$ and if the congruence information describes even values (i.e., is the pair $(0,2)$), then we can refine the interval into $[2,4]$ without losing any concrete state since values 1 and 5 can be ruled out from the concretization. In particular, this reduction may turn an abstract state into the pair (\perp, \perp) when no value in the interval satisfies the congruence information. For instance, if the congruence information describes the set of values that are a multiple of 4 (i.e., is $(0,4)$), and if the interval is $[1,3]$, then the concretization is actually empty, and we can reduce $([1,3], (0,4))$ to (\perp, \perp) . Finally, when the interval boils down to a singleton, then the congruence information can be refined too.

Analysis with a Reduced Product Domain Reduced product (definition 5.2) is generally vastly superior to the regular product of abstract domains (definition 5.1) in terms of precision [27] since it lets constraints in either member of the product refine the information described by the other member. This follows from the precision gains noticed in examples 5.5 and 5.6. In particular, coalescent product often improves the precision of the analysis of condition tests, as shown in the following example.

Example 5.7 (Coalescent product-based analysis of a condition test) We study the program shown in figure 5.2. Although it is contrived, this program illustrates how coalescent product contributes to the analysis precision.

First, we consider an analysis that does not perform the reduction of the coalescent product. The abstract state computed at the entry of the true branch of the condition statement is $\{x \mapsto \perp, y \mapsto [1,1]\}$. At the end of the true branch, the analysis computes the abstract state $\{x \mapsto \perp, y \mapsto [0,0]\}$. Computing the point-wise abstract union of this abstract state with the abstract state produced on the false branch produces the abstract state $\{x \mapsto [8,8], y \mapsto [0,1]\}$, which is obviously imprecise. Indeed, the true branch is not reachable in the concrete semantics, and y is always equal to 1.

```

int x,y;
x = 8;
y = 1;
if(x < 0){
    y = 0;
}else{}

```

Figure 5.2

Analysis with a coalescent product domain

We now consider an analysis that carries out the reduction. Then, the abstract state that is obtained at the entry of the true branch is M_{\perp}^{\sharp} and so is the abstract state at the exit of that branch. As a consequence, the analysis produces $\{x \mapsto [8,8], y \mapsto [1,1]\}$ as a post-condition, which is the expected result.

Other forms of reduced product (e.g., the one presented in example 5.6) achieve similar precision gains.

For this reason, a static analysis based on a *reduced product* is in general more precise than a *product of static analyses*. To better understand the implication of this remark, let us assume that we need to infer two distinct (seemingly unrelated) semantic properties by static analysis of a same program. Then performing two separate analyses generally yields less precise results than doing a single analysis based on a reduced product, as the reduced product-based analysis can refine abstract information at any time and not only at the very end.

On the other hand, the definition of the optimal reduced product is often hard and may be costly to implement. Therefore, a common techniques relies on a product representation (an abstract value of A_{\bowtie} is a pair made of an element of A_0 and of an element of A_1) and on a *reduction operation* that refines abstract states. Examples 5.5 and 5.6 provide instances of reduction operations. Such an operation should be sound (it should not lose any concrete state) and should ideally turn any abstract element into its optimal representation. However, in practice, an approximate reduction operation is often preferable to an optimal one when it allows achieving a sufficient level of precision at a lower computational cost. Moreover, reduction is typically performed only at specific points where it is known that it can provide a useful precision gain. As an example, we consider the reduction presented in example 5.5. This reduction propagates \perp information to all dimensions when the possible values of any

variable can be abstracted by \perp . Since condition tests are the only kind of statement where the analysis may derive such a fact from a non- \perp abstract precondition, the coalescent product reduction needs to be computed only at these locations.

```
int x0, x1, x2;
if(x0 > 0){
    x0 = x0 + 1;
} else{
    (do nothing)
}
x2 = x1 / (x0 - 1);
```

Figure 5.3

A challenging program with respect to abstract join

5.1.3 Describing Properties Involving Case Splits

We now study abstractions to reason over properties that involve logical case splits. In particular, this includes *disjunctive properties*.

First, we remark that we have already studied in section 5.1.1 an abstract domain construction that allows expressing some disjunctive properties. Indeed, the Boolean diagram-based abstract domain depicted in figure 5.1(c) can represent exactly disjunctive properties of the form $(b = \text{true} \wedge \dots) \vee (b = \text{false} \wedge \dots)$. Nevertheless, not all relevant disjunctive properties are of that form.

As an example, we consider the program in figure 5.3. We make the assumption that the execution may start from any state. Then after the conditional statement, x_0 is not equal to 1, so the division at the next line is safe. We now discuss the verification of this property.

However, if we use a static analysis based on intervals similar to that defined in chapter 3, we obtain $(-\infty, +\infty)$ as a range for x_0 at that point. All values in this range can be observed in concrete executions, except value 1. Therefore, this analysis is imprecise and fails to prove that the division is safe due to this. We may seek for another numerical abstract domain that does not suffer from this imprecision.

Unfortunately, none of the domains shown in chapter 3 allow computing more precise information on x_0 , as we can easily show. First, we can observe that any abstract domain such that all abstract elements describe *convex* sets of values (in the geometrical sense) cannot avoid this loss of precision. Indeed, there exist executions where x_0 is smaller than 0 at the division location and executions where it is greater than 2 at that same point. This remark rules out the existence of a successful analysis using not only intervals but also constants, octagons, polyhedra, and so forth. Additionally, we remark that the abstract domains of signs and congruences cannot capture exactly the set $(-\infty, 0] \cup [2, +\infty)$.

In the logical point of view, there are two ways to address this issue:

- We can build a domain that supports disjunctive reasoning and is able to express properties, such as

$$A_0 \vee A_1 \vee \cdots \vee A_n$$

- We can construct an abstract domain that supports reasoning by case analysis, which boils down to conjunctions of implications:

$$(A_0 \Rightarrow B_0) \wedge (A_1 \Rightarrow B_1) \wedge \cdots \wedge (A_n \Rightarrow B_n)$$

The following paragraphs study these two approaches.

Disjunctive Completion In the analysis of the program shown above using intervals, the imprecision occurs at the control flow join point. This imprecision is because the abstract join adds the value 1 to the concretization of the result. Therefore, a straightforward way to avoid this issue is to change the abstract domain so that we can compute a very precise abstract join.

We call *exact abstract join* an over-approximation of unions of concrete sets that does not lose precision. In chapter 3, we required abstract join to ensure that $\gamma(M_0^\sharp) \cup \gamma(M_1^\sharp) \subseteq \gamma(M_0^\sharp \sqcup^\sharp M_1^\sharp)$ for all abstract states M_0^\sharp and M_1^\sharp . By contrast, an exact abstract join is an operator that satisfies the following formula:

$$\gamma(M_0^\sharp) \cup \gamma(M_1^\sharp) = \gamma(M_0^\sharp \sqcup^\sharp M_1^\sharp)$$

Some abstract domains allow defining such an operator. For instance, the two variants of the sign abstract domains shown in figure 3.5 satisfy this property.

However, many abstract domains do not allow defining such an operator. This includes the abstract domains of intervals (figure 3.5(c)), of linear equalities (figure 3.6(a)), and of convex polyhedra (figure 3.6(b)). The

disjunctive completion [27] of an abstract domain (A, \sqsubseteq) is obtained by adding elements to A so that it becomes possible to define an exact abstract join. In the case of the abstract domain of intervals, we need to add all finite disjunctions of intervals. This allows expressing exactly $(-\infty, 0] \cup [2, +\infty)$ and thus addresses the imprecision remarked above.

However, disjunctive completion is generally a very costly and rather impractical approach. The issue is that the representation of abstract states tends to become prohibitively large. This also brings a new issue regarding the analysis of loops; indeed, a widening should be defined, either by limiting the size of symbolic disjunctions or by computing approximate union of well-chosen basic disjuncts.

Reduced Cardinal Power We now study the construction of an abstract domain to represent properties defined by case analysis, as a conjunction of implications of the form $A_i \Rightarrow B_i$. The idea is to let two abstract domains respectively represent the left-hand side and the right-hand side of each implication. The general notion of *cardinal power abstraction* achieves this and may be used for many kinds of properties. In particular, it allows reasoning by case analysis over various notions of program behaviors, such as reachable states or feasible executions. Thus, we denote the set of all possible such behaviors by \mathcal{E} .

Definition 5.3 (Cardinal power abstract domain) We assume that two abstract domains A_0, A_1 are defined together with their concretization functions $\gamma_0 : A_0 \rightarrow \wp(\mathcal{E})$ and $\gamma_1 : A_1 \rightarrow \wp(\mathcal{E})$ (where we denote the set of program behaviors by \mathcal{E}). We write \sqsubseteq_0 and \sqsubseteq_1 for the order relations induced by the concretizations. The cardinal power abstraction [27] is defined by

- the set of abstract elements A_{\rightarrow} made of the monotone functions from A_0 to A_1 (the origin of the name cardinal power is that we consider elements of $A_1^{A_0}$); and
- the concretization $\gamma_{\rightarrow} :$

$$\gamma_{\rightarrow}(a_{\rightarrow}) = \{m \in \mathcal{E} \mid \forall a_0 \in A_0, m \in \gamma_0(a_0) \Rightarrow m \in \gamma_1(a_{\rightarrow}(a_0))\}$$

The abstract ordering induced by a_{\rightarrow} is the point-wise extension of \sqsubseteq_1 .

Intuitively, such an abstract element maps a property a_0 described by an element of A_0 to a property $a_{\rightarrow}(a_0)$ described by an element of A_1 such that, if a concrete memory satisfies a_0 , then it also satisfies $a_{\rightarrow}(a_0)$. Therefore, definition 5.3 indeed captures the notion of conjunction of implications that was introduced earlier. This notion offers a great level of flexibility since it is parameterized by two abstractions (one for each side of the implication). The following paragraphs discuss several common choices for A_0 .

We can observe that the Boolean diagram-based abstract domain depicted in figure 5.1(c) can be viewed as an instance of cardinal power. Indeed, the elements of this abstract domain naturally describe conjunctions of implications, for instance,

if b contains **true** then numerical condition $x > 10$ holds,
 and if b contains **false** then numerical condition $x < 3$ holds.

In the same way as for the product abstract domains, cardinal power abstract domains can be made more precise by *reduction*. For instance, the information in a_0 may refine that in $a \rightarrow (a_0)$. In practice, abstract elements of cardinal power abstract domains can be represented using arrays or functional arrays. Moreover, we may use a more compact representation that does store the full definition of all the entries $(a_0, a \rightarrow (a_0))$; indeed, when one such entry can be recomputed from others, it can be omitted from the abstract state without changing the overall meaning.

State Partitioning We call *state partitioning* the application of definition 5.3 under the assumption that \mathcal{E} is the set of memory states M . We consider a few interesting choices for A_0 in this setup (under the assumption that A_1 is the abstract domain of intervals).

First, we let A_0 describe the abstract domain of signs with elements $\{\perp, [< 0], [= 0], [> 0], \top\}$ (where we denote the set of negative numbers by $[< 0]$ and we denote the set of positive numbers by $[> 0]$). The following abstract state describes exactly the set of states that we need to capture to analyze precisely the program of figure 5.3:

$$\begin{aligned} [< 0] &\mapsto (-\infty, -1] \\ [= 0] &\mapsto [0, 0] \\ [> 0] &\mapsto [2, +\infty) \end{aligned}$$

As another example, if some variables have Boolean type, and A_0 describes their values (and abstracts away all numerical variables), we obtain exactly the relational Boolean abstraction described in example 5.2.

Other interesting choices of A_0 include information on the context:

- We say that a static analysis is *flow-sensitive* if it computes one abstract state per program point, which accounts for all the states observed at that point (all analyses presented in chapters 2, 3, and 4 are of that kind). A flow-sensitive analysis can be viewed as an

instance of state partitioning, where the elements of A_0 correspond exactly to the program labels (or control states), as shown in section 4.1.1, so that the abstract elements of A_{\rightarrow} are functions from labels to local invariants.

- Although we have not considered languages with procedures yet (we consider them in sections 8.2 and 8.4.1), we say that a static analysis is *context-sensitive* if the analysis discriminates, to some extent, the contexts of multiple procedure calls [100]. A context-sensitive analysis can also be viewed as an instance of state partitioning, where A_0 describes context information, so that an element of A_{\rightarrow} maps a calling context to abstract information relative to that context.

Elements of A_0 fix the structure of partitions used during the analysis. In some cases, these partitions need to be determined by the analysis itself, based on the target program or on the target properties. This is called *dynamic partitioning* [13, 61].

Trace Partitioning We call *trace partitioning* [56, 97] the application of definition 5.3 under the assumption that \mathcal{E} is the set of all program executions, defined as sequences of states produced during a run of a program. In this view, A_0 is an abstraction that describes sets of program executions.

In general, all forms of state partitioning discussed above are also instances of trace partitioning, where A_0 describes only the last state of execution traces.

More interesting examples resort to abstractions that describe global execution properties. An example is the case where A_0 discriminates executions depending on sequences of program points visited in the past. For instance, we can use an instance of trace partitioning that distinguishes program executions that went through either branch of a condition test instruction. This also solves the imprecision observed in the analysis of the program in figure 5.3:

$$\begin{aligned} \text{the "true" branch was executed} &\implies x_0 \in [1, +\infty) \\ \wedge \text{ the "false" branch was executed} &\implies x_0 \in (-\infty, 0] \end{aligned}$$

5.1.4 Construction of an Abstract Domain

In general, an abstract domain is characterized by the set of logical properties that it can express and their machine representation. Moreover, it should also come with basic abstract operations to build static analyzers.

More precisely, each abstract domain that we have defined so far is defined by the set of logical predicates that its elements describe. The implementation of an abstract domain requires a representation for abstract elements. In some cases, this representation is obvious. For instance, an element of the lattice of interval is either bottom or an interval made of two bounds, so it boils down to a sum type with one case defining pairs. On the other hand, elements of the abstract domain of convex polyhedra may be viewed either as conjunctions of linear inequalities or as geometric objects, with edges and vertices.

The abstraction relation defines the meaning of each abstract element; thus, it bridges the gap between the representation of abstract elements and, for instance, the concrete states that they describe. It plays a very important role since it underlies the statement of soundness of the analysis (e.g., theorems 3.6 and 4.4) and its proof. In practice, one should identify the nature of the relation between concrete and abstract elements, due to the implications on the resulting static analysis. For instance, when the abstraction relation has a best abstraction function α (definition 3.4), one may identify a most precise abstraction for any concrete property. On the other hand, when no best abstraction relation can be defined, some concrete properties have no best approximation in the abstract domain; thus, some analyses may necessarily return imprecise results. Depending on the abstraction relation, one may also need to implement a reduction operator over the abstract domain (as the reduced product construction does, as shown in definition 5.2).

Typical abstract operations include abstract lattice operations (inclusion test, abstract union, widening) and transfer functions (to over-approximate post-conditions for assignments, condition tests, and other basic program commands). Their design is guided by the abstraction relation and the goal to ensure soundness. In fact, it is often a good practice to derive the definition of these operations and to construct the proof of soundness in the same time.

When constructing the abstract domain required for a static analysis, one typically first considers the set of logical predicates that need to be computed and then chooses the representation and formalize the abstraction relation. In this process, it is often most helpful to identify when standard constructions such as reduced product or cardinal power apply, and what basic abstract domains can be used:

- When constraints that bind the values of distinct variables are required, *relational* abstract domains are required.
- When abstract elements should describe conjunctions of predicates of different forms, one needs to consider *product* or *reduced product*

abstract domain compositions (section 5.1.2).

- Finally, disjunctive properties generally call for (partial) *disjunctive completion* or for partitioning abstract domains based on *cardinal power* (section 5.1.3).

5.2 Advanced Iteration Techniques

We have shown in chapters 2, 3, and 4 that the inference of loop invariants involves the computation of sequences of abstract iterates. This process computes weaker and weaker abstract states until stabilization; hence, the computation of abstract iterations is a common source of imprecision. There exist many techniques to improve the precision of the static analysis of loops. We study some of them in the following paragraphs, after we recall a basic setup for the analysis of a loop **while**(\mathcal{B}) $\{\mathcal{C}\}$:

- We assume that a concrete function F from sets of states to sets of states accounts for the effect of the condition on loop entry followed by the body of the loop (as shown in chapter 3, $F = \llbracket \mathcal{C} \rrbracket_{\wp} \circ \mathcal{F}_{\mathcal{B}}$).
- We assume that an abstract function $F^{\#}$ over-approximates F in the sense that $F \circ \gamma \subseteq \gamma \circ F^{\#}$, and that $M^{\#}$ is an abstract pre-condition for the loop.
- We call G the function defined by $G(X) = \gamma(M^{\#}) \cup F(X)$.
- Then the analysis needs to compute an over-approximation for the join of the iterates of G , which is also its least fixpoint (the notion of least fixpoint is used in section 5.2.3, but not in sections 5.2.1 and 5.2.2):

$$M_{\text{loop}} = \bigcup_{i \geq 0} F^i(\gamma(M^{\#})) = \text{lfp } G$$

- To achieve this, we propose to use a widening operator ∇ and to compute the converging sequence of abstract iterates defined by

$$\begin{aligned} M_0^{\#} &= M^{\#} \\ M_{k+1}^{\#} &= M_k^{\#} \nabla F^{\#}(M_k^{\#}) \end{aligned}$$

- We let $G^{\#}$ denote the function defined by $G^{\#}(X) = M^{\#} \sqcup^{\#} F^{\#}(X)$ (which is thus also such that $G \circ \gamma \subseteq \gamma \circ G^{\#}$).
- We denote the limit of this sequence by $M_{\lim}^{\#}$, which is reached after finitely many iterates, and the analysis returns the sound loop post-

condition $\mathcal{F}_{\neg B}^\sharp(M_{\text{lim}}^\sharp)$.

We use the language of chapter 3 and the interval abstract domains, but all techniques discussed in the following would apply to other setups as well.

5.2.1 Loop Unrolling

In practice, the first iteration(s) of a loop often has a special effect. For instance, a common practice is to let the first iteration perform some initialization tasks. Thus, letting the analysis handle the first iterations separately from the rest is often beneficial to precision.

```

if(x < 0 || x > 1000){
    x = 0;
}
else{
    x = 1 + x;
}
while(i > 0){
    if(x < 0 || x > 1000){
        x = 0;
    }
    else{
        x = 1 + x;
    }
    input(i);
}
(b) Unrolling initialization iteration

if(x < 0 || x > 1000){
    x = 0;
}
else{
    x = 1 + x;
}
input(i);
}
(a) A loop with initialization cycle

```

Figure 5.4

Loop unrolling

Let us consider the program in figure 5.4(a). The loop is executed a nondeterministically chosen, nonnull number of times. When the execution enters the loop for the first time, the variable x is uninitialized; thus, it may store any integer value. However, after the first execution of the loop body, it is in the range $[0,1001]$. Subsequent iterations of the loop preserve this invariant; indeed, when x reaches 1001, it is reset to 0. This range constraint also holds when the execution eventually exits from the loop.

However, the analyses relying on the iteration strategies described in chapters 3 and 4 cannot infer this range, whatever the abstraction. The reason for this limitation is that the analysis of loops essentially computes a local invariant at the loop entry point, and this invariant needs to cover all states at loop entry, including the initial states; therefore, the most precise invariant they can compute at loop head accounts for states where x may take any possible value.

To compute a better result, we need to single out the first iteration in the loop from the subsequent iterations, as we observed that the value of x is unconstrained only at the beginning of the first iteration. Intuitively, we would like to unroll the first iteration, and analyze it separately from the subsequent iterations. The program of figure 5.4(b) captures this intuition. While x is still assumed to be uninitialized at the beginning, the most precise range constraint over x at the loop head is indeed $[0,100]$, and it allows deriving the precise post-condition mentioned above.

```

x := 0;                                x := 0;
while(rand()){                         while(x ≤ 100){
    if(rand()){                      if(x ≥ 50){
        x := -1                     x := 10
    }else{                           }else{
        x := x + 2                  x := x + 1
    }
}
}
(a)                                     (b)

```

Figure 5.5

Basic loop examples where analysis can be improved using basic widening techniques

The principle of *loop unrolling* [12] is to delay the application of abstract unions during the analysis of programs such as the one in figure 5.4(a). The delay may affect one iteration or more. We now formalize this intuition with the notations recalled at the beginning of section 5.2. The analysis unrolling the N first iterations of the loop computes the following sequence:

$$\begin{aligned} M_0^\sharp &= M^\sharp \\ M_{k+1}^\sharp &= \begin{cases} F^\sharp(M_k^\sharp) & \text{if } k < N \\ M_k^\sharp \sqcap F^\sharp(M_k^\sharp) & \text{otherwise} \end{cases} \end{aligned}$$

Once this sequence reaches a limit M_{\lim}^\sharp , it returns

$$\mathcal{F}_{\neg B}^\sharp(M_0^\sharp) \sqcup^\sharp \dots \sqcup^\sharp \mathcal{F}_{\neg B}^\sharp(M_{N-1}^\sharp) \sqcup^\sharp \mathcal{F}_{\neg B}^\sharp(M_{\lim}^\sharp)$$

This approach delays join, hence the loss of precision that it may induce. In particular, if we consider figure 5.4(a) and let $N = 1$, it effectively avoids to merge the first iteration states with the states observed later; hence, it allows computing the range $[0,1001]$ for x .

5.2.2 Fixpoint Approximation with More Precise Widening Iteration

We now discuss a few techniques related to the computation of widening sequences. Indeed, we showed in chapter 2 that widening enforces convergence at the cost of a significant loss of precision. However, in many cases, this loss of precision can be significantly reduced, as the next paragraphs show, based on a couple of contrived but representative examples.

Delaying Widening with Union Let us first consider the program shown in figure 5.5(a). We write **rand()** for an expression that returns a random value (it would be trivial to simulate it in the language of section 3.1 using an **input** command and an additional variable). In this example, the variable x ranges in $[-1, +\infty)$ since it starts at 0, may be set to -1 at any time, and may take arbitrarily high values. A widening sequence following the definitions of chapter 3 would produce the following result:

- Before entering the loop, the range for x is $[0,0]$.
- At the exit of the first iteration, x is in $[-1,2]$; thus, the first iterate is $[0,0] \nabla [-1,2] = [-\infty, +\infty]$. At this stage, the abstract iteration obviously terminates since it reached the most imprecise possible result.

The issue is that the first iteration causes x to take various new values, thus weakening the previous $[0,0]$ in both directions. The standard widening presented in chapter 3 produces rather coarse results in such a case. While widening the right bound to $+\infty$ provides the desired result, it is not acceptable to widen the left bound to $-\infty$.

An alternative approach is to *delay* the application of widening [12]. This could be done by loop unrolling as in section 5.2.1, but loop unrolling is not

really necessary here as the first iteration is not any different from the subsequent ones. A middle ground solution consists in using regular abstract union $\sqcup^\#$ for the first N iterations in the loop. This way, the results of the N first iterations are likely more precise than if widening were used. This amounts to computing the following iteration sequence:

$$\begin{aligned} M_0^\# &= M^\# \\ M_{k+1}^\# &= M_k^\# \sqcup^\# F^\#(M_k^\#) && \text{if } k \leq N \\ M_{k+1}^\# &= M_k^\# \nabla F^\#(M_k^\#) && \text{if } k > N \end{aligned}$$

The soundness of the result of this iteration sequence can be justified in the same way as in section 3.3.3.

It is also possible to unroll the N first iterations (as done in section 5.2.1) and then to apply $\sqcup^\#$ instead of ∇ for the next N' iterations. Note that both techniques are different: unrolling completely postpones join until after the loop invariant computation, whereas using abstract join for the first abstract iterations allows initiating the abstract convergence (though using a slower and more precise operation than ∇).

Assuming $N = 2$, the analysis of the program of figure 5.5(a) now produces the following sequence of ranges for x :

| | | |
|-----------------|--|--------------------|
| at rank $k = 0$ | $[0, 0]$ | |
| at rank $k = 1$ | $[0, 0] \sqcup^\# [-1, 2] = [-1, 2]$ | |
| at rank $k = 2$ | $[-1, 2] \sqcup^\# [-1, 4] = [-1, 4]$ | left bound stable |
| at rank $k = 3$ | $[-1, 4] \nabla [-1, 6] = [-1, +\infty)$ | |
| at rank $k = 2$ | $[-1, +\infty) \nabla [-1, +\infty) = [-1, +\infty)$ | both bounds stable |

Widening with Threshold We now consider the program of figure 5.5(b). This example was studied in figure 3.9(b), and we noted there that a basic iteration with widening computes the range $[0, +\infty)$ for x instead of the best possible invariant $[0, 50]$. The issue here is that the iteration starts with the upper bound 0 on the value of x , which is not stable; thus, it gets widened all the way to $+\infty$. However, we remark that 50 would be stable. In fact, other values would be stable, such as 70 or even 120; if x is greater than 100, the loop exits, so the value of x does not increase anymore, and if we enter the loop with any value greater than 50, x gets reset to 10.

It is possible to solve this issue with a slower and more precise widening. Instead of widening all unstable bound constraints to $+\infty$ in a single step, this new widening does so in several steps and stops at pre-defined *threshold values*

[12]. Let us assume that we consider only one threshold B . Then the widening with threshold can be defined as follows (we show only the right bounds, since the case of the left bounds is symmetric):

$$[n, p] \nabla_{\gamma} [n, q] = \begin{cases} [n, p] & \text{if } p \geq q \\ [n, B] & \text{if } p < q \leq B \\ [n, +\infty) & \text{if } B < q \end{cases}$$

Assuming $B = 50$, we get the following sequence:

$$\begin{aligned} \text{at rank } k = 0 & \quad [0, 0] \\ \text{at rank } k = 1 & \quad [0, 0] \nabla_{\gamma} [0, 1] = [0, 50] \quad \text{since } 0 < 1 \leq 50 \\ \text{at rank } k = 2 & \quad [0, 50] \nabla_{\gamma} [0, 50] = [0, 50] \quad \text{both bounds are stable} \end{aligned}$$

5.2.3 Refinement of an Abstract Approximation of a Least Fixpoint

The previous subsections discussed how the analysis of loops can be made more precise before and during the computation of a sequence of abstract iterates.

In certain cases, it is also possible to improve the precision of the result of an abstract iteration sequence after it converges. To show this, we use the fixpoint characterization of the loop semantics made in section 3.3.3. Recall the basic setup of loop analysis there.

First, we observe that once M_{\lim}^{\sharp} has been computed and provides an over-approximation of M_{loop} , then so does $G^{\sharp}(M_{\lim}^{\sharp})$. Indeed, let us assume that M_{\lim}^{\sharp} over-approximates M_{loop} ; then,

- since the concrete semantic function G is monotone and $M_{\text{loop}} \subseteq \gamma(M_{\lim}^{\sharp})$ (since M_{\lim}^{\sharp} is a sound over-approximation of M_{loop}), we have $G(M_{\text{loop}}) \subseteq G(\gamma(M_{\lim}^{\sharp}))$;
- since M_{loop} is a fixpoint of G , $G(M_{\text{loop}}) = M_{\text{loop}}$, we have $M_{\text{loop}} \subseteq G(\gamma(M_{\lim}^{\sharp}))$;
- since $G \circ \gamma \subseteq \gamma \circ G^{\sharp}$, we have $G(\gamma(M_{\lim}^{\sharp})) \subseteq \gamma(G^{\sharp}(M_{\lim}^{\sharp}))$;
- thus, by composing implications, we derive $M_{\text{loop}} \subseteq \gamma(G^{\sharp}(M_{\lim}^{\sharp}))$.

In other words, given a concrete fixpoint of the concrete function G , applying G^{\sharp} again to any sound approximation of this concrete fixpoint produces another sound approximation. This means that, once the algorithm of theorem 3.5 stabilizes, we can simply compute one more iteration in the abstract level and still obtain a sound approximation of the states at loop head. Very often, this additional iteration actually provides an increase in precision. In particular,

condition tests may help refine the information in the loop invariant. Obviously, this process may be repeated several times. In fact, it can even be generalized with the use of a *narrowing* operator [26].

```

x = 0;
while(rand() && x < 50){
    x = 1 + x;
}
(a)

x = 0;
while(rand() && x < 50){
    if(rand()){
        y = x;
    }
    x = 1 + x;
}
(b)

```

Figure 5.6

Refinement of loop invariants

To illustrate this, we first consider the program of figure 5.6(a) and do not make use of any of the unrolling or threshold techniques presented in the previous paragraphs. This means that the algorithm of theorem 3.5 or 4.3 will compute the loop invariant M_{lim}^\sharp with the imprecise range $[0,+\infty)$ for x . The semantic function F associated to the loop body is composed of a test that the value of x is strictly smaller than 50, and of an incrementation of x ; thus, applying the corresponding analysis function to M_{lim}^\sharp yields $\{x \mapsto [1,50]\}$. After an abstract join with the entry abstract state $\{x \mapsto [0,0]\}$, we thus get $\{x \mapsto [0,50]\}$. The above remark proves $[0,50]$ is a sound over-approximation of the values that x may take at loop head. We also remark that this is the most precise range that we could hope for since it is exactly the set of all the values that x may take at that point.

However, this technique has some important limitations that can be observed in the case of the program of figure 5.6(b). In this case, the reachable states are all such that $0 \leq y \leq x \leq 50$. Using interval analysis, we thus expect the abstract loop invariant $\{x \mapsto [0,50], y \mapsto [0,50]\}$. As in the case of the code of figure 5.6(a), the abstract iteration sequence computed by the algorithm of theorem 3.5 or 4.3 produces $\{x \mapsto [0,+\infty), y \mapsto [0,+\infty)\}$. After applying the same technique as above, the test allows refining the information on x into the interval $[0,50]$.

However, the assignment to y may be performed or not. Thus, the resulting abstract state is $\{x \mapsto [0,50], y \mapsto [0,+\infty)\}$. We may apply again the same technique as above, but still the range on y will not improve, for the same reason. The issue here is twofold. First, $\{x \mapsto [0,50], y \mapsto [0,+\infty)\}$ over-approximates several fixpoints of the concrete function G ; in fact, the concretization of $\{x \mapsto [0,50], y \mapsto [0,+\infty)\}$ is a fixpoint for G but is strictly less precise than the range of the least fixpoint that we would like the analysis to over-approximate. Second, the justification that we gave above to prove that applying G to the limit of the sequence of abstract iterates works not only for the least fixpoint of G but also for any fixpoint. In other words, applying G again will not allow refining analysis results below any fixpoint that the analysis already overshot. The consequence is that analyzing precisely the program of figure 5.6(b) requires the use of threshold widening since a loss of precision on y cannot be compensated for *a posteriori*.

5.3 Sparse Analysis

In this section we discuss a technique for reducing the analysis cost (in terms of memory and time consumption) *without* sacrificing the analysis precision.

Since all operations in an abstract interpreter definition are computable, the gap is fairly narrow between the definition and its implementation. The computable abstract semantics of section 3.3 is directly implementable, and the global fixpoint iteration algorithms in section 4.3 are straightforward enough to see that they faithfully implement the analysis specification.

However, blindly implementing the analysis as specified in the worklist-based fixpoint algorithm is not sufficient to ensure that the analysis scales up in practice. Such implementations may still be too costly to globally analyze, say, million-line programs. As an example of the complexity that the analysis must handle, see figure 5.7. The example C program (`less-382`) is over 20,000 lines of code, and the figure shows the hairy complexity of its call flows between procedures.

There exists lots of room for efficient implementation. In particular, the fixpoint iteration algorithm in section 4.3.2 leaves plenty of room for cost reduction. The time and space footprint of an analysis can be reduced by exploiting both the spatial and temporal sparsities of the program semantics, as follows.

- Spatial sparsity: Usually, each program portion (an expression, a statement, a sequence of statements, a procedure, a loop body, etc.) accesses only a small part of the whole memory.
- Temporal sparsity: After the definition (write) of a memory location, its use (read) is not immediate but a while later.

By exploiting these two sparsities in the semantics, hence reducing the time and memory footprint, we can substantially improve the scalability of the analysis. We call this cost-reduction technique *sparse analysis* [85].

The sparse analysis technique is largely independent of the underlying analysis definitions. Analysis designers first design a global and correct abstract interpretation whose scalability is unattended. Upon this underlying sound static analysis specification, analysis designers add the sparse analysis techniques to improve its scalability.

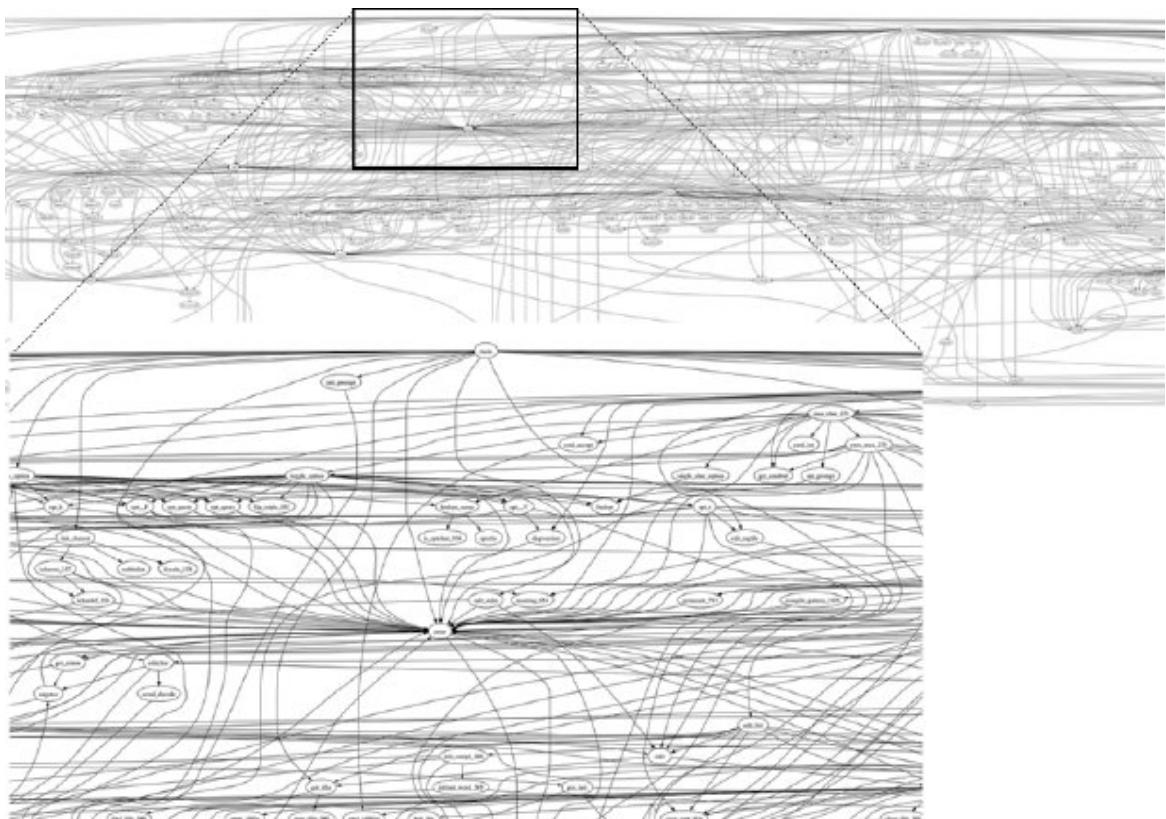


Figure 5.7

Illustration of the complexity of the semantics of real programs that static analysis must handle: call graph of `less-382` (23,822 lines of code)

The sparse analysis technique preserves the precision of the underlying analysis. Implementing the technique on top of a given global analysis only improves the analysis cost; the resulting sparse version computes the same analysis result as the underlying analysis.

The cost-reduction benefit of the sparse analysis is noticeable when the analysis is to estimate the abstract memory state at every program label. That is, the analysis result is a table from program labels to their abstract memory states. If the abstract memory state is again a table from abstract locations to abstract values, then the analysis result is a table in

$$L \rightarrow (A^\# \rightarrow V^\#).$$

Thus, since the number of labels and the number of abstract locations are proportional to the size of the input program to analyze, naive implementation of the above analysis will entail a huge memory footprint. For example, for a one-million-line input program, a naive implementation is to bookkeep a table of some million abstract locations (proportional to the number of lines) at each of its million labels (proportional to the number of lines).

The sparse analysis technique exploits the typical reality in program semantics: both the necessary states that flow and the necessary edges along which states flow are just fractions of the whole. First, we need to exploit a prevalent spatial sparsity. We do not have to bookkeep the whole memory state at each label. Rather, it is enough to keep only a fraction of the memory that will be used for the program part at the label. If the program part contains a statement that accesses, for example, only two variables, only those two entries are necessary. We can exploit this spatial sparsity to reduce the memory footprint.

Second, a temporal sparsity exists too. Memory entries do not have to follow the order in the program text. We do not have to deliver post-memory of each program label blindly following the syntactic control flow of the program. Recall that the semantic function is to transform the pre-memory at each label by the semantics of the program portion at the label into its post-memory. The post-memory becomes the pre-memory of the next label. Although for most cases this next label is syntactically determined, for example, at the next line of the program text, we can deliver post-memory directly to where it is used. When a memory entry is updated at the current label, we directly deliver the new entry to the label where the updated entry needs to be read. When a memory location is updated we can move the result directly to the place where the location is read.

We can exploit this temporal sparsity to reduce the time footprint, bypassing the labels whose code does not need those entries.

5.3.1 Exploiting Spatial Sparsity

This technique is sometimes called *abstract garbage collection*, or *frame rule* in separation logic [95]. When analyzing a program portion, we need only the part of the memory that will be accessed in that program portion. We do not need the whole memory.

```

x = x + 1;
y = y - 1;
z = x;
v = y;
ret *a + *b

```

Figure 5.8

Code fragment: we assume that `a` points to `v` and `b` to `z`

In our analysis context, for each program label we need only a part of the memory that will be accessed by the program portion associated with the label. Suppose that for each program label we know which memory locations are used. The abstract semantic function

$$F^\# : (L \rightarrow M^\#) \rightarrow (L \rightarrow M^\#)$$

becomes

$$F_{\text{sparse}}^\# : (\mathbb{L} \rightarrow M_{\text{sparse}}^\#) \rightarrow (\mathbb{L} \rightarrow M_{\text{sparse}}^\#),$$

where the $M_{\text{sparse}}^\#$ denotes the set of memories ($M^\#$) whose entries ($\text{dom}(M^\#)$) are restricted to the abstract locations $\text{Access}^\#(l)$ that may be accessed by the program portion of each label l :

$$M_{\text{sparse}}^\# = \{M^\# \in M^\# \mid \text{dom}(M^\#) = \text{Access}^\#(l), l \in \mathbb{L}\} \cup \{\perp\}$$

The set $\text{Access}^\#(l)$ of abstract locations to be used for each label l of the input program can be computed by a sound pre-analysis, which is typically coarser, hence quicker yet still sound, than the main analysis.

Example 5.8 (Spatial sparsity) Consider the C-like imperative program fragment in figure 5.8. Suppose that the analysis is to compute the memory state at each program label. The whole memory state is a table from all variables (**x**, **y**, **z**, **v**, **a**, and **b**) to their values. Figure 5.9(a) shows the case where the analysis bookkeeps the whole memory state at every label and where the flow edges (black arrows) are directly extracted from the program text. Note that each statement needs a fraction of the memory as its pre-state. For example, the first statement accesses only **x**, the second statement only **y**, and the last statement only **a**, **b**, **v**, and **z**. Bookkeeping the whole memory state at every label is wasteful. Figure 5.9(b) shows the case where the analysis bookkeeps only the necessary fractions of the whole memory.

5.3.2 Exploiting Temporal Sparsity

We let the analysis follow the semantic dependence to directly deliver the memory effect to its use point, not blindly following the syntactic control flow. For each program label, its defined locations are directly passed to its use labels, the program fragment of which will use the defined locations.

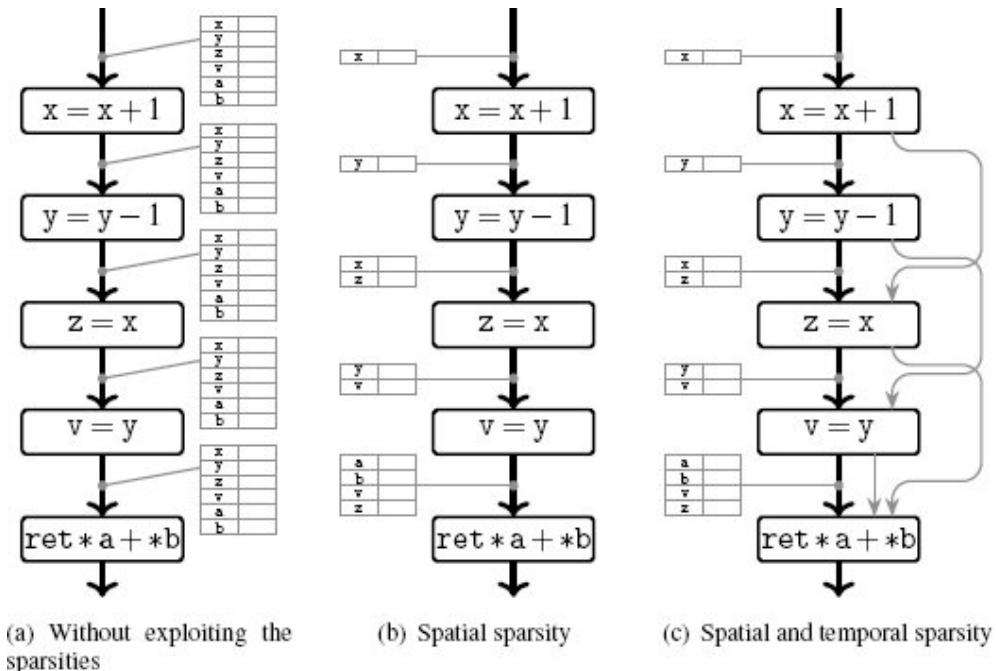


Figure 5.9

Spatial and temporal sparsity of program semantics

This *def-use* chain information that should be available before the analysis can be over-approximated by yet another analysis, which is coarser, hence quicker yet still sound, than the main analysis.

Assuming the availability of the def-use chain information, the temporally sparse analysis is defined by streamlining the abstract one-step relation

$$(l, M^\#) \hookrightarrow^\# (l', M'^\#) \text{ for } l' \in \text{next}^\#(l, M^\#)$$

so that the link $\hookrightarrow^\#$ should become sparse, from a label where a location is defined to a label where the defined location is read.

At label l , the input memory state $M^\#$ consists only of the locations that are to be accessed at label l . For each location defined (written) at l , the label l' is for a program portion that reads the defined location.

The $\text{next}^\#(l, M^\#)$ is to determine, in the abstract semantics, the def-use relation from definition point l to its use point l' for the locations $\text{dom}(M'^\#)$. In case that program portion at l does not define a location, l' would be the same as in the non-sparse case, and $M'^\#$ would be equal to $M^\#$.

Example 5.9 (Temporal sparsity) Consider again the program fragment in figure 5.9. Figure 5.9(c) shows sparse flow edges (the rounded, skipping arrows). For example, the defined variable x in the first statement “ $x = x + 1$ ” can directly flow to the third statement “ $z = x$ ”, which uses that new x , skipping the second statement. As opposed to the black edges, the rounded gray arrows are sparse, skipping the statements between the definitions and their uses.

5.3.3 Precision-Preserving Def-Use Chain by Pre-Analysis

For the sparse version to preserve the precision of the underlying original static analysis, the def-use chain information from a pre-analysis must be defined as follows.

Definition 5.4 (Safe def and use sets from pre-analysis) Let $D^\#(l)$ and $U^\#(l)$ be the sets of abstract locations, respectively defined and used by the original non-sparse abstract semantics of the program portion at label l . Let $D_{\text{pre}}^\#$ and $U_{\text{pre}}^\#$ be, respectively, those that are computed by the preanalysis.

We say $D_{\text{pre}}^\#$ and $U_{\text{pre}}^\#$ are safe whenever

- the def and use sets from the pre-analysis over-approximate those of the original non-sparse analysis:

$$\forall l \in \mathbb{L} : D_{\text{pre}}^\#(l) \supseteq D^\#(l) \quad \text{and} \quad U_{\text{pre}}^\#(l) \supseteq U^\#(l); \text{ and}$$

- all spurious definitions from the pre-analysis are included in the use set from the pre-analysis:

$$\forall l \in \mathbb{L} : U_{\text{pre}}^\#(l) \supseteq D_{\text{pre}}^\#(l) \setminus D^\#(l)$$

The precision-preserving def-use chain from the pre-analysis is defined as follows.

Definition 5.5 (Def-use chain information from pre-analysis) Let $D_{\text{pre}}^\#$ and $U_{\text{pre}}^\#$ be, respectively, safe def and use sets from a pre-analysis as in definition 5.4. We say that two labels a and b have a def-use chain

for an abstract location η whenever $\eta \in D_{pre}^\sharp(a)$, $\eta \in U_{pre}^\sharp(b)$, and η may not be redefined between the two labels: for every intermittent label c in the execution paths from a to b , $\eta \notin D_{pre}^\sharp(c)$.

Definitions 5.4 and 5.5 warrant that the resulting sparse analysis version should have the same precision as the original non-sparse analysis. Note that the def-use chain defines the “express highway” along which the sparse analysis delivers the abstract memory portions. If this express highway covers all the def-use edges of the original analysis, then the sparse analysis will compute the same analysis result as the original analysis because it uses the same abstract semantics as the original analysis.

The first condition of safe def-use sets (definition 5.4) is intuitive; under approximation may miss some def-use edges of the original analysis. We have to cover all def-use edges. Over-approximating the def sets (D^\sharp), however, may also result in missing some def-use edges. The second condition recovers the def-use edges that can be blocked by over-approximating definitions in D_{pre}^\sharp that are extra to D^\sharp . A definition can block a def-use edge if an extra definition is in an abstract execution path between the def-use labels. By enforcing such extra definitions to also be included in the uses, the def-use chain is recovered. Consider figure 5.10(a). The def-use edge for location η is from label a to label b because it is defined at a , used at b , and not defined at the intermediate label c . An over-approximating pre-analysis can conclude the η may be defined also in c , hence, instead of the def-use edge from a to b , edge from c to b is only possible (figure 5.10(b)). The second condition of definition 5.4 requires all such extra definition ($\eta \in D_{pre}^\sharp(c)$) to be reflected also in the use ($\eta \in U_{pre}^\sharp$), hence recovers the missing edge as in figure 5.10(c).

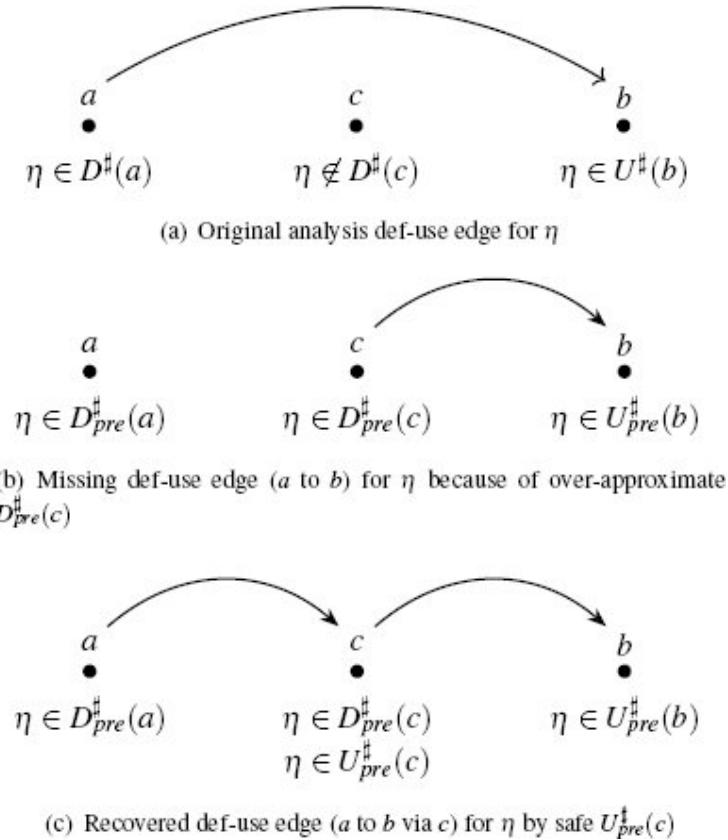


Figure 5.10

Def-use edge by over-approximating safe pre-analysis

5.4 Modular Analysis

Independently of the sparse analysis technique of the previous section, we can further improve the analysis scalability also by separately analyzing components of a program then stitching the componential results into the final result for the whole program. As compilers translate files separately and then link them into a whole-program translation, static analysis too can separately analyze components of a program and then stitch the componential results into a monolithic whole-program analysis result. Such static analysis is called *modular analysis*.²

Suppose that the unit of modular analysis is a procedure (function) in the input program. A modular analysis analyzes each procedure of the input

program and then links them into the whole-program analysis result. Modular analysis enables a kind of incremental analysis and the improvement of analysis precision. It computes an abstract summary of each procedure that is parameterized by its call context. Later, at link time, the analysis can derive an abstract post-condition sensitive (hence precise) to each call context. When a procedure is modified, the analysis has only to recompute its abstract summary and update the instantiations at its call sites.

As an example, consider a static analysis for detecting buffer overruns in C-like imperative programs. The analysis estimates the possible sizes of stack or heap-allocated buffers and the value ranges of the buffer access expressions. The analysis issues an alarm whenever it cannot prove that a buffer access always looks for a cell within the buffer area. Suppose we use intervals to over-approximate the range of index values and buffer sizes.

5.4.1 Parameterization, Summary, and Scalability

Separately analyzing procedures without knowing their calling context (pre-states) involves parameterization. We parameterize the calling context by symbols. From this symbolic pre-state, we analyze procedures. The result is summarized in terms of the parameter symbols for the calling context. Later, at link time, when the calling context is known, the analysis can finalize the actual analysis result by instantiating the summary with the actual calling context. This instantiation at link time makes the whole-program analysis scalable and precise; it avoids repeated analyses of procedures and entails context-sensitive results for each call.

Parameterization Consider a basic interval analysis that works with intervals $[l, h]$ whose bounds l and h are constants (integers or infinite symbols $-\infty$ and ∞). Forwardly analyzing each procedure in isolation needs a parameterization of the unknown calling context (pre-state). For unknown intervals in the possible calling context, we parameterize the interval's lower and upper bounds. We let the, if any, intervals of the pre-state have bounds as symbols. From these symbolic intervals of the pre-state, the forward analysis result of each procedure is the post-state in terms of the symbolic parameters and a condition on these parameters to ensure that all buffer access expressions inside the procedure stay within the target buffer area. Later, at link time, the symbolic parameters of the pre-state and the return state are instantiated and no-buffer-overrun conditions are resolved.

Summary-Based We analyze each procedure going forward starting from a symbolic pre-state (memory image) for its parameters and free variables. As it goes forward it gathers constraints that in the end play as a pre-condition that dictates in symbolic form the safety conditions for buffer accesses inside the procedure. When we resolve the symbolic safe conditions to be violated, an alarm is raised.

Scalability The scalability stems from the way that the modular analysis treats reasoning involving multiple procedures (interprocedural analysis). We compute a description of what a procedure does, its *summary*, in isolation from its calling context. The isolation is possible by parameterizing the calling contexts during the computation of the summary. We then compute an approximation of the global behavior of the program by stitching the procedure summaries in accordance with each call context.

When a procedure is modified, the whole-program analysis result is quickly obtained by updating only the parts that transitively depend on the procedure for which abstract states have changed. These local updates are done by instantiating the precomputed summaries of the involved procedures.

5.4.2 Case Study

Let us show how a modular analysis works by looking at simple examples. Suppose that the analysis goal is to estimate the sizes of buffers and the ranges of their indexing expressions. We examine the analysis summaries for a few procedures and then describe how they can be stitched together. We use the C syntax in example code.

First of all, let us consider a dereference that involves formal parameters:

```
void set_i(int *arr, int index) {
    arr[index] = 0;
}
```

For this procedure, the parametric context is for the parameters `arr` and `i`:

```
arr  ↪ (offset : [s0, s1], size : [s2, s3])
index  ↪ [s4, s5]
```

The safety condition is

$$[s_0 + s_4, s_1 + s_5] < [s_2, s_3].$$

The pointer parameter `arr` is an offset from the beginning of a buffer of some size. The offset is a symbolic interval $[s_0, s_1]$ whose lower and upper bounds

are, respectively, symbol s_0 and symbol s_1 . The buffer size and parameter i are again symbolic intervals. The safety condition for the buffer access $\text{arr}[i]$ is expressed as its offset $[s_0 + s_4, s_1 + s_5]$ (the pointer $\text{arr}[s_0, s_1]$ shifted right by i [s_4, s_5]) should be non-negative and less than the buffer size $[s_2, s_3]$.

Second, let us take a look at a malloc wrapper:

```
char * malloc_wrapper(int n) {
    return malloc(n);
}
```

For this procedure, the symbolic procedure summary would be

```
n  ↪  [s6, s7]
ret  ↪  (offset : [0, 0], size : [s6, s7]).
```

For brevity, we do not include the case that malloc fails. The return value is a pointer whose offset is $[0,0]$ and size is the parameter n that is a symbolic interval $[s_0, s_1]$.

Finally, let us take a look at the original example, but using `malloc_wrapper` in place of `malloc` and `set_i` in place of direct writes:

```
void interprocedural() {
    int *arr = malloc_wrapper(9 * sizeof(int));
    int i;
    for (i = 0; i < 9; i += 1) {
        set_i(arr, i); // safe
        set_i(arr, i + 1); // alarm
    }
}
```

For each call, the modular analysis instantiates the symbolic summary of procedures by the actual calling context and raises an alarm if it could conclude its, if any, safety condition false. From the actual parameter 9 in call `malloc_wrapper(9)` the symbolic interval $[s_6, s_7]$ becomes $[9, 9]$; hence, the array pointer `arr` becomes a pointer of offset $[0, 0]$ to a buffer of size $[9, 9]$. For the call `set_i(arr, i)`, the actual parameters `arr` and `i` instantiate the symbolic summary of `set_i` into

```
arr  ↪  (offset : [0, 0], size : [9, 9])
index  ↪  [0, 8],
```

and the safety condition is

$$[0 + 0, 0 + 8] < [9, 9].$$

The safety condition is true; hence, no alarm is raised.

For the second call `set_i(arr, i+1)`, the newly instantiated summary of `set_i` is

$$\begin{aligned} \text{arr} &\mapsto (\text{offset} : [0, 0], \text{size} : [9, 9]) \\ \text{index} &\mapsto [1, 9], \end{aligned}$$

and the safety condition is

$$[0 + 1, 0 + 9] < [9, 9].$$

The safety condition is false because 9 in $[1, 9]$ cannot be smaller than $[9, 9]$; hence, the analysis raises an alarm.

Note that the modular analysis can analyze the original procedure just as accurately when the specific dereferencing operations are abstracted by procedures, even though the procedures are analyzed independently. To do this we need an accurate enough notion of summary, as well as a good means of stitching together summaries.

5.5 Backward Analysis

While the previous chapters and sections have considered static analyses that start from over-approximations of sets of input states and compute over-approximations of a set of output states, we can actually design static analyses to compute over-approximate pre-conditions from a post-condition. This section provides an intuitive introduction to such analyses and to their applications.

5.5.1 Forward Semantics and Backward Semantics

The semantics used in chapter 2 consists of a set of reachable states. It is based on the assumption that the program always starts from a given set of initial states (the square $[0, 1] \times [0, 1]$ as fixed in section 2.1). Then the computation is a *forward* process that iteratively uncovers successor states. The semantics used in chapters 3 and in 4 is defined in the forward direction and computes successor states from predecessor states.

Let us recall the semantics in chapter 3. The semantics of expressions (figure 3.2) takes a set of states and returns the set of values the expression evaluates to; it is thus also defined in forward style. The semantics of Boolean expressions (figure 3.3) used in condition tests is similar and is thus also forward. However, we also defined the operation \mathcal{F}_B that takes as inputs a Boolean expression B

and a set of states and that filters out the states such that \mathbf{B} evaluate to **false**, keeping only those states such that \mathbf{B} evaluates to **true**:

$$\mathcal{F}_{\mathbf{B}}(M) = \{m \in M \mid \llbracket \mathbf{B} \rrbracket(m) = \text{true}\}$$

More generally, we could define $\llbracket \mathbf{B} \rrbracket_{\text{bwd}}$ and let $\mathcal{F}_{\mathbf{B}}$ be defined from it as follows:

$$\begin{aligned}\llbracket \mathbf{B} \rrbracket_{\text{bwd}}(v) &= \{m \in \mathbb{M} \mid \llbracket \mathbf{B} \rrbracket(m) = v\} \\ \mathcal{F}_{\mathbf{B}}(M) &= M \cap \llbracket \mathbf{B} \rrbracket_{\text{bwd}}(\text{true})\end{aligned}$$

The semantics $\llbracket \mathbf{B} \rrbracket_{\text{bwd}}$ is defined in *backward* style, as it takes a value (i.e., a result) and returns the set of states that lead to this value. Therefore, we call it a *backward semantics*.

We also observed in chapter 3 that the analysis of a condition test boils down to the computation of an over-approximation of the result of $\mathcal{F}_{\mathbf{B}}$, using an operation $\mathcal{F}_{\mathbf{B}}^\sharp$:

$$\text{for all conditions } \mathbf{B}, \text{ and for all abstract states } M^\sharp, \mathcal{F}_{\mathbf{B}}(\gamma(M^\sharp)) \subseteq \gamma(\mathcal{F}_{\mathbf{B}}^\sharp(M^\sharp))$$

Therefore, the analysis of a condition test is actually a form of *local backward analysis*. We give other examples in the rest of this section.

This notion of backward semantics also applies to other elements of our example language (or to other languages):

- The backward semantics of a scalar expression takes a scalar value and returns the set of states that lead to this value, just like the backward semantics of a Boolean expression.
- The backward semantics of a command takes a set of output states and produces the set of states that may lead to such outputs:

$$\begin{aligned}\llbracket \mathcal{C} \rrbracket_{\text{bwd}}(M) &= \{m \in \mathbb{M} \mid \exists m' \in \llbracket \mathcal{C} \rrbracket(\{m\}), m' \in M\} \\ &= \{m \in \mathbb{M} \mid \llbracket \mathcal{C} \rrbracket(\{m\}) \cap M \neq \emptyset\}\end{aligned}$$

We remark that the backward semantics of a command return a set of states that *may* lead to some of the output states. Due to non-determinism, an input state may lead to several executions, some of which lead to M and some of which do not. The above definition is based on the preference to not omit any state that *may* lead to M , but one may be interested in other kinds of pre-conditions, where only input states that only produce executions leading to M are included. We do not consider all possible choices in this book and focus on the above definition. The following paragraphs discuss the possible applications of such a semantics.

5.5.2 Backward Analysis and Applications

A first application of backward analysis is to discover how a program may reach a given state. This is a dual issue to the computation of an over-approximate post-condition for a given pre-condition considered in the previous chapters, but it is just as useful in practice.

An Example As an example, we consider the program \mathbf{C} shown in figure 5.11. It consists of a very simple absolute value computation. Variable x_0 is an input; thus, we assume that it may take any value (hence the **input** command), whereas x_1 stores the result. Obviously, the output states are such that x_1 is positive, and indeed, the abstract post-condition computed by the analysis of chapter 3 is $\{x_0 \mapsto \top, x_1 \mapsto [0, +\infty)\}$.

First, let us assume that we are interested in the abstract post-condition defined by the constraints $2 \leq x_1 \leq 5$. The backward semantics $\llbracket \mathbf{C} \rrbracket_{\text{bwd}}$ maps this post-condition to the set of states where x_0 is in $[-5, -2] \cup [2, 5]$ right after the **input** statement. Therefore, the most precise result that we can expect a sound approximation of this backward semantics to return is this union of intervals. However, if using the interval abstract domain, the result can only be an interval containing $[-5, -2] \cup [2, 5]$; thus, it may be at best $[-5, 5]$. The interpretation is that this range contains all the states that lead to an absolute value in $[2, 5]$. An analysis based on a partitioning abstraction (section 5.1.3) may return a more precise invariant that includes only the states where x_0 is in $[-5, -2] \cup [2, 5]$.

Now, let us assume that we are interested in the abstract post-condition defined by the constraint $x_1 \leq -3$. If we look at the concrete executions, we can observe that there is no input state that produces such an output state. Indeed, the result of the absolute value is necessarily non-negative; thus, no output state may satisfy the above constraint. In other words, when it is applied to the set of states such that $x_1 \leq -3$, the backward semantics returns the empty set. Therefore, we expect the backward analysis of this program to return \perp when applied to this program and the abstract post-condition $\{x_0 \mapsto \top, x_1 \mapsto (-\infty, -3]\}$.

```

int x0,x1;
input(x0);
if(x0 > 0){
    x1 := x0;
} else{
    x1 := -x0;
}

```

Figure 5.11

Backward analysis of a simple program

Applications This example illustrates a second application of backward example (besides the analysis of condition tests). By computing an over-approximation of the sets of states leading to some given behavior, it provides a *necessary* condition for this behavior to occur. Dually, it can also compute a *sufficient* condition for a given behavior *not* to occur:

- a necessary condition to return a value in [2,5] is to start with an input value that lies in [-5,-2] \cup [2,5];
- the absolute value never returns a negative value.

Such information may be useful in itself, for instance, for the purpose of program understanding. Indeed, given a large legacy program to maintain or extend, a developer may want to know when a program may return a given value. While this would be trivial in the case of the program shown in figure 5.11, large code may benefit from automatic analysis techniques. Information on pre-conditions may also be useful to help some other static analysis, as we address in section 9.1.2. We show another application of backward analysis in section 5.5.3.

Definition of a Backward Analysis After discussing the meaning and applications of the backward analysis, we study its definition. In chapters 3 and 4, the static analysis algorithms were derived from the definition of the concrete semantics. The same methodology applies here. We consider a few cases, using the same notations as chapter 3.

The backward semantics $\llbracket \text{skip} \rrbracket_{\text{bwd}}$ of a **skip** command is the identity function, just like the forward semantics. Therefore, if we let $\llbracket \text{skip} \rrbracket_{\text{bwd}}^{\#}$ also

denote the identity function, then it defines a sound approximation of $\llbracket \text{skip} \rrbracket$ **bwd**:

$$\llbracket \text{skip} \rrbracket_{\text{bwd}}^\sharp(M^\sharp) = M^\sharp$$

Similarly, the analysis of a sequence boils down to the composition of the analysis of each command, but in the reverse order compared to the forward analysis:

$$\llbracket C_0; C_1 \rrbracket_{\text{bwd}}^\sharp(M^\sharp) = \llbracket C_0 \rrbracket_{\text{bwd}}^\sharp(\llbracket C_1 \rrbracket_{\text{bwd}}^\sharp(M^\sharp))$$

The case of an assignment command is of course more complex. The definition of a backward analysis function shares some similarity with the case of condition tests discussed in section 5.5.1 although it requires a bit more work. Indeed, let us consider the assignment command $x := E$ and the abstract post-condition M^\sharp and discuss how to compute a sound abstract pre-condition:

- If x does not appear in the right-hand side E (we call such an assignment “non-invertible”), then we simply need to apply the operator $\mathcal{F}_{x=E}^\sharp$ and then forget all constraints on x (the assignment overwrites the value of x , so x may store any value in the pre-condition).
- If x appears in the right-hand side E (we call such an assignment “invertible”), constraints over the old value of x can usually be derived from constraints over the new value. As an example, let us assume that the analysis uses the interval domain, that E is $x + 2$, and that the post-condition is $\{x \mapsto [10,12]\}$, then a range over the old value can be computed by simply subtracting 2 to its bounds, so we obtain the abstract pre-condition $\{x \mapsto [8,10]\}$.

As one can observe, the definition of the backward analysis of an assignment command depends on the abstract domain and on the form of the right-hand-side expression.

The backward semantics of a condition command composes the effect of the branches with the effect of the test and combines both branches using set union (observe that the only difference with the concrete semantics given in chapter 3 is that the effects of the test and of the branches are composed in the reverse order):

$$\begin{aligned} \llbracket \text{if}(B) \{C_0\} \text{else} \{C_1\} \rrbracket \text{ bwd}(M) &= \mathcal{F}_B(\llbracket C_0 \rrbracket \text{ bwd}(M)) \cup \mathcal{F}_{\neg B}(\llbracket C_1 \rrbracket \\ &\quad \text{bwd}(M)) \end{aligned}$$

We derive the following abstract semantics:

$$[\![\text{if}(B)\{C_0\}\text{else}\{C_1\}]\!]_{\text{bwd}}^{\sharp}(M^{\sharp}) = \mathcal{F}_B^{\sharp}([\![C_0]\!]_{\text{bwd}}^{\sharp}(M^{\sharp})) \sqcup^{\sharp} \mathcal{F}_{\neg B}^{\sharp}([\![C_1]\!]_{\text{bwd}}^{\sharp}(M^{\sharp}))$$

As in the case of the forward analysis, the backward analysis of a loop command requires the computation of an approximation of a fixpoint. It involves abstract iterates (with widening if the abstract domain has infinite height) and is very similar to section 3.3.3, so we do not detail it here.

5.5.3 Precision Refinement by Combined Forward and Backward Analysis

We now present a third application of backward analysis and show that it can help refine results of a forward static analysis. To do this, we focus on a simple example that consists of a single execution branch (which we could think of as an excerpt of a larger program), shown in figure 5.12. Since we ignore all the other branches in the program, it may be more convenient to think of it in the transitional framework provided in chapter 4.

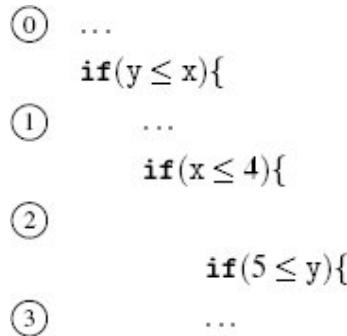


Figure 5.12

Example of forward backward analysis

First, we observe that the third true branch is *not* feasible; indeed, there exist no values for x and y that satisfy all three condition tests encountered over the branch. Therefore, we expect static analysis to prove that no execution reaches point ③ when starting from point ① (and whatever the assumption on the values of the variables at that point).

While a forward analysis with the abstract domain of convex polyhedra (definition 3.9) can establish this fact, the abstract domain of interval is too weak to prove it. Indeed, let us compare the results produced by these two analyses at each point:

| | Intervals | Polyhedra |
|---|--|----------------------------|
| ① | $\{x \mapsto T, y \mapsto T\}$ | T |
| ② | $\{x \mapsto T, y \mapsto T\}$ | $y \leq x$ |
| ③ | $\{x \mapsto (-\infty, 4], y \mapsto T\}$ | $y \leq x \wedge x \leq 4$ |
| ④ | $\{x \mapsto (-\infty, 4], y \mapsto [5, +\infty)\}$ | \perp |

This table clearly shows the difference between the two analyses: while the domain of convex polyhedra stores relations and ultimately demonstrates that the three conditions are not feasible in the same time, the domain of intervals drops the relation $y \leq x$ and thus fails. In general, any non-relational domain has no chance of success here.

Let us now consider intervals only and perform a backward analysis after the above forward analysis. More precisely, the backward analysis shall start with the abstract post-condition computed in the above forward phase: it simply assumes the abstract state $\{x \mapsto (-\infty, 4], y \mapsto [5, +\infty)\}$ at ④. This backward analysis provides another approximation of the set of executions that go through the branches of figure 5.12. The table below shows the results of the backward analysis, in the order in which they are computed:

| | Backward analysis |
|---|--|
| ④ | $\{x \mapsto (-\infty, 4], y \mapsto [5, +\infty)\}$ |
| ③ | $\{x \mapsto (-\infty, 4], y \mapsto [5, +\infty)\}$ |
| ② | $\{x \mapsto (-\infty, 4], y \mapsto [5, +\infty)\}$ |
| ① | $\{x \mapsto (-\infty, 4], y \mapsto [5, +\infty)\}$ |
| ① | \perp |

What happens here is that the backward analysis propagates range information that occurs late in the execution back to the beginning, where the test $y \leq x$ can clearly be proved infeasible even when using only intervals. As a result, we obtain the *empty* pre-condition. Applying a forward analysis again finishes to prove that no state can traverse this branch, as shown in the following table.

| | 1st forward phase | 1st backward phase | 2nd forward phase |
|---|--|--|-------------------|
| ① | $\{x \mapsto T, y \mapsto T\}$ | \perp | \perp |
| ② | $\{x \mapsto T, y \mapsto T\}$ | $\{x \mapsto (-\infty, 4], y \mapsto [5, +\infty)\}$ | \perp |
| ③ | $\{x \mapsto (-\infty, 4], y \mapsto T\}$ | $\{x \mapsto (-\infty, 4], y \mapsto [5, +\infty)\}$ | \perp |
| ④ | $\{x \mapsto (-\infty, 4], y \mapsto [5, +\infty)\}$ | $\{x \mapsto (-\infty, 4], y \mapsto [5, +\infty)\}$ | \perp |

The refinement process shown here can be useful both on small and on larger programs [96]. It can even be applied very locally to refine the analysis of tests,

following a method referred to as *local iterations* [51].

This forward-backward iteration process may be iterated as many times as required to improve precision. Of course, due to termination and cost consideration it should always be limited in practice.

Moreover, each analysis phase may use the previous one to compute more precise results. In particular, the backward analysis of some operations may be imprecise but could be improved by intersection with the results of the previous forward phase.

² This section mostly reuses the explanations and example code that initially appeared in <https://research.fb.com/inferbo-infer-based-buffer-overrun-analyzer/>.

6 Practical Use of Static Analysis Tools

Goal of This Chapter This chapter discusses the issues related to the use of a static analysis tool. It focuses on the user perspective on the analysis and deliberately stays away from the internals of analysis tools, which are the topics of other chapters. It does not substitute for the manual of a specific static analyzer but provides guidelines on how to better understand it, so as to efficiently and correctly deal with the deployment of static analysis techniques.

Recommended Reading: [S], [D], [U] This chapter is mainly targeted at developers and engineers who are using or planning to use static analysis tools to verify software projects. To some extent, notions exposed in this chapter have been covered from a more theoretical point of view in chapters 3 and 4, so students who feel comfortable with the practical implications of the frameworks exposed in the previous chapters may skip this one. We also recommend the reading of this chapter to designers of static analyzers who would like to better plan the interaction with their analyses.

Chapter Outline In section 6.1, we study the adequacy between a verification goal and a static analysis tool. Section 6.2 addresses the task of setting up the analysis of a target program, including the choice of hypotheses about the program that need to be taken into account. Section 6.3 describes the exploitation of results produced by a static analyzer. Section 6.4 studies the deployment of static analysis tools in a broader context.

6.1 Analysis Assumptions and Goals

In the next few subsections, we consider a language L (with a syntax and a concrete semantics) and a semantic property \wp that are fixed. To fix the ideas, L could be the Java language, and \wp could be the absence of

uncaught exception, including null pointer exceptions or array-out-of-bounds exceptions (these exceptions cannot be caught at run-time and will cause the program to crash). We also assume that a static analyzer A is available that inputs programs written in language L .

Depending on the application domain, soundness with respect to all executions may be absolutely required or not. For instance, in the case of safety-critical software, it is very important to make sure that certain kinds of errors will never happen. Other application domains may not require such a stringent verification yet benefit from a partial verification that omits certain aspects and is not sound with respect to the actual semantics of programs. Regardless of the intended application domain, we consider it very important to know when analysis results allow deriving certain semantic facts, so that users are able either to require a fully sound analysis or to decide what behaviors the analysis may ignore.

The crucial question, before even running A to verify some actual program, is to what extent the results that A is going to produce will allow actually establishing the property of interest ρ . In chapters 3 and 4, we showed abstract interpretation-based static analyzers that come with a soundness theorem; this theorem expresses the correctness property guaranteed by the analysis. For example, theorems 3.6 and 4.2 state that the analysis accounts for all outputs that can be produced by a program. Therefore, we discuss the soundness property of A in the following paragraphs, so as to determine to what extent A is able to establish the property ρ .

Analysis Target Properties In general, the soundness property of a static analyzer is of the form “for any program p that satisfies some assumption H , the analysis A computes a semantic property of the form S that is satisfied by all executions of p .” To check that A can be useful to verify ρ , we need to make sure that there exist properties of this form S that entail the target semantic property ρ . On the other hand, if the results of the analysis are not logically strong enough, there is no chance that it ever allows proving ρ .

Let us consider the verification of absence of uncaught exceptions in Java programs as an example target property. If A computes precise information about the integer values that a program manipulates, and

produces range information over indexes in array lookup and write operations, then we can hope for A to prove the absence of index-out-of-bounds exceptions. On the other hand, if A only computes information about the reachable program points (i.e., the program locations that may be visited during program executions), then there is no hope that it proves a reachable lookup instruction $a[i]$ will never fail with an index-out-of-bounds exception.

Source Language Semantics The second point to confirm is that A applies to the class of programs that are under consideration and that it guarantees sound results for them. The semantics of a programming language is often so complex that it is very hard to design static analysis tools that follows it completely without many important assumptions. Let us consider some examples.

Most programming languages support single-precision and/or double-precision floating-point arithmetic, as described in the IEEE 754 standard [5]. This standard specifies the format of the floating-point representation, and the arithmetic operations, with various rounding modes. The actual results depend on the rounding mode (which may be “to the closest value,” “toward 0,” “toward $+\infty$,” or “toward $-\infty$ ”) of the machine on which the program is executed. In fact, the situation may be even more complex, as compilers sometimes perform optimizations and may evaluate expressions made of constants. Then one should take into account the rounding mode of the machine used to compile the program. When analyzing a program that makes extensive use of floating-point data, it is important to make sure the static analysis tool accounts either for all possible rounding modes or at least for the rounding mode(s) used for the evaluation of the program. Failing to do so may lead to unsound results, such as floating-point ranges that do not include all the values one may observe at run-time.

The low-level description of the memory layout provides a second example of programming language feature that may need to be handled in a specific way. Let us consider the C programming language. The C semantics as defined in the standard of the language [4] does not fully specify the representation of data, the size of data types, and the alignment of structure fields. To understand the physical representation of each entity manipulated by a given program, developers also need to refer to the

Application Binary Interface (ABI) and to the documentation of the compiler that they use. This issue is well known, since it often makes the portability of software (action to adapt a given program to different host architectures) quite challenging. Indeed, addresses computed by pointer arithmetics are likely to evaluate to different locations, depending on the architecture, which means that the programs may also behave differently. Some developers do actually make use of such features intentionally, for example, when they know the target architecture precisely, and when they write programs that will interact with other low-level software. From the static analysis point of view, this dependency of the physical representation makes it very difficult to produce as a result of a single analysis some abstract invariants that precisely account for all behaviors of a program, for all possible target architectures. There exist several ways to cope with this issue:

- One may let the analysis depend only on the C standard and reject any program the correctness of which depends on architecture-specific or compiler-specific assumptions.
- On the other hand, one may build and use a static analysis tailored for a specific architecture or set of architectures and that is able to very precisely take into account this assumption.

The execution order provides a third and last example. While some languages fully specify it (as Java does), others do not. The C language falls in this case and leaves the execution order of many expressions undefined. This means that when a program contains the expression $f(&x) + g(&x)$, the language semantics does not fix which sub-expression ($f(&x)$ or $g(&x)$) is evaluated first. The intent of the language designers is to let future compiler developers implement optimizations that cannot be foreseen when the language is designed and that would require modifying the evaluation order. As a consequence, the effective execution order may vary from one compiler to another and not even be consistent in a given compiler run. From the point of view of the design of a static analysis tool, the main solutions are

- to let the analysis account for all possible orders (though this is likely to be very costly and to reduce precision);

- to consider only one execution order that is fixed in the soundness statement of the tool, and openly ignore all other possible execution orders; then the analysis is sound only with respect to that order;
- to check that the analyzed program does not contain expressions the result of which depends on the execution order, and to select a fixed order to be considered for the analysis; in that case, programs that may follow a different order are explicitly rejected during the analysis.

In general, it is the responsibility of the user to make sure that the semantic assumptions of the analysis tool are compatible with the intended use of the analyzed program. As an example, when using a known compiler that follows a known evaluation strategy, a static analysis tool following the same execution order can be used safely. On the other hand, when the order is not known, this approach may lead to unsound results.

Back to Soundness The gist of the two previous paragraphs boils down to the soundness theorem that the static analysis tool A satisfies. In this context, the soundness theorem may either take the form of a formal mathematical statement, such as theorems 3.6 and 4.3, or may consist of a documentation in plain English that states formally what the analyzer computes and under which assumption. It usually takes the following form.

Theorem 6.1 (Soundness) *The static analyzer A is sound with respect to the subset of programs L_{Snd} (where $L_{\text{Snd}} \subseteq L$) and to the subset of executions E_{Snd} (where $E_{\text{Snd}} \subseteq E$) in the following sense:*

*For all programs p in L_{Snd} ,
for all execution traces e in E_{Snd} ,
if e is an execution of p ,
then $\mathcal{A}(p)$ fully accounts for the execution e .*

A theorem of this form precisely characterizes what programs and program behaviors A is able to account for and what it ignores:

- Programs outside L_{Snd} are unsupported and either they may be rejected outright or their semantics may not be covered (e.g., if the analyzer rejects or ignores commands with an unknown execution order, programs that contain such patterns are outside L_{Snd}).

- Executions outside E_{Snd} may not be covered (e.g., if the analysis assumes a specific execution order and ignores all other orders, then traces corresponding to other execution orders are outside E).

Generally, a static analysis tool that defines L_{Snd} and E_{Snd} precisely is much more useful than one that does not make these clear, in the sense that it allows users to assess whether the programs they are interested in are covered and ultimately what was actually proven. We may observe that theorems 3.6 and 4.4 come with no such restriction. This is because the language considered in chapter 3 is simple and requires no such limitation. On the other hand, real-world programming languages typically have a much more complex semantics, which would make it extremely hard to build a static analysis tool that accounts for all executions without any restriction.

From the user point of view, when the correctness of the target software is critical, one should make sure that the limitations mentioned in theorem 6.1 are acceptable and satisfied by the target software. Otherwise, users may fail to understand that a static analysis may incorrectly handle an incorrect program due to invalid assumptions. On the other hand, when static analysis is aimed only at improving the quality of non-critical software, such well-understood obliviousness is usually not problematic.

Static Analyzer Implementation Bugs and Soundness As we have been commenting on the soundness of an analysis tool and its application in verification to catch some classes of bugs, an important question is what happens if the tool A itself contains implementation bugs. Obviously, the main concern is that the verification will emit invalid results, as this could cause software defects to be missed. The situation is a bit similar to that of a buggy compiler that emits incorrect code and causes issues that are very hard to diagnose: a programmer may rightfully consider the source code of a program correct and then struggle to understand why the compiled binary has unexpected behaviors. Therefore, we now discuss the correctness of the *implementation* of a static analysis tool and the possible consequences of implementation bugs inside the static analyzer itself.

Not all static analyzer bugs are dangerous. For instance, let us consider the case of an implementation bug that causes the analyzer A to crash. This

bug will not cause the analyzer to inaccurately claim a target program is correct when it is not. Indeed, a user should consider that a static analyzer crash amounts to rejecting the program. On the contrary, such a bug may only prevent correct programs from being verified correct (which is problematic, but less so than issuing a wrong claim that the property of interest holds). The consequences of an implementation bug that reduces the precision of the analysis are similar.

Soundness bugs are bugs of A that may cause it to issue a wrong result, which violates its (formal or informal) soundness guarantee. As an example, a static analyzer that is supposed to discover all run-time errors and fails to report a possible division by zero would be unsound. Developers of static analysis tools usually do their best to avoid such issues, but they are hard to avoid completely since static analyzers are often complex pieces of software. From the user perspective, several techniques may reduce this risk, such as

- the use of several tools that were designed and implemented independently, so as to compare their results; and
- the testing of the analyzer on small well-understood benchmarks to become more confident that the results meet the expectations.

Additionally, some techniques may be applied to the design and implementation of static analyzers, and increase the confidence in their results. One approach consists of verifying the soundness of the static analyzers A using, for instance, assisted proving techniques discussed in section 1.4.2 [65]. Another approach lets each run of A issue a certificate that the results it produced are correct (then the certificate may need to be checked using an external, independent tool). Last, certain tools are provided with a “qualification kit” which comprises additional documentation and test cases to let users make sure the tool returns satisfactory results on their own production code. From the user point of view, and when verifying safety-critical code (possibly with the intent of certifying it), it is often a better choice to select an analysis tool that benefits from either of these techniques.

Analysis Precision and Partial Completeness While the previous paragraphs focused on the soundness of the analysis (i.e., whether it delivers results that can be trusted), users should also care about the ability of the analysis

to compute abstract semantic facts that are sufficiently precise. In particular, to be useful, the analysis should actually be able to prove the property of interest often enough. Indeed, soundness may be easily met by an analyzer that always returns an overly coarse approximation of the program semantics, but such an analyzer would be useless from a user standpoint. Therefore, the *precision* of the analysis, namely, its ability to establish the property of interest in many cases, is of great importance in practice.

Ideally, users who have a well-identified family of software to analyze would wish to rely on a static analysis tool that is *complete* with respect to this set of programs, which means that it returns abstract results that are always conclusive for these programs. Due to undecidability (see section 1.3.3), completeness with respect to all input programs cannot be achieved in the same time as soundness and automation. The notion of *partial completeness* with respect to a family of input programs weakens that of completeness by focusing on a given set of programs to analyze. In theory, completeness is achievable for a given program, which means that if one program to analyze is given, it is possible to craft a static analysis tool that is able to cope with it. However, this analyzer would of course not be complete for all possible input programs, which means that it would still fail on infinitely many others. This result generalizes to a finite set of input programs. However, it does not extend to infinite sets of programs, which is of course the common case.

In practice, any analysis is a compromise between the precision of the results and the resources (time and memory) it requires. Parameterizable analysis tools can generally be tuned to either use simpler abstract predicates, and ideally return faster, or use more complex abstract domains for better precision, yet possibly at the cost of increased time or memory consumption. Such parameters are discussed in section 6.2.2. Furthermore, in some cases it is also possible to trade some coverage of program inputs for faster execution, based on the soundness under assumption of theorem 6.1: by adopting assumptions that restrict the set of executions that are considered during the analysis (e.g., by using narrower input ranges, or by considering only certain execution orders among others), one may obtain both better run-time and better precision. In particular, this approach is often acceptable in noncritical applications or when all executions of interest are still covered.

However, an important remark is that many static analysis tools rely on an abstract interpretation function that is *not* monotone (due to the computation of widening sequences, or to the use of nonmonotone abstract operators as observed in chapter 3), which means that increasing precision locally (using more accurate predicates or restrictive assumptions) may fail to improve the overall level of precision. Similarly, the consumption of resources by a static analysis tool does not always increase when considering more sophisticated abstract domains and analysis algorithms. In some cases, improving precision (possibly by the means of an apparently more costly analysis) may successfully prove certain states unreachable, hence reducing the amount of work the analysis needs to do for a given program. These remarks mainly show that tuning an analysis for precision and more efficient use of resources is a hard task in general that requires a good understanding of the analysis tool and of the trade-offs that its parameterization implies. We discuss these issues further later in this chapter.

6.2 Setting Up the Static Analysis of a Program

We now assume that the static analysis tool A is adequate to compute a sound abstraction of the semantic property of interest, and we discuss the next task, namely, the configuration of the analysis and its execution.

Configuring a static analysis tool can be a complex task. Thus, we consider only the main aspects. We distinguish two sets of parameters: first, section 6.2.1 focuses on soundness-critical points such as the definition of the code to analyze; second, section 6.2.2 presents parameters supposed to guide the analysis and that may affect its performance and precision by tuning the abstract domain and iteration strategy.

6.2.1 Definition of the Source Code and Proof Goals

The most important parameters of a static analyzer designate the code to analyze, the properties to attempt proving, and the assumptions that are available to do that. We first discuss programs and consider goals and assumptions afterward.

Form of the Program to Verify: Complete versus Incomplete In the following, we distinguish two common situations, depending on the form of the analysis target program:

- We call *whole-program analysis* the static analysis of a program, the source code of which is fully available, and that should be verified in its own specific context.
- Conversely, we call *program-fragment analysis* the static analysis of a software component such as a library, that is meant to be called in a context specified by a general API (application programming interface).

The difference may seem subtle, yet these call for dissimilar ways to configure the analysis.

Whole-Program Analysis A whole-program analysis may take place when the full program text is available. This case suits well the verification of a software application at the end of its development and before it is shipped. Indeed, a complete piece of software generally consists of a well defined set of functions with a known entry point, and the initial states and inputs are also usually well specified. In the case of a whole-program analysis, we can supply the analysis tool with

- all the program files (e.g., as a bunch of C files);
- the entry point (e.g., the function `main`); and
- information on all possible inputs (e.g., input ranges).

Depending on the tool, the files may need to be in a specific format. For instance, unprocessed C code is often not supported by static analyzers for the C language; then users should make sure to preprocess their code, if possible with the preprocessor that is meant to be used for compilation.

The entry point usually boils down to the selection of a function.

Information on inputs comprises ranges for all input variables of the program, and also an abstraction of the content of the initial memory states. For instance, let us discuss the case of C programs. Depending on the compiler and the program loader, static memory locations may be initialized to zero (initialization to zero of the static space of the program), or simply to whatever value they store when the program starts. Either way, the users should specify what assumption the static analysis should make. Failure to do so would lead to results that do not soundly account for the program

executions being considered. In general these assumptions may be either encoded as plain assertions in the program to analyze or communicated to the analysis tool as external parameters.

Analysis of Program Fragments A program-fragment analysis considers a piece of code that is not complete and hence cannot be executed as is. For instance, when an application is incomplete because its development is not finished yet, or because a part of it is developed separately, we may not be able to supply all its components to the analysis tool. Libraries also fall into this category: by nature, a library comprises functions meant to be called by a client application, and the internals of such client codes are unknown when the library is developed. In this case, the data that should be supplied to the static analysis are a bit more complex to define, and include

- the available portion of the code;
- the definition of all the possible entry points for the part of the code that is available (usually, there are several of them, especially in the case of a library);
- the definition of the states that are expected for each entry point; and
- information of all other possible inputs.

Usually, most of these pieces of information are available as part of APIs and documentations of software fragments. The specification of the code to analyze, of the entry points, and of the expected inputs is similar to the case of a whole-program analysis.

However, the definition of states that are expected for each entry point is usually much more complex than for a whole application. For instance, let us consider the case where the program to analyze is a library that implements a container data structure (e.g., a hash table). Such a library typically provides functions to initialize the container data structure, to add or remove an element, or to iterate over the structure. Each of these should be called only in states that meet some specific properties. For instance, a removal function may be applied only to a well-formed container that contains the element to suppress. As a consequence, one should communicate to the analysis of each library function assumptions about the state of the container. Depending on the analyzer, two main approaches can be considered:

- use an interface provided by the analyzer to specify the initial states; or
- write a code harness that creates a valid initial state before calling a library function; this technique essentially reduces the analysis of a program fragment to that of a whole program, including the harness plus a library function (and of course, if the library has several entry point, its verification may require the implementation of several harnesses as well).

Bottom-Up and Top-Down Analyses The distinction between whole-program analysis and program-fragment analysis may have other consequences for the way the analysis is performed. There are several ways to analyze programs that consist of many functions or procedures. *Top-down* analyses start from the entry function and analyze other functions whenever a call site is encountered. *Bottom-up* analyses start with basic functions and do not reanalyze functions in their calling contexts. In general, top-down analyses are more relevant in the case of a whole-program analysis, whereas bottom-up are more naturally adapted to the analysis of program fragments, even if it is entirely possible to apply top-down analyses to each entry point of a library to verify, or to analyze a whole application in bottom-up style. Top-down analyses can more easily make use of the calling contexts of each procedure. The main advantage of the bottom-up approach is to compute a summary per procedure (as shown in section 5.4, in the case of a buffer overflow detection analysis) that can be reused for each call site, thereby alleviating the need to reanalyze procedure bodies multiple times. On the other hand, procedure summaries often require more sophisticated abstract domains. In particular, taking procedure calling contexts into account requires more involved procedure summaries. Procedure abstractions are defined and compared more in detail in section 8.4.1.

Calls to External Code Another aspect related to the availability of the code is when the program to analyze calls some external functions that we cannot provide to the analyzer. This is the case when analyzing a whole program or a program fragment that calls one or several external libraries, or system calls. A possible reason not to provide these components is that they may be very large and would increase the analysis time too much.

Another possible reason is that they may be available only as precompiled binaries.

The standard approach to solve this case is to rely on *function stubs* or, for short, *stubs*. A stub is a function mockup that has the same signature as the function it simulates but that does not contain the full code. Instead, it just describes assumptions and assertions that can be understood by the analyzer, so as to faithfully account for the effect of the real function. For instance, let us consider a function that computes the square root of an integer argument. Clearly this function should apply only to non-negative values. Moreover, it will always return a non-negative result. Therefore, its stub should contain an assertion that its input is non-negative, and its result may be any possible non-negative integer. Of course, this description is overly conservative; however, we may hope that it is precise enough for the analyzer to reason on the program that calls it.

Selection of Analysis Goals Before starting the analysis process, the user should also make sure the analysis will work toward the right property to verify. Depending on the analyzer, there are several ways to achieve this.

Most commonly, some analyzers are dedicated to the verification of a specific property, and no parameterization is required in order to specify this property. For instance, some static analysis tools are designed specifically for the verification of the absence of a specific set of run-time errors, for the verification of termination, or for computing a conservative approximation of the set of reachable control states.

Sometimes, other tools require properties to be specified as code. For instance, programming languages such as C or Java feature an assertion mechanism, which can encode properties of the form “whenever a given program point is reached, some given condition over the values of the variables of the program should hold.” In this case, the properties to verify should be integrated into the program and be defined with the same semantics as the source language itself.

A last approach relies on external specification languages. In that case, the documentation of the static analysis tool defines the syntax and semantics of a language of assertions that remain external to the program to analyze. The user should comply with this definition and make sure to encode faithfully the property of interest.

In all cases, and as we noted in section 6.1, it is important to ensure the analysis will tackle the right property, so as to prevent it from producing results that are unsound or irrelevant.

6.2.2 Parameters to Guide the Analysis

In addition to the program and property to verify, static analyzers often take additional parameters to tune the way the analysis is performed and its overall cost-accuracy performance. The next paragraphs review a few common such parameters, and afterward we discuss how they can be chosen.

Selection of the Abstract Domain Some static analysis tools implement several abstract domains and may be set to use one or another. As we have shown in the previous chapters, the abstract domain fixes the set of logical predicates to be used for the analysis, their computer representation, and the algorithms to reason over them (e.g., to compute over-approximations of post-conditions and unions of sets of states). Therefore, the choice of the abstract domain has huge impact on the way the analysis performs. Choosing too expressive an abstract domain may cause the analysis to take too much time and/or too much memory. On the other hand, if the abstract domain is not expressive enough, the analysis is likely to fail to prove the intended results. In fact, using an abstract domain that is not adapted may cause not only the analysis to lack precision and to fail to prove the property of interest but also to explore a far larger state space as a consequence of that, which means it may also consume too much time and memory.

In sections 2.2 and 3.2, we introduced a few common abstract domains that can be used by analyzers that only need to manipulate numerical constraints:

- Constants and intervals define non-relational abstractions (an abstract value cannot capture a constraint over two variables).
- Convex polyhedra and octagons define relational abstractions (some constraints over several variables may be expressed).

These remarks illustrate the considerations that should guide the selection of the abstract domain: polyhedra bring high precision but are costly (the number of constraints is not bounded by a function of the number of

variables), whereas the non-relational domains mentioned above are strictly less expressive but much cheaper. Some static analysis tools such as ASTRÉE [12] feature a switch that allows selecting a specific abstract domain. The benefit is to tune the precision/cost ratio.

Additionally, section 5.1 described ways to define even more expressive abstract domains from basic ones. For instance, we presented reduced product of abstract domains as a way to express conjunctions of constraints defined in distinct domains. We also presented partitioning abstractions that allow capturing implication properties, and to partition states or traces so as to gain expressiveness. Some static analysis tools allow not only selecting the abstract domain but also constructing it so as to capture a wider class of logical properties.

Furthermore, it is also possible to apply a more expressive and expensive abstract domain in a local manner, using *packing*: the idea is to cut down the cost of the expensive abstraction by limiting the number of variables this expensive abstraction is applied to, to a small group called *pack* [12]. The choice of the packing strategy has no effect on soundness (whatever the packs, the analysis produces an over-approximation of the behaviors of the program) but may affect precision or cost.

Iteration Strategy Handling loops (or, equivalently, recursion) precisely and at a reasonable cost is one of the most challenging issues in static analysis. In section 5.2, we showed a few techniques that improve the precision of the analysis of loops. These techniques boil down to simple additional parameters:

- Loop unrolling (section 5.2.1) unfolds the first iterations and postpones the application of abstract join and of widening to a later stage of the analysis of loops.
- Delayed widening (section 5.2.2) lets the analysis make use of a more precise abstract join (instead of widening) for a few iterations.
- Widening with threshold (section 5.2.2) slows down the loss of precision incurred by widening, so as to search more precise stable constraints.
- Post-fixpoint refinement iteration (section 5.2.3) lets the analysis continue after a post-fixpoint has been computed, so as to refine it

by computing a few additional iterations.

Each of these technique (as well as others) may be employed and controlled by specific analysis parameters (number of unrolled iterations, abstract joins and post-fixpoint iterations, number and value of widening thresholds, etc.).

Analysis Coverage and Time-Outs The amount of resources required by a static analysis run is an important consideration. It may range from a nonmeasurable amount of time and memory in the case of simple analysis to days of computation on powerful machines when attempting to verify strong semantic properties on large software. We can cite various strategies that may be applied in order to bound the duration when the analysis takes too long. For instance, *time-outs* may specify an upper bound on the analysis time, so as to keep the analysis cost under control. Failure to terminate within the time-out bound would be interpreted as a failure to establish the property of interest. Thus, it would not be unsound. Another approach is to let the analysis deliberately ignore some well-specified parts of the program, or behaviors, thereby relying on a specific instance of soundness under assumption (theorem 6.1).

Analysis Outputs Static analyzers also commonly feature options that affect the analysis output. One option is to let the analysis produce only basic (e.g., “pass”/“fail”) results that merely say whether the property of interest could be proved or not. On the other hand, more verbose logs provide additional information and are usually required for the user to fully understand why a given run of the analyzer falls short of proving the property of interest. The level of verbosity in analysis logs impacts both the amount of resources required (due to the time to produce the outputs and to the space to store them) and the usability of the analysis (as we address in section 6.3).

Automatic Parameter Selection Methods While many static analysis tools feature manual parameterization methods, automatic parameterization is often the preferred choice for most users. Indeed, manual parameterization requires both time and expertise, especially when using an analyzer that is highly parameterizable. Therefore, we list the main techniques for automatic selection of static analysis parameters.

- **Syntactic pre-analysis:** Before the analysis starts, a preprocessing phase searches the code to analyze for some patterns and selects abstract domain and iteration strategies to use accordingly. For instance, this approach is extensively used in the ASTRÉE analyzer [12]. Since the pre-analysis is syntactic, it may not be robust enough to identify an accurate parameterization.
- **Machine-learning-based pre-analysis:** This approach lets the static analysis tool learn how to select its parameters efficiently thanks to precomputed relations among program syntactic features, analysis parameters, and corresponding results [18]. Thus, it defines a more systematic way to construct syntactic pre-analyses.
- **Semantic pre-analysis:** This also takes place before the analysis but consists of another semantic static analysis that is generally much simpler than the main analysis [86]. This approach is often more robust than syntactic pre-analysis but harder to design.
- **Semantic online analysis:** A limitation of pre-analyses is that they cannot make use of all the semantic information computed by the main analysis (before they take place before it). Thus, another approach lets the main analysis parameterize itself. Such approaches are hard to design but can be very powerful. As an example, the co-fibered abstract domain [108] construction lets a static analysis select dynamically the kind of abstract predicates it requires. For example, it is possible to automatically adapt the widening strategy based on semantic information from the abstract states ([78]).

Automatic versus Manual Parameterization When an analysis based on automatic parameterization fails to provide the expected level of precision and performance, manual parameterization comes in handy. Depending on the tool, it may involve

- adding command line parameters or filling in equivalent settings in a graphical user interface; or
- inserting directives as comments into the code of the program to analyze.

In practice, a good approach consists of using automatic parameterization first and tweaking the analysis parameterization manually afterward. From the user point of view, the process is as follows:

1. Carefully set the parameters related to the definition of the code to analyze and property to verify.
2. Leave all the analysis-guiding parameters set to default/automatic values, and run the analysis.
3. Inspect the results (section 6.3).
4. When the analysis takes too long or is too imprecise, investigate the reason for this.
5. Attempt to remedy this with alternative manual parameters, and run the analysis again.

This method enjoys the advantages both of automatic parameterization (less work or expertise required for the first analysis) and of manual parameterization (ability to better tune the analysis).

6.3 Inspecting Analysis Results

The previous section discussed the parameterization of a static analysis before launching it, and we now turn to the next stage in the analysis workflow, which aims at exploiting the results produced by the analysis either to conclude the verification or to understand how to improve the analysis parameterization so as to obtain more satisfactory results.

Analysis Success or Failure to Establish the Property of Interest The main output most users are interested in is whether the static analysis could achieve the proof of the property of interest \wp on the input program. In all cases, when the tool is supposed to be *sound*, the answers are always conservative: when the analysis claims it successfully proved the property \wp , we can naturally trust it holds, but when the analysis reports a failure to prove it, the property may hold or may not hold. When \wp is global (e.g., “does the whole input program terminate?”) and cannot be split into simpler ones, this result boils down to a “yes” or “no” answer. When \wp consists of a conjunction of local properties, the analysis result also splits down into smaller pieces of information. As an example, let us assume that \wp denotes the absence of run-time errors in the input program p . Then we can define the set $O =$

$\{o_0, \dots, o_n\}$ of operations of p that are potentially dangerous and may trigger a run-time error. Then \wp writes down as “no execution crashes due to o_0 and no execution crashes due to o_1 and ...” so that the analysis result also splits down into a series of “yes” or “no” answers.

In the following, we call *alarm* a negative report issued by the analyzer, which means that the analysis failed to prove the property of interest either as a whole (in the case of a global property) or because one of its components could not be verified (in the case of a conjunction of local properties). The next paragraphs discuss the meaning and consequences of alarms raised at the end of a static analysis and how to investigate them.

Semantics of Alarms and Soundness In general, an alarm indicates not only that the analysis fails to prove part or all of the target property but also that a loss of precision is taking place.

As an example, we consider the case of a static analysis that attempts to verify the absence of out-of-bound array accesses in C programs. When a C program attempts to write into an array, using an index value that does not fit its bounds, several behaviors are possible. The program may either crash due to a segmentation fault or write into a memory location that is not within the array. In the latter case, the execution will continue, albeit with a corrupt memory state. Identifying the variable or structure stored at the memory location that gets corrupted is very difficult in general. From the static analysis perspective, it is very hard to account for such executions where the memory was corrupted. Thus, a reasonable approach is to let the analysis report an alarm when it cannot prove an array access is safe, and to “cut out” the corresponding executions, which means the analysis reports all possible array index out-of-bounds accesses but does not say anything about what happens after such an incident. At a high level, this analysis behavior is somewhat similar to a parser that stops at the first error it encounters, as error recovery would be hard to even specify. However, in the case of static analyzers, it happens only for the executions that cannot be proved correct. While this may look unsound (as a crash occurring after several occurrences of out-of-bounds accesses may go unreported), it is actually sound in the following sense: any program execution is either correct or exhibits out-of-bounds accesses, and in that case, the analysis will report at least one unverified array access operation. This analysis behavior

is an instance of the notion of soundness under assumption described in theorem 6.1.

The gist of this discussion is that the user should pay attention to the way the static analysis resumes after an alarm:

- If it continues with an over-approximation of the possible states, the analysis still covers all executions, even after an erroneous behavior that corrupts the state.
- If it cuts out executions, the results after the alarm point are valid only for executions that do not exhibit such an erroneous behavior.

Additional Analysis Outputs In addition to alarms, static analyzers also often produce more extensive analysis logs. Logs are very useful to understand the alarms produced by static analyzers. There are two main kinds of logs:

- Logs that are produced at the end of the analysis can summarize global information, such as sound invariants that describe an over-approximation of all states at all program points.
- Logs that are produced during the analysis can only describe partial information that is made available at the time they are displayed.

Alarms Triage As mentioned above, the fact that the analysis emits some alarm does not mean the analyzed program does not satisfy the semantic property of interest. Indeed, each alarm may be

- a *true alarm* that indicates a real defect of the program, and that can be observed on at least one concrete execution; or
- a *false alarm* that is due to the imprecision of the analysis.

When the analysis emits a number of alarms, the next step in the verification process is to diagnose each alarm so as to decide whether it is a true alarm or a false alarm. We call this process *alarm triage*.

Alarm triage allows not only understanding whether or not the analyzed program satisfies the property of interest but also finding out how to better configure the analysis (e.g., by manual parameterization, as discussed in section 6.2) or by improving the static analyzer itself (as discussed in chapter 5). When an alarm turns out to be false, it is thus important to determine its origin:

- If it is due to overly imprecise assumptions provided in the analysis setup (as discussed in section 6.2), then we can simply fix these assumptions and resume the analysis.
- If it is because the abstract domain of the analyzer cannot express crucial proof steps, a more expressive abstract domain should be selected (if available) or implemented (this case is beyond the scope of this chapter).
- If it is due to imprecisions that stem from the iteration or the analysis algorithms, alternative parameters should be used so as to better guide the analysis (if available), or the analysis has to be reimplemented (again, in this chapter we consider only the user side of static analysis and do not discuss the design and implementation of the analyzer).

Ideally, the triage of an alarm should produce either a concrete counterexample that characterizes the problem or a set of logical arguments that the analysis missed and would allow completing the proof. Alarm triage is often a rather tedious and time-consuming task; thus, we discuss three families of techniques to cope with it. In practice, one often needs to combine several of these techniques.

Manual Inspection The most basic method relies on a manual search through analysis logs to localize a possible loss of precision or make sure that the alarm corresponds to a real issue. To do this, the user should look at invariants computed during the analysis and interpret them so as to decide whether they show some imprecision that may explain the analysis failure.

In general, manual inspection requires some expertise from the user, but a few basic techniques allow progressing more quickly. For instance, it is usually better to start with the first alarm that is reported, following the order of the analysis computation. Indeed, when the analysis reports an alarm, it is often the sign that it starts losing precision, and this loss of precision may cause a cascade of alarms by domino effect. Thus, understanding the first report often gives a clue about other subsequent alarms. Moreover, invariants at critical points (e.g., loop heads) should be looked at first, as they often correspond to natural program invariants and can thus more easily be compared with the expected results.

Automatic Refinement Several automatic techniques based on static analysis may help better understand an alarm raised by a static analysis. First, dependence analysis and slicing [109] compute automatically the part of a program that has an impact on a specific observation of program behaviors. In the case of an alarm, they can extract the set of program commands, the result of which may affect the occurrence or absence of an error, hence limiting the part of the program that should be inspected. Backward analysis techniques (section 5.5.3) can refine the information known about the executions that would produce an error at an alarm point, and can either lead to showing no error can happen at that point or help come up with a counter-example. The idea is to compute a tighter approximation of the executions that reach the alarm point and violate the property of interest. When this tighter approximation is empty, the alarm is proved false. Otherwise, it can still help better characterize a counter-example. Last, constraint solving attempts to identify the constraints over an execution that violates the property of interest at the alarm point and to identify sufficient error conditions. When a set of realizable error conditions is identified, we may be able to produce a counter-example.

Empiric Ranking and Logical Clustering Techniques Empiric techniques can also be used so as to *rank* alarms and help better cope with them. They do not need to perform any sound or formal proof search, which opens a wide range of possibilities. For instance, ranking based on kinds of alarms suggests addressing some class of issues first (e.g., that are judged more severe, or easier to fix at a cheaper cost). Also, machine learning techniques may help compute categories of alarms that would help users better decide where to look at first [66]. Last, empirical knowledge may show that a given analysis tool provides more accurate results on certain families of alarms than on others, which also suggests where to start. These empiric techniques can be combined with the manual approach discussed before. For example, when a clustering technique identifies similarities between alarms, then users may initially investigate the origin of only one alarm per cluster and assess whether the other alarms of a cluster have a similar cause. This makes triage simpler. Such techniques aim at reducing the amount of time users need to spend on the manual investigation of alarms. A particular case of logical clustering where alarms may be grouped together to speed up

investigation is when there exist dependences among them [75]. Let us consider an example analysis for array bound checking: when a possible out-of-bound access is detected in a loop with index variable *i*, it is likely that other accesses in the loop body and with the same index variable have the same issue, so that corresponding alarms are related.

Reparameterization and Reanalysis Analysis reparameterization may help improve the analysis in terms of performance or precision. When triage puts in evidence better parameter choices (e.g., by showing that certain loops could be analyzed more precisely with a higher number of unrolls), a new analysis is likely to improve the results overall.

However, this is not always the case; indeed, the frameworks of chapters 3 and 4 (theorem 4.3) do not assume that the abstract semantics is a monotone function (in fact, the widening used to analyze loops is generally not monotone in its second argument); thus, a local increase in precision (e.g., for the analysis of the body of a given loop) may actually not improve the precision overall, or even deteriorate it. Such effects may be quite nonintuitive. It is thus useful for users to be aware of the behavior of the static analyzer they rely on.

Similarly, some alternative analysis parameters may be identified that cut down the cost of the analysis. Again, a local performance improvement may unintuitively cause a worse total analysis time. In particular, when a new analysis uses less expressive abstract domains and analysis algorithms, a loss of precision may also take place and ultimately cause the analysis to be slower, as it may also need to explore more program branches (due to the decrease in precision).

6.4 Deployment of a Static Analysis Tool

To conclude this chapter, we discuss the deployment of a static analysis tool in the software development workflow.

In the following, we call *dispatch model* the way the analysis is integrated into the software development process. Depending on the application domain (e.g., safety-critical or not) and on the requirements (e.g., short or long software development cycle), there can be different

dispatch models. Each case has implications related to the kind of static analysis that can be used and to the usefulness of partial analysis results.

As it is not possible to account for all possible workflows, we present in the following paragraphs two specific categories of dispatch models, even though real cases most often fit somewhere in between.

Online Analysis Dispatch Model We call *online analysis dispatch model* a setup where programmers frequently perform static analysis runs as part of the development process. This means the software is reanalyzed whenever updates are made, compiled, committed into a revision system, or tested. The motivation for such a model is to catch certain families of bugs early. An extreme case of in-line analysis is the compiler-level analyses such as typing. Other analyses may be deployed this way to verify safety properties (absence of basic run-time errors) and security properties (absence of illegal information flows, with a rather simple flow model). In this model, developers should be able to take advantage by themselves of the analysis outputs. Thus, the parameterization of the analysis and the triage of alarms need to be lightweight. Moreover, incremental and modular analysis techniques are much more adapted to such a model, because of the speed and memory consumption constraints. Indeed, developers cannot afford to wait too long for analysis results, if they have to commit their updates quickly. Depending on the criticality of the software, the developers may be able to make great use of analysis tools that cover only a well-specified set of program behaviors, or to take advantage of partial triage of analysis alarms. Such an approach relies on the soundness under assumption principle described in theorem 6.1. For example, the INFER static analysis tool [14] is designed specifically for this dispatch model. As another example, the CLOUSOT static analysis tool [39] verifies code contracts over procedures in a modular way and explicitly assumes that distinct pointer parameters refer to memory regions that are not aliased. This means that the analysis does not cover all executions but still covers all executions satisfying this assumption, which allows for much faster analyses, while users who are aware of this behavior will understand the scope of the results.

Offline Analysis Dispatch Model We call *off-line analysis dispatch model* a setup where the analysis is performed as part of a validation phase that is separate from the software design and implementation stages. Then the software is not analyzed whenever it is modified but only after larger sets of modifications are made. The verification may be done each time a significant revision is issued. In the case of critical software, verification is carried out by a team that is not taking part in the actual programming (for the sake of separation of concerns, or to rely on more specialized teams for each phase). As a consequence, more costly analyses are feasible than in the case of an in-line analysis (the analysis will not slow down developers). This means that precise and global analyses can be successfully deployed that way and address more involved classes of semantic properties of interest (e.g., the computation of precise global invariants, or the verification of absence of run-time errors in complex software). Moreover, when a specialized team is responsible for running the analysis, tools that require a higher level of expertise (both for parameterization and for alarm triage) can be used. Last, when the application domain requires thorough software certification criteria to be met (as in avionics), soundness is critical. Thus, the assumptions on which the analysis relies should be well understood, and all alarms should be manually checked with help from some interactive triage tools.

Hybrid Dispatch Models The two cases discussed in the previous paragraphs are a bit extreme, and most software development processes lie somewhere in between. As an example, a possible model relies on frequent periodic analysis runs, with dedicated teams working closely with the developers to identify faults. Even though it is not possible to summarize all such models here, the principles presented in the previous paragraphs may serve as guidelines to optimize general verification processes.

7 Static Analysis Tool Implementation

Goal of This Chapter This chapter shows how to design and implement a static analysis tool and demonstrates the strong parallel between the concrete semantics, a concrete interpreter, the static analysis definition, and its implementation. We provide source code in OCaml for the implementation of two simple analyses that both target the basic programming language introduced in figure 3.1 and that respectively correspond to the compositional style presented in chapter 3 and to the transitional style presented in chapter 4. We use the OCaml language since it can express type definitions and operations over abstract syntax trees, and symbolic computations in a concise and intuitive manner.

We assume that the main elements of the theoretical foundations presented in chapters 3 and 4 are known (though it may be read in parallel, or before the most advanced points of these two chapters are fully mastered). In this chapter we focus on the implementation of a *non-relational* static analysis since, as we noted in chapter 3, relational abstract domains require more sophisticated data structures and algorithms and are outside the scope of this book. The code we provide is optimized not for efficiency but for size, so as to make it as easy to understand as possible (though we comment on efficiency later in this chapter).

Recommended Reading: [S], [D] This chapter is important to designers of static analysis tools. It should also be useful for readers and students who seek a better understanding of the internals of static analyzers. In fact, it may even be studied side by side with chapters 3 and 4, as we are considering the same language in this chapter. On the other hand, it may be skipped by readers who are less interested in the internals of a static analyzer, or by readers who are more interested in the use of a static analyzer (although a

good understanding of the internal design of static analysis tools is helpful when using one).

Chapter Outline We follow the methodology exposed in previous chapters, which defines the semantics first, then fixes the abstraction, and lastly derives the analysis from the semantics and the abstraction. In section 7.1 we define the representation of programs, recall the concrete semantics, and show how it relates to implementations of concrete interpreters. In section 7.2 we describe the data structures that represent the abstract values and the implementation of the basic operations on them. Section 7.3 focuses on the implementation of the static analysis of expressions and conditions. The last two sections describe the two kinds of analyses studied in depth previously: section 7.4 discusses the implementation of a static analyzer based on a compositional semantics (as shown in chapter 3), and section 7.5 describes the implementation of a static analyzer based on a transitional style semantics (as shown in chapter 4).

7.1 Concrete Semantics and Concrete Interpreter

The first step toward the design of the analysis is the definition of the syntax of programs and their semantics. We cover this phase in this section.

Syntax of Programs We use the same language as in chapter 3, and figure 7.1(a) recalls the grammar of figure 3.1.

In comparison with the example language of chapter 4 (section 4.4), this language lacks only the dynamic goto (**goto E**). For illustrative brevity we chose to use this minimal common syntax in illustrating implementation code for both compositional and transitional styles of static analyzers. Having studied the code, the readers will find it easy to extend the transitional-style code to cover other language constructs (dynamic gotos, function calls, etc.).

Figure 7.1(b) shows the definition of the corresponding data types in OCaml. This definition follows very closely the grammar of figure 7.1(a), except for commands, where data types also need to embed a notion of labels, so as to support the transitional-style analysis of chapter 4. Constants are denoted by type **const**, which is defined here as **int**, namely machine integers. Similarly, figure 7.1(b) lets variable names be designated by

consecutive non-negative integers (as in x_0, x_1, \dots), which will make code simpler later. Types `bop`, `rel`, `expr`, and `command` all boil down to sum types, defined by lists of constructors that describe all the ways to build a value of these types. For instance, `expr` stands for expressions, and an expression is either a constant `Ecst c` or a variable `Evar v`, or the application of a binary operator to a pair of expressions `Ebop (b, e0, e1)` (where `b` is the operator, and `e0` and `e1` are the sub-expressions). Conditions are defined by a product type that encloses the comparison operator, the variable, and the constant. Commands are described by two types. First, type `command` also boils down to a sum type, with six cases that correspond to the six basic kinds of commands of figure 7.1(a). Second, type `com` consists of pairs made of an integer (the type `label`) and an element of type `command`: the integer stands for the program point at which this command occurs, and the second element defines the command.

Representation of Memory States (or Memories) We now show that the concrete semantics is surprisingly close to an *interpreter*, namely, a program that inputs an abstract syntax tree and an initial memory state and performs the evaluation of the program starting from this memory state.

| | | | |
|--|---|---|--|
| $n \in \mathbb{V}$ $x \in \mathbb{X}$ $\odot ::= + - * \dots$ $\otimes ::= < \leq == \dots$ $E ::= \text{scalar expressions}$ $ n$ $ x$ $ E \odot E$ | $B ::= \text{Boolean expressions}$ $ x \otimes n$ | $C ::= \text{commands}$ $ \text{skip}$ $ C; C$ $ x := E$ $ \text{input}(x)$ $ \text{if}(B)\{C\} \text{else}\{C\}$ $ \text{while}(B)\{C\}$ | $\text{type label} = \text{int}$ $\text{type const} = \text{int}$ $\text{type var} = \text{int}$ $\text{type bop} = \text{Badd} \mid \text{Bsub} \mid \text{Bmul}$ $\text{type rel} = \text{Cineq} \mid \text{Csup}$ $\text{type expr} =$ $ \text{Ecst of const}$ $ \text{Evar of var}$ $ \text{Ebop of bop} * \text{expr} * \text{expr}$ $\text{type cond} =$ $ \text{rel} * \text{var} * \text{const}$ $\text{type command} =$ $ \text{Cskip}$ $ \text{Cseq of com} * \text{com}$ $ \text{Cassign of var} * \text{expr}$ $ \text{Cinput of var}$ $ \text{Cif of cond} * \text{com} * \text{com}$ $ \text{Cwhile of cond} * \text{com}$ $\text{and com} =$ $ \text{label} * \text{command}$ |
| (a) Syntax of programs | | (b) Abstract syntax tree of programs | |

Figure 7.1

Syntax of programs

We have provided a representation for program abstract syntax trees in figure 7.1(b), but we still need a representation for memory states. Recall that a memory was defined in chapter 3 as a function from variables to values:

$$M = X \rightarrow V$$

In figure 7.1(b), we let the representation of a variable be an integer value. Assuming that the variables of a program are always a contiguous sequence of integers starting from 0, we can simply let a memory be represented by

an array, where the cell of index i denotes the value of the variable i . The definition of the type `mem` is thus trivial and provided in figure 7.2. In the following, we note `read` and `write` for two functions to read and write into a variable:

- Function `read: var -> mem -> const` inputs a variable index and a memory, and returns the value of the variable.

```

type mem = const array
(* read: var -> mem -> const *)
let read x m = m.(x)
(* write: var -> const -> mem -> mem *)
let write x n m =
  let nm = Array.copy m in
  nm.(x) <- n;
  nm

type state = label * mem

```

Figure 7.2

Definition of memory states and of states

- Function `write: var -> const -> mem -> mem` inputs a variable index, the new value, and a memory and returns the memory obtained after the write.

Moreover, a program state is made of a label and a memory. Therefore, the corresponding type `state` is defined as a pair type shown in figure 7.2.

Semantics of Expressions Figure 7.3(a) recalls the semantics of expressions as defined in chapter 3, and figure 7.3(b) provides an implementation for this evaluation algorithm. For clarity, we show the type of the function `sem_expr` in the comment above its definition. This evaluation function inputs an expression and a memory and returns the value of the expression in that memory. It is defined recursively over the syntax of expressions, just like the semantics. It relies on a function `binop` to compute the result of the application of any binary operator to any pair of values, in the same way

as f_{\odot} (since this function is trivial, we omit it). We remark that this interpreter follows precisely the same structure as the semantics of expressions.

Following the same principles, figure 7.4 recalls the semantics of condition tests and presents its implementation. This interpreter simply applies a function `relop`, which computes the value of any comparison operation applied to any pair of values (since this function is trivial, we omit it). Again, the semantics of Boolean expressions is very close to the code of the interpreter for Boolean conditions shown in figure 7.4(a).

Compositional Semantics of Commands Figure 7.5(a) recalls the definition of the semantics of commands. This semantics takes a slightly different form compared to the semantics of expressions and conditions since it operates on *sets* of memory states. As a consequence, the equivalent interpreter differs from the semantics of commands since it simulates only one run of the program.

| | |
|--|---|
| $\llbracket E \rrbracket : M \longrightarrow V$ $\begin{aligned} \llbracket n \rrbracket(m) &= n \\ \llbracket x \rrbracket(m) &= m(x) \\ \llbracket E_0 \odot E_1 \rrbracket(m) &= f_{\odot}(\llbracket E_0 \rrbracket(m), \llbracket E_1 \rrbracket(m)) \end{aligned}$ <p style="text-align: center;">(a) Semantics of expressions</p> | <pre>(* sem_expr: expr * -> mem -> const *) let rec sem_expr e m = match e with Ecst n -> n Evar x -> read x m Ebop (o, e0, e1) -> binop o (sem_expr e0 m) (sem_expr e1 m)</pre> <p style="text-align: center;">(b) Equivalent interpreter</p> |
|--|---|

Figure 7.3

Evaluation of expressions

```

(* relop: rel -> const
 *      -> const -> bool *)
let relop c v0 v1 =
  match c with
  | Cineq -> v0 <= v1
  | Csup -> v0 > v1
(* sem_cond:
 *      (rel * var * const)
 *      -> mem -> bool *)
let sem_cond (c, x, n) m =
  relop c (read x m) n

[[B]] : M —→ B
[[x ⊕ n]](m) = f⊕(m(x), n)
(a) Semantics of tests
(b) Equivalent interpreter

```

Figure 7.4

Evaluation of Boolean expressions

Its definition is given in figure 7.5(b). If we accept the fact that it operates on a single element rather than on a set, it is mostly similar to the mathematical definition in figure 7.5(a). Even then, the cases of the skip commands, sequences of commands, assignments and input commands are very similar in both versions. Minor differences arise in the case of conditional commands, and loop commands. In the case of conditional commands, the semantics needs to compute the union of the output states of both branches, whereas the interpreter evaluates only one branch, according to the result of the condition. In the case of loop commands, we observe the same difference. The reason for this is that the interpreter computes the effect of *a single* execution, whereas the semantics accounts for all of them. It would be possible to design a concrete interpreter that computes all executions of a program and that follows the semantics in a very faithful manner, but it would not terminate whenever the program has at least one non-terminating execution. Despite this difference, the structure of the code of the interpreter for these two commands remains similar to that of the mathematical definition of their semantics. Therefore, the interpreter code

gives a strong intuition for the definition of the semantics, and vice versa, although they are not completely identical.

$$\begin{aligned}
 \llbracket C \rrbracket_{\mathcal{P}} : \wp(\mathbb{M}) &\longrightarrow \wp(\mathbb{M}) \\
 \llbracket \text{skip} \rrbracket_{\mathcal{P}}(M) &= M \\
 \llbracket C_0; C_1 \rrbracket_{\mathcal{P}}(M) &= \llbracket C_1 \rrbracket_{\mathcal{P}}(\llbracket C_0 \rrbracket_{\mathcal{P}}(M)) \\
 \llbracket x := E \rrbracket_{\mathcal{P}}(M) &= \{m[x \mapsto \llbracket E \rrbracket(m)] \mid m \in M\} \\
 \llbracket \text{input}(x) \rrbracket_{\mathcal{P}}(M) &= \{m[x \mapsto n] \mid m \in M, n \in \mathbb{V}\} \\
 \llbracket \text{if}(B)\{C_0\} \text{else}\{C_1\} \rrbracket_{\mathcal{P}}(M) &= \llbracket C_0 \rrbracket_{\mathcal{P}}(\mathcal{F}_B(M)) \cup \llbracket C_1 \rrbracket_{\mathcal{P}}(\mathcal{F}_{\neg B}(M)) \\
 \llbracket \text{while}(B)\{C\} \rrbracket_{\mathcal{P}}(M) &= \mathcal{F}_{\neg B} \left(\bigcup_{i \geq 0} (\llbracket C \rrbracket_{\mathcal{P}} \circ \mathcal{F}_B)^i(M) \right)
 \end{aligned}$$

(a) Semantics of commands

```

(* val sem_com: ccom -> mem -> mem *)
let rec sem_com (l, c) m =
  match c with
  | Cskip -> m
  | Cseq (c0, c1) -> sem_com c1 (sem_com c0 m)
  | Cassign (x, e) -> write x (sem_expr e m) m
  | Cinput x -> write x (input( )) m
  | Cif (b, c0, c1) ->
      if sem_cond b m then sem_com c0 m
      else sem_com c1 m
  | Cwhile (b, c) ->
      if sem_cond b m then
        sem_com (l, Cwhile (b, c)) (sem_com c m)
      else m
  
```

(b) An interpreter for commands

Figure 7.5

Evaluation of commands

```

(* step: prog -> state -> state *)
let step p (l, m) =
  match find p l with
  | Cskip ->
    (next p l, m)
  | Cassign (x, e) ->
    (next p l, write x (sem_expr e m) m)
  | Cinput x ->
    (next p l, write x (input( )) m)
  | Cif (b, c0, c1) ->
    if sem_cond b m then (fst c0, m)
    else (fst c1, m)
  | Cwhile (b, c) ->
    if sem_cond b m then (fst c, m)
    else (next p l, m)

```

Figure 7.6

Transition step

Transitional Semantics of Commands We can also provide an implementation for the transition relation used in chapter 4, which expresses how a program may step from one state to the next one. In the following, we let a program be represented by a data type `prog`. To navigate through the program, we assume the following two functions are defined:

- Function `next: prog -> label -> label` inputs a program and a label (program point); it then returns the successor label (e.g., as defined in section 4.4).
- Function `find: prog -> label -> com` inputs a program and a label; it then returns the command corresponding to this label.

The function `step` inputs a program `p` and a state `(l, m)` and first searches for the command located at label `l`; then it computes the next operation to be performed by the program depending on the form of this command. This function is presented in figure 7.6. In each case, we remark

that its definition follows the same structure as the compositional semantics of figure 7.5. For instance, in the case of an assignment command, it also evaluates the expression and carries out the write operation, and it then returns the resulting memory state together with the label corresponding to the next statement to execute. In the case of the input statement, this function assumes that a value is read from an oracle; thus, it computes a single execution. In this point of view, it is similar to the definition of figure 7.5.

```
type val_abs =
| Abot
| Atop
| Apos
| Aneg
type nr_abs = val_abs array
```

Figure 7.7

Non-relational abstraction machine representation

The evaluation of a call to this function corresponds to a single step of program execution (we remark that it does not perform any iteration in the case of a loop command). To compute a complete execution, one simply needs to call this function `step` repeatedly.

7.2 Abstract Domain Implementation

As noted in chapter 3, it is usually better to separate the design of the abstraction and that of the abstract interpreter itself. We follow the same approach in this chapter; thus, this section focuses on the implementation of the abstract domain, of the data structures to represent the abstract elements, and of the basic operations on them (bottom and top constants, abstract inclusion test, abstract union, widening, scalar operations, condition tests, etc.).

Data Types We first need to set the representation of the non-relational abstraction. In this paragraph, we assume that we use the value abstract

domain based on signs defined in example 3.6 (figure 3.5(b)). We discuss other non-relational abstract domains in the end of the section.

The types of abstract elements are shown in figure 7.7:

- The type `val_abs` describes the elements of the value abstract domain and consists of a sum type; it consists of four constructors corresponding to each of the abstract elements $\perp, \top, [\geq 0], [\leq 0]$.
- The type `nr_abs` describes the elements of the non-relational abstract domain, using arrays to represent functions (as such, it is very similar to the type `mem` used for the concrete interpreter; moreover, we assume it supports the same `read` and `write` operations to access or modify the abstract value associated to a variable).

Operations over the Value Abstract Domain Figure 7.8 displays the source code for the operations on the value abstract domain. These include

- the `val_incl` function, which decides whether the abstract ordering relation holds for a pair of abstract elements;
- the function `val_cst`, which implements the ϕ_V operation, which maps a constant to an abstract element that over-approximates it (in the case of the constants abstraction, it coincides with the best abstraction function),
- the function `val_sat`, which over-approximates the effect of condition tests,
- the function `val_ajoin`, which over-approximates concrete unions, and
- the function `val_binop`, which implements the computation of f_\odot^\sharp for each operator \odot of the language (it is not shown fully).

The function `val_sat` inputs refines an abstract value using on the condition expressed a comparison operator and a constant; for instance, `val_sat Csup 8 Atop` returns `Apos` because all numbers greater than 8 are positive. The structure of each of these functions is quite straightforward. The functions `val_binop`, `val_sat`, and `val_ajoin` are defined by case analysis, just like the mathematical functions that they implement. Moreover, the constant `val_top` is the top abstract value,

which approximates any set of scalars. Last, the constant `val_bot` defines the bottom element. We note that this lattice has finite height; thus, `val_join` can be used not only as an abstract union operator but also as a widening (as shown in section 3.3.3).

Operations over the Non-Relational Abstract Domain Figure 7.9 presents the implementation of the operations over the non-relational domain abstract elements. The function `nr_is_le` decides abstract ordering by checking that inclusion holds for each variable; thus, it boils down to an iteration over the abstract environment. The function `nr_join` computes an over-approximation for the union of sets of stores and also proceeds component per component. The function `nr_is_bot` checks whether an abstract element describes exactly the empty set of memory states. Last, `nr_bot` inputs an abstract element and returns an element describing the empty set of stores, over the same variables. We remark that each of these functions simply uses the operations over the value abstract domain.

Using Another Non-Relational Abstract Domain The abstract domain implementation provided in figures 7.7, 7.8, and 7.9 may look compact and trivial, but it corresponds to a very simple abstract domain. Realistic static analyses are likely to require much more expressive abstract domains. As observed in the previous paragraph, in the case of a non-relational domain based on a different value abstraction, we can simply reuse the code of figure 7.9 and substitute our implementation of the lattice of signs (including the type `val_abs` of figure 7.7 and the operations shown in figure 7.9) with the implementation of the new (and more realistic) value abstract domain. To illustrate this, we consider an alternate implementation of the value abstract domain, based on the constants abstraction. This value abstraction is based on the following abstract elements:

- \perp , which represents no value (as usual);
- abstract elements of the form $[n]_C$ for all scalars n —the element $[n]_C$ represents only the scalar value n ; and

```

let val_bot = Abot      (* val_bot: val_abs *)
let val_top = Atop       (* val_top: val_abs *)
(* val_incl: val_abs -> val_abs -> bool *)
let val_incl a0 a1 = a0 = Abot || a1 = Atop || a0 = a1
(* val_cst: const -> val_abs *)
let val_cst n = if n < 0 then Aneg else Apos
(* val_sat: rel -> int -> val_abs -> val_abs *)
let val_sat o n v =
  if v = Abot then Abot
  else if o = Cineq && n < 0 then
    if v = Apos then Abot else Aneg
  else if o = Csup && n >= 0 then
    if v = Aneg then Abot else Apos
  else v
(* val_join: val_abs -> val_abs -> val_abs *)
let val_join a0 a1 =
  match a0, a1 with
  | Abot, a | a, Abot -> a
  | Atop, _ | _, Atop | Apos, Aneg | Aneg, Apos -> Atop
  | Apos, Apos -> Apos
  | Aneg, Aneg -> Aneg
(* val_binop: bop -> val_abs -> val_abs -> val_abs *)
let val_binop o v0 v1 =
  match o, v0, v1 with
  | _, Abot, _ | _, _, Abot -> Abot
  | Badd, Apos, Apos -> Apos
  | Badd, Aneg, Aneg -> Aneg
  | Badd, _, _ -> Atop
  (* ... *)

```

Figure 7.8

Implementation of the operations on the value abstraction

```

(* nr_bot: nr_abs -> nr_abs *)
let nr_bot aenv = Array.map (fun _ -> Abot) aenv
(* nr_is_bot: nr_abs -> bool *)
let nr_is_bot aenv =
  Array.exists (fun a -> a = val_bot) aenv
(* nr_is_le: nr_abs -> nr_abs -> bool *)
let nr_is_le aenv0 aenv1 =
  let r = ref true in
  Array.iteri
    (fun x a0 -> r := !r && val_incl a0 (read x aenv1))
    aenv0;
  !r
(* nr_join: nr_abs -> nr_abs -> nr_abs *)
let nr_join aenv0 aenv1 =
  Array.map
    (fun x a0 -> val_join a0 (read x aenv1))
    aenv0

```

Figure 7.9

Implementation of the operations on the non-relational abstraction

- \top , which represents any set of values.

The implementation of this value abstract domain is shown in figure 7.10. The data type for the representation of the abstract elements boils down to a simple sum type, with three constructors, corresponding, respectively, to \perp , to elements of the form $[n]_C$, and to \top . The operations follow from this definition, and their code is quite straightforward as well.

The implementation of a more sophisticated non-relational domain (e.g., intervals) would follow the same principles. Obviously, a separate widening operator is required when the value domain has infinite chains. The use of a relational abstract domain would require more radical changes since type `nr_abs` would not be reused, and all the functions of figure 7.9 would need to be replaced with code implementing entirely different algorithms. We do not discuss this topic in this book and refer the reader to the APRON

library [63] for the presentation of the interface of a relational abstract domain.

7.3 Static Analysis of Expressions and Conditions

We now assume that the implementation of a scalar abstract domain is available and follows the structure shown in section 7.2. In this section, we implement static analysis functions for scalar expressions and for Boolean conditions. Indeed, these functions will be useful both for the analysis based on a compositional semantics (chapter 3) and for the analysis based on a transitional semantics (chapter 4).

```

type val_cst = Abot | Acst of int | Atop
let val_bot = Abot
let val_top = Atop
let val_incl a0 a1 = a0 = Abot || a1 = Atop || a0 = a1
let val_cst n = Acst n
let val_sat o n v =
  match o, v with
  | _, Abot -> Abot
  | Cineq, Acst i -> if i > n then Abot else v
  | Csup, Acst i -> if i <= n then Abot else v
  | _, _ -> v
let val_join a0 a1 =
  match a0, a1 with
  | Abot, a | a, Abot -> a
  | Atop, _ | _, Atop -> Atop
  | Acst _, Acst _ -> if a0 = a1 then a0 else Atop
let val_binop o v0 v1 =
  match o, v0, v1 with
  | _, Abot, _ | _, _, Abot -> Abot
  | Badd, Atop, _ | Badd, _, Atop -> Atop
  | Badd, Acst i0, Acst i1 -> Acst (i0 + i1)
  | ...

```

Figure 7.10

Alternate implementation of a value abstraction: constants

Analysis of Scalar Expressions Figure 7.11 shows the analysis of scalar expressions. Figure 7.11(a) recalls the definition given in section 3.3, and figure 7.11(b) shows the corresponding implementation. These functions compute an over-approximation for the result of an expression, when evaluated in a memory described by a given abstract pre-condition. We observe that the forms of both definitions are very similar and follow the inductive structure of scalar expressions, just like the concrete semantics of expressions (figure 7.3):

- In the case of a constant, the approximation is provided by the `val_cst` function.
- In the case of a variable, the approximation of its value is read in the non-relational abstract state thanks to function `read`.

$$\begin{aligned} \llbracket E \rrbracket^\sharp : \mathbb{A} &\longrightarrow \mathbb{A}_\gamma \\ \llbracket n \rrbracket^\sharp(M^\sharp) &= \phi_\gamma(n) \\ \llbracket x \rrbracket^\sharp(M^\sharp) &= M^\sharp(x) \\ \llbracket E_0 \odot E_1 \rrbracket^\sharp(M^\sharp) &= f_\odot^\sharp(\llbracket E_0 \rrbracket^\sharp(M^\sharp), \llbracket E_1 \rrbracket^\sharp(M^\sharp)) \end{aligned}$$

(a) Abstract semantics of scalar expressions

```
(* ai_expr: expr -> nr_abs -> val_abs *)
let rec ai_expr e aenv =
  match e with
  | Ecst n -> val_cst n
  | Evar x -> read x aenv
  | Ebop (o, e0, e1) ->
    val_binop o (ai_expr e0 aenv) (ai_expr e1 aenv)
```

(b) Code of the abstract interpreter for scalar expressions

Figure 7.11

Static analysis of scalar expressions

- The effect of binary operators is over-approximated by the function `val_binop`.

We remark that each case in this function simply uses an operation defined as part of the value abstract domain, in section 7.2. This explains why the code in figure 7.11(b) is so concise and simple. Using another non-relational domain instead of the sign abstract domain would not require the function `ai_expr` to be modified. On the other hand, and as we remarked in section 3.3, the use of a relational abstract domain would require a radical structure change, and more sophisticated algorithms.

Analysis of Condition Tests As remarked in sections 3.3 and 4.4.4, the analysis of a condition test \mathbf{B} consists of the abstract interpretation of a function $\mathcal{F}_{\mathbf{B}}$ or $\text{filter}_{\mathbf{B}}$ that filters out states depending on whether they satisfy \mathbf{B} or not. As noted in section 5.5.1, it can be viewed as a backward analysis of the standard semantics of Boolean expressions (recalled in figure 7.4) that enriches abstract states with information derived from the condition. Therefore, its structure is different from that of `ai_expr`. The function `ai_cond` shown in figure 7.12 applies the function `val_sat` defined in section 7.2 to compute a sound over-approximation of the values that satisfy the condition \mathbf{B} . It returns either the abstract state \perp when the condition is not feasible, or the refined abstract state otherwise.

```
(* ai_cond: cond -> nr_abs -> nr_abs *)
let ai_cond (r, x, n) aenv =
  let av = val_sat r n (read x aenv) in
  if av = val_bot then nr_bot aenv
  else write x av aenv
```

Figure 7.12

Code of the static analysis of condition tests

7.4 Static Analysis Based on a Compositional Semantics

At this point, we have implemented the operations related to the abstract domain, and functions to analyze expressions and conditions. Based on this, we now construct a static analyzer that inputs a program and an abstract pre-condition (which is usually defined as \top at the beginning of a complete program, meaning that we assume no information is available at that point) and computes an abstract post-condition that over-approximates all program behaviors. This analysis implements the definitions of section 3.3.

Analysis of Commands The abstract semantics of a command consists of a function that inputs an abstract pre-condition and returns an over-approximated abstract post-condition. In section 3.3, it was defined by

induction over the syntax. We recall it in figure 7.13(a). The direct implementation of the function `ai_com` is shown in figure 7.13(b), and also proceeds by induction over the syntax. Let us discuss more specifically the case of a few kinds of commands.

When the input abstract element denotes the empty set of memory states, the analysis immediately returns the bottom element as well.

The analysis of a sequence boils down to the composition of the analysis of each of the sub-commands, in the same way as the concrete semantics composes the semantics of each sub-command.

Just like in the concrete semantics, the analysis of an assignment command evaluates the expression and writes the result in the position corresponding to the variable. The only difference with the concrete semantics is that the evaluation of the expression takes place in the value abstract domain (using `ai_expr`) instead of the set of machine integers.

The analysis of a condition command consists of three phases: it first performs the analysis of the condition test (for both the true and false branches, which, respectively, correspond to the Boolean expression and to its negation); second, it performs the analysis of both branches; and last, it applies the abstract join operator `ai_join` (defined in section 7.2).

$$\begin{aligned}
\llbracket \text{skip} \rrbracket_{\mathcal{P}}^{\sharp}(M^{\sharp}) &= M^{\sharp} \\
\llbracket C_0; C_1 \rrbracket_{\mathcal{P}}^{\sharp}(M^{\sharp}) &= \llbracket C_1 \rrbracket_{\mathcal{P}}^{\sharp}(\llbracket C_0 \rrbracket_{\mathcal{P}}^{\sharp}(M^{\sharp})) \\
\llbracket x := E \rrbracket_{\mathcal{P}}^{\sharp}(M^{\sharp}) &= M^{\sharp}[x \mapsto \llbracket E \rrbracket^{\sharp}(M^{\sharp})] \\
\llbracket \text{input}(x) \rrbracket_{\mathcal{P}}^{\sharp}(M^{\sharp}) &= M^{\sharp}[x \mapsto T_Y] \\
\llbracket \text{if}(B)\{C_0\}\text{else}\{C_1\} \rrbracket_{\mathcal{P}}^{\sharp}(M^{\sharp}) &= \llbracket C_0 \rrbracket_{\mathcal{P}}^{\sharp}(\mathcal{F}_B^{\sharp}(M^{\sharp})) \sqcup^{\sharp} \llbracket C_1 \rrbracket_{\mathcal{P}}^{\sharp}(\mathcal{F}_{\neg B}^{\sharp}(M^{\sharp})) \\
\llbracket \text{while}(B)\{C\} \rrbracket_{\mathcal{P}}^{\sharp}(M^{\sharp}) &= \mathcal{F}_{\neg B}^{\sharp}(\text{abs_iter}(\llbracket C \rrbracket_{\mathcal{P}}^{\sharp} \circ \mathcal{F}_B^{\sharp}, M^{\sharp}))
\end{aligned}$$

(a) Abstract semantics of commands

```

(* postlfp: (nr_abs -> nr_abs) -> nr_abs -> nr_abs *)
let rec postlfp f a =
  let anext = f a in
  if nr_is_le anext a then a
  else postlfp f (nr_join a anext)
(* ai_com: com -> nr_abs -> nr_abs *)
let rec ai_com (l, c) aenv =
  if nr_is_bot aenv then aenv
  else
    match c with
    | Cskip -> aenv
    | Cseq (c0, c1) -> ai_com c1 (ai_com c0 aenv)
    | Cassign (x, e) -> write x (ai_expr e aenv) aenv
    | Cinput x -> write x val_top aenv
    | Cif (b, c0, c1) ->
        nr_join
        (ai_com c0 (ai_cond b aenv))
        (ai_com c1 (ai_cond (cneg b) aenv))
    | Cwhile (b, c) ->
        let f_loop = fun a -> ai_com c (ai_cond b a) in
        ai_cond (cneg b) (postlfp f_loop aenv)

```

(b) Code of the abstract interpreter for commands

Figure 7.13

Abstract interpretation of programs

The case of loop commands is more complex and interesting. The analysis of a loop requires the computation of a sequence of abstract iterates, until it converges (section 3.3.3). To implement this, we define the higher-order function `postlfp`: it inputs a function `f` and an abstract element `a` and checks if $(f\ a)$ is smaller than `a` according to the abstract ordering; if so, it terminates; otherwise, it computes an abstract join and calls itself again so as to compute one more iterate. The analysis of the loop then simply uses this function: it calls it with the function `f_loop` as an argument, which describes the analysis of one iteration of the loop, including the analysis of the condition test (the condition should evaluate to `true` to stay in the loop) and the analysis of the body of the loop. When the iteration returns the loop head invariant, it applies the analysis of the condition test again for the loop exit (the condition should then evaluate to `false`).

Note that we assumed a lattice of finite height, so that any sequence of increasing iterates eventually stabilizes. Thus, abstract iteration can use `ai_join`; if we remove this assumption, we would need to use a function `ai_widen` that implements widening instead of `ai_join`.

This concludes the implementation of the design of the analysis. To analyze a program, we simply need to call function `ai_com`, with the program to analyze and the abstract pre-condition as arguments. Usually, we make no assumption on the initial state and let the abstract pre-condition be the top element.

As remarked in section 3.3, this static analysis function can easily be instrumented to also accumulate an abstraction of the reachable states at any program point. This modification requires a global table that maps labels to local abstract states and supports two functions:

- `storage_find: label -> nr_abs` (retrieval of the—possibly bottom—abstract state recorded at a given label)
- `storage_add: label -> nr_abs -> unit` (update of the abstract state at a given label)

Then the accumulation of the reachable abstract states at any program point boils down to a single line addition in the beginning of function `ai_com` so as to join its argument `aenv` with the element previously associated to `l` in this global map:

```

let rec ai_com (l, c) aenv =
  storage_add l (nr_join (storage_find l) aenv);
  if nr_is_bot aenv then aenv
  else
    match c with
    (* ... as in the previous implementation ... *)

```

Optimizing the Representation of Abstract States So far, we have deliberately opted for a representation that leads to simpler yet less efficient code. The choice of efficient data structures is crucial to achieve efficient static analysis algorithms.

First, the abstract states could be represented as a coalescent product (example 5.5). So far, we let the *bottom* abstract state be represented as an array of `val_bot` elements, which makes testing for bottom inefficient. By contrast, the coalescent product representation suggests using a sum type to describe `nr_abs`, with one case denoting the bottom abstract state and another case for non-bottom abstract states.

Second, the use of an array structure for the non-bottom abstract states leads to inefficient copies in `write`. A real-world static analyzer would use more efficient data structures, such as persistent structures, that alleviate the need for such copies.

Improving the Analysis In chapter 5, we showed several advanced static analysis techniques. Many of these can be applied to the static analyzer that we have shown in this section, without adding much complexity to the code.

We have already discussed in section 7.2 and in the previous paragraphs how another abstract domain could be used instead of the basic signs abstraction used above:

- The use of another non-relational domain would require local changes to the value abstraction, as shown in figure 7.10.
- The use of a relational domain would require more global changes affecting `ai_expr` and `ai_cond`.
- The use of an abstract domain that has infinite chains would necessitate the use of a widening operator `ai_widen` instead of abstract union in function `postlfp`.

Section 5.2 presented several loop analysis techniques, such as loop unrolling, delayed widening, and decreasing iterations. They can all be implemented inside the functions `postlfp` and `ai_com`. As an example, the following `postlfp` function will unroll all loops twice:

```
let postlfp f a =
  let rec aux acc n a =
    let anext = f a in
    if n < 2 then
      aux (nr_join a anext) (n + 1) anext
    else if nr_is_le anext a then nr_join acc a
    else aux acc (n + 1) (nr_join a anext) in
  aux a 0 a
```

This function uses an accumulator to join separately unrolled iterations and the non-unrolled iterations, but except for this, it behaves entirely like the initial version of `postlfp`.

7.5 Static Analysis Based on a Transitional Semantics

We now provide an implementation for a second version of the analysis, which relies on the transitional framework of chapter 4.

Abstract Interpretation of an Execution Step First, we discuss the implementation of the function that computes the effect of all possible one-step transitions from a control point and abstract state. Essentially, this function over-approximates the effect of the transition relation that is used as a basis to design the analysis presented in chapter 4. The function `ai_step` is shown in figure 7.14. It inputs three arguments:

- The pair (l, c) designates a program point (label) and the associated command.
- The label l_{next} designates the next program point that follows the command c ; this is useful whenever the execution of a single step from the control state l may complete the execution of the command c and branch to the next command.
- The abstract element $aenv$ designates the current abstract memory at program point l .

It outputs a list of pairs made of a program point and an element of the non-relational abstract memory; each pair in that list denotes a program point after one transition step from $(l, aenv)$ and an over-approximation of the memories after that transition. This function is sound in the sense that it accounts for all possible one-step transitions from $(l, aenv)$.

The function `ai_step` performs a case analysis on the statement `C`. Most cases can actually be directly related to a case in the definition of the function `ai_com` presented in section 7.4:

- Skip commands, assignment commands, and input commands boil down to a one-step transition and are thus analyzed exactly in the same way as in section 7.4.
- A one-step transition at the beginning of a conditional command boils down to the evaluation of the test and the subsequent branching; thus, the analysis simply applies the condition test analysis function `ai_cond` on each of the two branches and returns the pairs made of the first label of each branch, with the corresponding new abstract state.
- The case of a loop statement is very similar to that of a conditional command; the only difference is that it branches to the loop head in the true case and to the next statement in the false case.

The function `ai_step` uses all the basic functions that were used by `ai_com` defined in section 7.4. Indeed, the analysis of assignment statements relies on the same function `ai_expr` for the analysis of expressions, and the analysis of conditions is done by the same function `ai_cond`. Therefore, both analyses share most of the code.

Global Iteration Function To compute the effect of arbitrarily long sequences of steps in the abstract, we need to iterate the function `ai_step`. In chapter 4 this was done using a worklist algorithm. Therefore, we now provide an implementation for this algorithm in figure 7.15. This function proceeds as follows:

```

(* ai_step: com -> label -> nr_abs
 *           -> (label * nr_abs) list *)
let rec ai_step (l, c) lnext aenv =
  match c with
  | Cskip ->
    [ (lnext, aenv) ]
  | Cseq (c0, c1) ->
    ai_step c0 (fst c1) aenv
  | Cassign (x, e) ->
    [ (lnext, write x (ai_expr e aenv) aenv) ]
  | Cinput x ->
    [ (lnext, write x val_top aenv) ]
  | Cif (b, c0, c1) ->
    [ (fst c0, ai_cond b aenv);
      (fst c1, ai_cond (cneg b) aenv) ]
  | Cwhile (b, c) ->
    [ (fst c, ai_cond b aenv);
      (lnext, ai_cond (cneg b) aenv) ]

```

Figure 7.14

Abstraction of the transition relation

- It initializes the global table `global_tbl` that is meant to store the abstract states associated to each program point (label) of the program being analyzed. In the beginning this structure only stores an invariant at the entry label `l`; it is subsequently updated whenever a program point is visited by the analysis.
- It initializes the worklist `wlist` with the entry program label; this structure is also updated whenever the analysis visits a new control state for which no stable invariant has been computed yet.
- Then it picks a program label `l` from the worklist and carries out the analysis of all the transitions from `l`:
 - It retrieves the corresponding command `c` and the current abstract memory `aenv` at `l`.

- It computes all the possible transitions from this point, using the function `ai_step` presented in the previous paragraph.
- For each element of the list (`lnext, apost`), it checks whether the new abstract memory is included in the previously known abstract memory at this label. If so, `lnext` does not need to be put back into the worklist; otherwise, it adds `lnext` to the worklist and updates the abstract memory at this program label.

```
(* ai_iter: prog -> nr_abs -> unit *)
let ai_iter p aenv =
  let (l, c) = first p in
  invs := I.add l aenv I.empty;
  let wlist = T.create () in
  T.add l wlist;
  while not (T.is_empty wlist) do
    let l = T.pop wlist in
    let c = find p l in
    let lnext = next p l in
    let aenv = I.find l !invs in
    let aposts = ai_step (l, c) lnext aenv in
    List.iter
      (fun (lnext, apost) ->
        let old_apost = storage_find lnext in
        if not (nr_is_le apost old_apost) then
          let new_apost = nr_join old_apost apost in
          invs := I.add lnext new_apost !invs;
          T.add lnext wlist
        ) aposts
  done
```

Figure 7.15

Abstract interpretation of transitional semantics: global iterator

When the function `ai_iter` terminates, the worklist is empty, which means that the analysis stabilized at all labels (program points) and thus

`global_tbl` stores a sound over-approximation of all the memory states that may reach each label (program point) in the program.

Note that the implementation in figure 7.15 has some gap from the algorithm in figure 4.5, yet the equivalence of the two is easy to observe. The difference on surface comes from the fact that the algorithm in figure 4.5 is macroscopic, while the implementation in figure 7.15 is microscopic. One iteration in figure 4.5 visits all the program labels in the worklist, and a new worklist of labels to visit next is constructed for the next iteration. On the other hand, one iteration in figure 7.15 picks and visits one label from the worklist, and new labels to visit next are added to the worklist. The $F^\#$ part in figure 4.5 is macroscopic too, making one-step transitions for all the labels in the worklist, partitioning the results by the labels, and joining all at each label into a single abstract memory. Implementation in figure 7.15 has the same effect as $F^\#$ except that it is microscopic, making one-step transition from a label and joining the memories produced by the one-step transition with the existing ones at the next labels.

As we have remarked above, the implementations of the static analysis derived from the compositional semantics and from the transitional semantics share most of their code. The main difference is the way they deal with iteration. In the case of the former, the iteration is local to the analysis of loops, whereas the analysis of other statements boils down to the composition of functions; thus, the function `ai_com` calls `postlfp` locally when it needs to analyze a loop. In the case of the latter, the iteration is global, and the main analysis function is the worklist iteration function `ai_iter`, which calls `ai_step` for the analysis of each transition.

Improvements of the Analysis As in section 7.4, the implementation of the analysis presented in figures 7.14 and 7.15 can be improved in many ways.

Since all the code related to the abstract domain is shared with the analysis of section 7.4, any change to a more expressive abstract domain would be done the same way. In case of a switch to a domain with infinite chains, a widening operator needs to be used instead of `nr_ajoin` in the iteration function `ai_iter`.

Optimizations related to the iteration strategy should be done inside the `ai_iter` function. For instance, techniques such as unrolling (section

5.2.1) should be integrated there. Similarly, when using a widening operator and applying it only at specific points, the selection of widening points should be carried out in this function too.

8 Static Analysis for Advanced Programming Features

Goal of This Chapter This chapter discusses the analysis of more realistic programming languages than those considered so far. Indeed, previous chapters focus on the foundations of static analysis, and hence opt for a somewhat contrived language. We now aim at considering full-featured languages.

Real-world programming languages feature arrays, pointers, dynamic memory allocation, functions, and other constructions than the basic arithmetic and Boolean operators considered so far. This chapter provides a high-level view of the static analysis techniques for such programming language features. As the range of these features is quasi-infinite (e.g., exceptions, multi-staged meta programming, parameterized modules, and dynamic dispatches), and the academic literature on their analysis is also very wide and thorough, we do not aim for an exhaustive coverage. Instead, we consider a few common and representative constructions and present the most significant techniques that cope with it. For each construction, we summarize the concrete semantics at a high level, discuss the salient properties related to static analysis, and present the main abstractions. We do not provide in-depth discussion of the analysis algorithms (although we give the underlying intuitions), as it would go beyond the scope of this introductory book (instead, we provide references). We stress the link between the properties of interest and the abstractions that are required to cope with them. Note that our approach benefits from the advantages of the abstract interpretation methodology as presented in previous chapters; indeed, we rely on the facts that the concrete semantics serves as a basis for the analysis design and that the analysis algorithms derive from the abstraction, so that the choices of the semantics and of the abstraction are key.

Recommended Reading: [S] (Master), [D], [U] (2nd reading, depending on need). Engineers and students can skip this chapter in a first reading and revisit it when dealing with the analysis of software relying on more advanced

programming features. Teachers will find here examples that can be used to illustrate in classes and practical sessions.

Chapter Outline Sections 8.1 and 8.2 first discuss the step-by-step design of abstract interpreters in the framework of chapter 4 for two common dynamic programming features: pointers and functions. While doing so, we demonstrate the convenience of the semantics-based approach to static analysis. Indeed, the design of an analysis is driven by the concrete semantics and the abstraction. Therefore, the presence of pointers and procedures does not require fundamentally different frameworks to formalize, and prove the analysis correct, even though novel abstractions need to be used.

| | |
|-----------------|--------------------------------------|
| $E ::= \dots$ | expression, as before (figure 4.6) |
| $\&x$ | location of a variable |
| malloc | location of a newly allocated memory |
| $*E$ | dereference of a memory location |
| $C ::= \dots$ | statement, as before (figure 4.6) |
| $*E := E$ | indirect assignment |
| $P ::= C$ | program |

Figure 8.1

Syntax of an example language with pointers and dynamic memory allocations

Subsequently, sections 8.3 and 8.4 focus more on the abstractions required to cope with complex programming languages. These two sections describe a few common programming languages features and corresponding basic abstractions. Features are treated separately, and abstractions are mostly presented at a high level, for the sake of concision. Section 8.3 focuses on features related to memory states, whereas section 8.4 studies features related to control states.

8.1 For a Language with Pointers and Dynamic Memory Allocations

8.1.1 Language and Concrete Semantics

Let us consider an imperative language that admits memory locations as values. The syntax of this example language is shown in figure 8.1. We extend

the imperative language of section 4.4 (figure 4.6) with the constructs that generate and use memory locations.

A **malloc** expression allocates an isolated fresh memory and returns its address. We let the allocated size be always a unit size that can store integers, labels, or addresses. (A more realistic construct for the dynamic memory allocation is considered in section 8.3.4.) Dereference expression $*E$ computes a location value from E and then returns the value stored at this location. Indirect assignment $*E_1 := E_2$ computes a location from E_1 and stores the value of E_2 into the location.

Example 8.1 (Language semantics) Consider the following program:

```
x := malloc;
y := &x;
*x := 5;
*y := *x
```

After the first statement, “ $x := \text{malloc}$ ”, the memory is $\{x \mapsto a\}$, where the a is the address of a fresh memory location. After the second statement, “ $y := &x$ ”, the memory becomes $\{x \mapsto a, y \mapsto x\}$. Then the assignment “ $*x := 5$ ” assigns integer 5 to the address stored in x ; hence, the memory becomes $\{x \mapsto a, y \mapsto x, a \mapsto 5\}$. Then the last assignment “ $*y := *x$ ” dereferences the address (a) stored in x , gets 5, and stores it to the address (x) stored in y . Hence, the memory in the end is

$$\{x \mapsto 5, y \mapsto x, a \mapsto 5\}.$$

This concrete semantics is formalized in the following.

Concrete Transitional Semantics Given a program, its initial states I , and the set L of the labels for its statements, the transitional-style semantics (chapter 4) is the least fixpoint of the following semantic function F :

$$\begin{aligned} S &= L \times M \\ F : \wp(S) &\rightarrow \wp(S) \\ F(X) &= I \cup Step(X), \end{aligned}$$

where

$$Step = \check{\wp}(\rightarrow).$$

Before we define the transition relation $\hookrightarrow \subseteq S \times S$ between concrete states, we have to define the semantic domains. The memories are as before, maps from locations to values, yet the location (address) domain A is extended to include dynamically allocated memory addresses as well as program variables. Since such locations can also be values of expressions, the value domain V

contains such locations as well as integers (for integer expressions) and labels (for goto-target expressions):

$$\begin{aligned}
 M &= A \rightarrow V && \text{memories} \\
 A &= X \cup H && \text{addresses (locations)} \\
 V &= Z \cup L \cup A && \text{values} \\
 X & && \text{set of variables} \\
 H & && \text{set of allocated heap addresses} \\
 L & && \text{set of statement labels}
 \end{aligned}$$

The H set denotes the locations generated by the **malloc** expressions. Given a program, every **malloc** expression is assumed to have a unique number μ , writing malloc_μ . Let the set of these malloc-site numbers be N_{site} . Then we can model the domain of memory addresses as

$$H = N_{site} \times N.$$

Thus, the addresses of fresh locations from a malloc_μ are $(\mu, 0), (\mu, 1), \dots$

Now we define the transition relation \hookrightarrow for the indirect assignment, which is the only new statement added to the imperative language of section 4.4 (figure 4.6). If label l is an indirect assignment statement, then its transition is

$$*E_1 := E_2 : (l, m) \hookrightarrow (\text{next}(l), \text{update}(m, \text{eval}_{E_1}(m), \text{eval}_{E_2}(m))),$$

where the $\text{update}(m, v_1, v_2)$ returns the same memory as m except that location v_1 has value v_2 . For example, for assignment statement “ $*x := 3$ ” the evaluation of x will return a location, a stored value in x , into which integer 3 is stored.

The $\text{eval}_E(m)$ computes the value of expression E given memory m . For expressions, there are new constructs that generate or dereference locations. The evaluation function eval for these three pointer expressions, together with the reminder of the variable expression case, are as follows:

$$\begin{aligned}
 \text{eval}_x(m) &= \text{fetch}(m, x) && \text{stored value in a variable } x \\
 \text{eval}_{\&x}(m) &= x && \text{variable as a location } \&x \\
 \text{eval}_{\text{malloc}_\mu}(m) &= (\mu, z) && \text{new number } z \text{ for malloc site } \mu, \text{ as a fresh location} \\
 \text{eval}_{*E}(m) &= \text{fetch}(m, \text{eval}_E(m)) && \text{dereference of a location,}
 \end{aligned}$$

where $\text{fetch}(m, v)$ returns the value of m at location v .

In summary, the types of the above semantic operations are

$$\begin{aligned}
\text{eval}_E &: M \rightarrow V \\
\text{update} &: M \times V \times V \rightarrow M \\
\text{fetch} &: M \times V \rightarrow V.
\end{aligned}$$

Though the programs in the above language may easily have two expressions point to the same location (*alias*), when we design a static analysis we do not need a separate concern about how to handle the alias behavior. This is because all behaviors of programs are defined within the semantics, and the alias behavior is one of the phenomena that appear from the semantics. Thus, in designing a static analysis it is sufficient to focus on the semantics. This is the power and convenience of semantics-based static analysis framework (chapters 3 and 4).

Now, designing a sound static analysis boils down to designing a sound abstract semantics.

8.1.2 An Abstract Semantics

Following the framework of chapter 4, we define an abstract semantics as follows:

$$\begin{aligned}
\text{abstract domain} \quad S^\# &= L \rightarrow M^\# \\
\text{abstract semantic function} \quad F^\# : S^\# &\rightarrow S^\# \\
F^\#(X^\#) &= \alpha(I) \cup^\# Step^\#(X^\#) \\
Step^\# &= \wp(\text{id}, \cup_M^\#) \circ \pi \circ \wp(\rightarrow^\#),
\end{aligned}$$

where $\rightarrow^\#$ is the one-step abstract transition relation $\subseteq S^\# \times S^\#$.

At this level, all things remain the same as described in chapter 4. The function π partitions a subset of $L \times M^\#$ by the labels, returning an element in $L \rightarrow \wp(M^\#)$. The abstract state domain $S^\#$ and the abstract transition relation $\rightarrow^\#$ need to satisfy the framework's required conditions:

- We design the set $L \rightarrow M^\#$ of abstract states as a CPO that is Galois connected with $\wp(L \times M)$:

$$\wp(L \times M) \xrightleftharpoons[\alpha]{\gamma} L \rightarrow M^\#$$

- We design the abstract memory $M^\#$ as a CPO that is Galois connected with $\wp(M)$:

$$\wp(M) \xrightleftharpoons[\alpha_M]{\gamma_M} M^\#$$

- We need to check that the abstract transition relation $\hookrightarrow^\#$ as a function satisfies the soundness condition:

$$\check{\wp}(\hookrightarrow) \circ \gamma \subseteq \check{\wp}(\gamma) \circ \hookrightarrow^\#$$

- We need to check that the abstract union operators $\cup^\#$ and $\cup_M^\#$ satisfy the soundness condition:

$$\cup \circ (\gamma_-, \gamma_-) \subseteq \gamma_- \circ \cup_-^\#$$

Then, for any input program, the algorithms of section 4.3 soundly approximate the concrete semantics of the program.

Abstract Domains The abstract domains $S^\#$, $M^\#$, $V^\#$, and $A^\#$ are all CPOs that are Galois connected with the corresponding concrete domains:

$$\wp(\mathbb{L} \times M) \xrightleftharpoons[\alpha]{\gamma} \mathbb{L} \rightarrow M^\# \quad \wp(M) \xrightleftharpoons[\alpha_M]{\gamma_M} M^\#,$$

where

$$\begin{aligned} \alpha(S) &= \{l \mapsto \alpha_M(\{m \mid (l, m) \in S\}) \mid l \in \mathbb{L}\}, \\ \gamma(s^\#) &= \{(l, m) \mid m \in \gamma_M(s^\#(l)), l \in \mathbb{L}\}. \end{aligned}$$

Now, the abstract domain $M^\#$ is the parameter.

Example 8.2 (A memory abstract domain) An example of such a domain $M^\#$ is

$$M^\# = (X \cup N_{site}) \rightarrow V^\#,$$

given that $V^\#$ is a Galois connected CPO. The X and N_{site} are finite sets of, respectively, the variables and the allocation sites in the input program.

The abstraction and concretization functions are as follows:

$$\begin{aligned} \alpha_M(M)(x) &= \alpha_V(\{m(x) \mid m \in M\}) && \text{if } x \in X \\ \alpha_M(M)(\mu) &= \alpha_V(\{m(\mu, n) \mid m \in M, n \in \mathbb{N}\}) && \text{if } \mu \in N_{site} \\ \gamma_M(M^\#) &= \{m \in M \mid \forall x \in X : m(x) \in \gamma(M^\#(x)), \\ &\quad \forall \mu \in N_{site}, \forall n \in \mathbb{N} : m(\mu, n) \in \gamma(M^\#(\mu))\} \end{aligned}$$

Example 8.3 (An abstract value domain) An example abstract value domain $V^\#$, a CPO that is Galois connected,

$$\wp(V) \xrightleftharpoons[\alpha_V]{\gamma} V^\#,$$

is achieved by abstracting kind-wise a set of values (integers, labels, and/or locations). We abstract its set of integers into an abstract integer, its set of labels into an abstract label, and a set of locations into an abstract location:

$$\wp(\mathbb{Z} \cup \mathbb{L} \cup \mathbb{A}) \xrightleftharpoons[\alpha_V]{\gamma_V} \mathbb{Z}^\sharp \times \mathbb{L}^\sharp \times \mathbb{A}^\sharp,$$

where

$$\begin{aligned}\alpha_V(V) &= (\alpha_Z(V \cap \mathbb{Z}), \alpha_L(V \cap \mathbb{L}), \alpha_A(V \cap \mathbb{A})), \\ \gamma_V(z^\sharp, l^\sharp, a^\sharp) &= \gamma_Z(z^\sharp) \cup \gamma_L(l^\sharp) \cup \gamma_A(a^\sharp).\end{aligned}$$

The abstract integers, abstract labels, and abstract locations need to be Galois connected CPOs:

$$\wp(\mathbb{Z}) \xrightleftharpoons[\alpha_Z]{\gamma_Z} \mathbb{Z}^\sharp \quad \wp(\mathbb{L}) \xrightleftharpoons[\alpha_L]{\gamma_L} \mathbb{L}^\sharp$$

$$\wp(\mathbb{A} = \mathbb{X} \cup \mathbb{N}_{site} \times \mathbb{N}) \xrightleftharpoons[\alpha_A]{\gamma_A} \mathbb{A}^\sharp = \wp(\mathbb{X} \cup \mathbb{N}_{site})$$

Note that the set of variables \mathbb{X} and **mallocsites** \mathbb{N}_{site} are finite for a program; thus, the powersets of those sets can be used as finite abstract domains.

Abstract Transition $\hookrightarrow^\#$ Given the above abstract domains, we define the abstract transition relation $\hookrightarrow^\#$ as follows:

$$*E_1 := E_2 : (l, M^\sharp) \hookrightarrow^\# (\text{next}(l), \text{update}^\#(M^\sharp, \text{eval}_{E_1}^\#(M^\sharp), \text{eval}_{E_2}^\#(M^\sharp)))$$

Note that the abstract transition is identical to the concrete one except that it uses abstract versions for the semantic operators (e.g., $\text{update}^\#$ for update).

The types of the semantic operators are as follows:

$$\begin{aligned}\text{eval}_E^\# &: M^\sharp \rightarrow V^\sharp \\ \text{update}^\# &: M^\sharp \times V^\sharp \times V^\sharp \rightarrow M^\sharp \\ \text{fetch}^\# &: M^\sharp \times V^\sharp \rightarrow V^\sharp\end{aligned}$$

The abstract evaluation operator $\text{eval}^\#$ can be defined as follows:

$$\begin{aligned}\text{eval}_x^\#(M^\sharp) &= \text{fetch}^\#(M^\sharp, x) \\ \text{eval}_{\&x}^\#(M^\sharp) &= \{x\} \\ \text{eval}_{\text{malloc}_u}^\#(M^\sharp) &= \{\mu\} \\ \text{eval}_{*E}^\#(M^\sharp) &= \text{fetch}^\#(M^\sharp, \text{eval}_E^\#(M^\sharp))\end{aligned}$$

Safe Memory Operations If we use the abstract domains of examples 8.2 and 8.3, sound memory read or write operations are as follows:

- The memory read operation $\text{fetch}^\#(M^\sharp, v^\sharp)$ looks up the abstract memory entry at the abstract location $l^\sharp \in \wp(\mathbb{X} \cup \mathbb{N}_{site})$ of v^\sharp . Since

the abstract location is a set of variables and **malloc** sites, the result is the join of all the entries:

$$\bigsqcup_{a \in l^\#} M^\#(a)$$

- The memory write operation $update^\#(M^\#, v_1^\#, v_2^\#)$ overwrites the memory entry (called *strong update*) when the abstract target location $l^\#$ of $v_1^\#$ means a single concrete location. Otherwise, the update cannot overwrite the memory. Every entry in the abstract memory that constitutes the target abstract location $l^\# \in \wp(X \cup N_{site})$ must be joined with the value $v_2^\#$ to store (called *weak update*):

$$\bigsqcup_{a \in l^\#} M^\#[a \mapsto M^\#(a) \sqcup v_2^\#]$$

Example 8.4 (Analysis of a pointer program) Consider the program in figure 8.2 that repeatedly allocates a new memory and overwrites integers to it. The columns labeled Early, Intermittent, and Stable show the snapshots of some entries of the abstract memory at each program point during the analysis. The Stable column shows the final result of the analysis.

We assume that the integer values are abstracted into the integer-interval domain.

First, note the Early column that captures the abstract memory entries right after the first iteration. The **x** variable has the abstract address μ for all the fresh memories allocated at the **malloc** expression. Note the abstract value stored at the μ address. The assignment to the abstract address μ must be the weak update because μ denotes multiple locations. At line 6, μ contains $[0,2]$, not $[2,2]$, as the assigning of $[2,2]$ to μ must be the join with the old value $[0,0]$.

In the Intermittent column, note the abstract values at μ at lines 4 and 6. The assignments do not overwrite but join the new value with the old one. The abstract values stored at μ keep expanding.

| | Early | Intermittent | Stable (= Early ∇ Intermittent) |
|---------------------------------|----------------------|----------------------|--|
| i := 0; | | | |
| 1: | $i \mapsto [0, 0]$ | $i \mapsto [0, 1]$ | $i \mapsto [0, +\infty]$ |
| while(true){ | | | |
| 2: | $i \mapsto [0, 0]$ | $i \mapsto [0, 1]$ | $i \mapsto [0, +\infty]$ |
| x := malloc_μ; | | | |
| 3: | $x \mapsto \mu$ | $x \mapsto \mu$ | $x \mapsto \mu$ |
| *x := i; | | | |
| 4: | $\mu \mapsto [0, 0]$ | $\mu \mapsto [0, 2]$ | $\mu \mapsto [0, +\infty]$ |
| i := i + 1; | | | |
| 5: | $i \mapsto [1, 1]$ | $i \mapsto [1, 2]$ | $i \mapsto [1, +\infty]$ |
| *x := i + 1 | | | |
| 6: } | $\mu \mapsto [0, 2]$ | $\mu \mapsto [0, 3]$ | $\mu \mapsto [0, +\infty]$ |

Figure 8.2

Analysis snapshots of a pointer program

The Stable column shows the abstract memory entries in the end. The expanding upper bounds of the values at i and μ are widened to $+\infty$ and become stable there.

Theorem 8.1 (Safety of $\hookrightarrow^\#$ for the pointer language) Consider the concrete one-step transition of section 8.1.1 and the abstract transition relation of section 8.1.2. If the semantic operators satisfy the soundness properties

$$\begin{aligned} \wp(\text{eval}_E) \circ \gamma_M &\subseteq \gamma \circ \text{eval}_E^\# \\ \wp(\text{update}) \circ \times \circ (\gamma_M, \gamma_V, \gamma_W) &\subseteq \gamma_M \circ \text{update}^\# \\ \wp(\text{fetch}) \circ \times \circ (\gamma_M, \gamma_W) &\subseteq \gamma \circ \text{fetch}^\# \\ \wp(\text{filter}_B) \circ \gamma_M &\subseteq \gamma_M \circ \text{filter}_B^\# \\ \wp(\text{filter}_{\neg B}) \circ \gamma_M &\subseteq \gamma_M \circ \text{filter}_{\neg B}^\#, \end{aligned}$$

then $\check{\wp}(\hookrightarrow) \circ \gamma \subseteq \check{\wp}(\gamma) \circ \hookrightarrow^\#$. (The \times is the Cartesian product operator of multiple sets.)

The proof is similar to the proof of theorem 4.4 (appendix B.3). Note that the definition of $\hookrightarrow^\#$ is homomorphic to that of \hookrightarrow ; that is, the structures of the two definitions are the same except that $\hookrightarrow^\#$ uses the abstract versions for the semantic operators. Thus, the soundness of $\hookrightarrow^\#$ follows rather straightforwardly from the soundness properties of the abstract semantic operators.

| | |
|------------------|--|
| $E ::= \dots$ | expression, as before (figure 4.6) |
| f | function name |
| $C ::= \dots$ | statement, as before (figure 4.6) |
| $E(E)$ | function call |
| return | return from call |
| $F ::= f(x) = C$ | function definition |
| $P ::= F^+C$ | program, list of function defs followed by a statement |

Figure 8.3

Syntax of an example language with functions

8.2 For a Language with Functions and Recursive Calls

8.2.1 Language and Concrete Semantics

Let us consider an imperative language that allows functions. We extend the imperative language in chapter 4 (figure 4.6) with function definitions and recursive calls (see figure 8.3).

The language has only flat function definitions (no nested function definitions) as in the C language. Hence, variables other than function parameters are all global variables. Returning a value from a function is to be simulated by an assignment to a global variable. The function to call is determined at run-time. The function names are first-class values: programmers can store function names in program variables or pass them as parameters to other functions. Hence, like function pointers in C, functions are not necessarily called by their defined names. The function part of the call statement can thus be any expression that evaluates to a function name. We assume that the names in a program for the functions and their parameters are all distinct and that every function has one parameter.

Example 8.5 (Semantics of recursive function calls) Consider the following program:

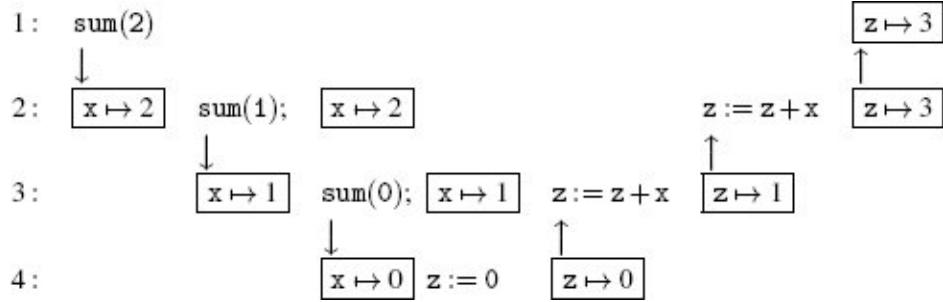
```
sum(x) = if(x = 0) {z := 0; return} else {sum(x - 1); z := z + x; return} sum(2)
```

The `sum` function is recursive, whose result is stored in the `z` variable. After two recursive calls `sum(1)` and `sum(0)`, the result 3 of `sum(2)` is stored in `z`.

The following diagram shows the memory snapshots during the execution. The down arrows are for calls, and up arrows for returns. Each horizontal line shows a part of the program to evaluate by each

call to sum. Line 2 shows the function body after the initial call `sum(2)`, line 3 the function body after the first recursive call `sum(1)`, and so on.

The memory entries for the `x` and `z` variables are shown inside boxes at the program points. Note that the parameter `x` needs to keep maximum three instances alive simultaneously in the memory (to store 2, 1, and 0) during the recursive calls:



Semantic Domains Because of recursive calls, the parameter of a recursive function may have multiple instances alive in the memory at any time during execution. At each recursive call, a new instance of the parameter of the function is allocated in the memory. The function body then uses this new instance as its parameter. The previous instance must be alive and be back to use after the return of the recursive call.

Thus, when defining the semantics of the above language, we need a semantic entity that determines the current instance of function parameters. In semantic jargon, such entity is a table called *environment*. The current environment determines which memory instances of parameters to use.

Also, at function call, the return context must be remembered in order to be recovered on the function return. Each return context consists of the return label (the next label after the function call) and the environment at the function call. Because of recursive calls, we need a stack of remembered contexts, with the most recent context being on the stack top. In semantic jargon, this stack of remembered contexts is called *continuation*.

Thus, we need four components in the state, in addition to the program labels: memory, environment, continuation, and a kind of global counter that we reference to generate fresh instances. The definitions of the semantic domains are shown in figure 8.4.

Given a program, the sets X , L , and F are, respectively, finite sets of variables, labels assigned to each statement, and function names.

An instance $\phi \in I$ is a time stamp, a kind of global counter that is used to differentiate the instances of formal parameters. We use the global counter ϕ as the instance of the formal parameter at each function call. Since a new instance

of a formal parameter is always needed at function calls, the global instance counter ticks at each function call.

Control Flow Is Dynamic Note that the control flow in the presence of function calls is dynamic; the return from a function call is determined at run-time to flow back to its most recent call site. Furthermore, a function is not always called by its defined name. Function to call is not syntactically explicit in the program text when its name is a value that is stored in a variable or passed as a parameter.

| | | |
|--|-------|--|
| $\langle l, m, \sigma, \kappa, \phi \rangle \in$ | $S =$ | $L \times M \times E \times K \times I$ |
| $m \in$ | $M =$ | $A \rightarrow V$ memories |
| $\sigma \in$ | $E =$ | $X \rightarrow I$ environments |
| $\kappa \in$ | $K =$ | $(L \times E)^*$ continuations (stacks of return contexts) |
| $\phi \in$ | I | instances |
| $a \in$ | $A =$ | $X \times I$ addresses |
| $v \in$ | $V =$ | $Z \cup L \cup F$ values |
| | X | set of variables and parameters |
| | L | set of statement labels |
| | F | set of function names |

Figure 8.4

Concrete semantic domains

Other than the dynamically determined control flow, we let the portion of the control flow that is known beforehand from the program syntax be prepared in `next(l)`, `nextTrue(l)`, and `nextFalse(l)`. They determine the next statement to execute upon completing the execution of each l -labeled statement. Every statement of a given program is uniquely labeled.

Concrete Transitional Semantics Concrete semantics of a program for a set I of input states is the least fixpoint

$$\text{lfp } F$$

of monotonic function

$$F : \wp(S) \rightarrow \wp(S)$$

$$F(X) = I \cup \wp(\hookrightarrow)(X).$$

The state transition relation $\langle l, m, \sigma, \kappa, \phi \rangle \hookrightarrow \langle l', m', \sigma', \kappa', \phi' \rangle$ is defined in figure 8.5. The transitions from an l -labeled statement are defined by case analysis over the corresponding statement. For each function f , let $body(f)$ be the label of its body statement, and let $param(f)$ be its formal parameter name.

The expression evaluation function $eval_E$, the memory update function $update_x$, and the filter functions $filter_B$ and $filter_{\neg B}$ are the same as before except that the location of a variable to read or write is the instance of the variable determined by the current environment; hence, they need the current environment σ :

$$\begin{aligned} eval_E &: M \times E \rightarrow V \\ update_x &: M \times V \times E \rightarrow M \\ filter_B &: M \times E \rightarrow M \end{aligned}$$

| | |
|---|---|
| $E_0(E_1) \ : \ \langle l, m, \sigma, \kappa, \phi \rangle \rightarrow \langle \text{body}(f),$ | |
| | $\text{bind}_x(m, \phi', v)$, parameter binding |
| | $\text{new-env}_x(\sigma, \phi')$, new environment |
| | $\text{push-context}(\kappa, \text{next}(l), \sigma)$, new continuation |
| | $\phi' \rangle$ |
| | where $f = \text{eval}_{E_0}(m, \sigma)$ |
| | $x = \text{param}(f)$ |
| | $v = \text{eval}_{E_1}(m, \sigma)$ |
| | $\phi' = \text{tick}(\phi)$ |
| return : | $\langle l, m, \sigma, \kappa, \phi \rangle \rightarrow \langle l', m, \sigma', \kappa', \phi' \rangle$ |
| | where $\langle l', \sigma', \kappa' \rangle = \text{pop-context}(\kappa)$ |
| skip : | $\langle l, m, \sigma, \kappa, \phi \rangle \rightarrow \langle \text{next}(l), m, \sigma, \kappa, \phi \rangle$ |
| input (x) : | $\langle l, m, \sigma, \kappa, \phi \rangle \rightarrow \langle \text{next}(l), \text{update}_x(m, z, \sigma), \sigma, \kappa, \phi \rangle$ for an input int z |
| $x := E$: | $\langle l, m, \sigma, \kappa, \phi \rangle \rightarrow \langle \text{next}(l), \text{update}_x(m, \text{eval}_E(m, \sigma), \sigma), \sigma, \kappa, \phi \rangle$ |
| $C_1; C_2$: | $\langle l, m, \sigma, \kappa, \phi \rangle \rightarrow \langle \text{next}(l), m, \sigma, \kappa, \phi \rangle$ |
| if (B) { C_1 } else { C_2 } | $\begin{aligned} & \rightarrow \langle \text{nextTrue}(l), \text{filter}_B(m, \sigma), \sigma, \kappa, \phi \rangle \\ & : \langle l, m, \sigma, \kappa, \phi \rangle \rightarrow \langle \text{nextFalse}(l), \text{filter}_{\neg B}(m, \sigma), \sigma, \kappa, \phi \rangle \end{aligned}$ |
| while (B) { C } | $\begin{aligned} & \rightarrow \langle \text{nextTrue}(l), \text{filter}_B(m, \sigma), \sigma, \kappa, \phi \rangle \\ & : \langle l, m, \sigma, \kappa, \phi \rangle \rightarrow \langle \text{nextFalse}(l), \text{filter}_{\neg B}(m, \sigma), \sigma, \kappa, \phi \rangle \end{aligned}$ |

Figure 8.5

Concrete transition relation \hookrightarrow

Note that a function name f is a constant expression. The evaluation $\text{eval}_f(m, \sigma)$ is the function name f itself.

The other semantic operations are defined as usual (see figure 8.6).

8.2.2 An Abstract Semantics

Following the framework of chapter 4, we define an abstract semantics as

$$\begin{array}{ll}
 \text{abstract domain} & S^\# = \mathbb{L} \rightarrow \mathbb{M}^\# \times \mathbb{E}^\# \times \mathbb{K}^\# \times \mathbb{I}^\# \\
 \text{abstract semantic function} & F^\# : S^\# \rightarrow S^\# \\
 & F^\#(X^\#) = I^\# \cup^\# Step^\#(X^\#) \\
 & Step^\# = \wp(\text{id}, \cup_R^\#) \circ \pi \circ \wp(\hookrightarrow^\#),
 \end{array}$$

where $\hookrightarrow^\#$ is the one-step abstract transition relation $\subseteq S^\# \times S^\#$. The function π partitions a set $\subseteq L \times M^\# \times E^\# \times K^\# \times I^\#$ by the labels, returning a set $\subseteq L \times \wp(M^\# \times E^\# \times K^\# \times I^\#)$.

$$\begin{aligned}
body &: F \rightarrow L \\
bind_x &: M \times I \times V \rightarrow M \\
new-env_x &: E \times I \rightarrow E \\
push-context &: K \times L \times E \rightarrow K \\
pop-context &: K \rightarrow L \times E \times K \\
tick &: I \rightarrow I
\end{aligned}$$

where

$$\begin{aligned}
bind_x(m, \phi, v) &= m[\langle x, \phi \rangle \mapsto v] \\
new-env_x(\sigma, \phi) &= \sigma[x \mapsto \phi] \\
push-context(\kappa, l, \sigma) &= \langle l, \sigma \rangle . \kappa \quad (\text{stack top } \langle l, \sigma \rangle \text{ and the rest } \kappa) \\
pop-context(\langle l, \sigma \rangle . \kappa) &= \langle l, \sigma, \kappa \rangle \\
tick(\phi) &= \phi' \quad (\text{new } \phi')
\end{aligned}$$

Figure 8.6

Other semantic operators

The $\cup_R^\#$ is an upper bound operator of $M^\# \times E^\# \times K^\# \times I^\#$. The state abstract domain $S^\#$ and the abstract transition relation $\hookrightarrow^\#$ are required to satisfy the framework conditions:

- The abstract state $S^\#$ is to be a CPO, Galois connected with $\wp(S)$:

$$\begin{array}{c}
\wp(L \times M \times E \times K \times I) \\
\overset{\gamma}{\longleftarrow} \underset{\alpha}{\longrightarrow} L \rightarrow M^\# \times E^\# \times K^\# \times I^\#
\end{array}$$

where the abstract component domains are also Galois connected CPOs:

$$\begin{array}{ccc}
\wp(M) & \xrightleftharpoons[\alpha_M]{\gamma_M} & M^\# \\
& & \\
\wp(E) & \xrightleftharpoons[\alpha_E]{\gamma_E} & E^\#
\end{array}$$

$$\begin{array}{ccc}
\wp(K) & \xrightleftharpoons[\alpha_K]{\gamma_K} & K^\# \\
& & \\
\wp(I) & \xrightleftharpoons[\alpha_I]{\gamma_I} & I^\#
\end{array}$$

- The abstract one-step transition relation $\hookrightarrow^\#$ as a function is required to satisfy

$$\check{\rho}(\hookrightarrow) \circ \gamma \sqsubseteq \check{\rho}(\gamma) \circ \hookrightarrow^\#.$$

- The abstract $U^\#$ in $S^\#$ and $U_R^\#$ in $M^\# \times E^\# \times K^\# \times I^\#$ are required to satisfy

$$U \circ (\gamma_-, \gamma_-) \subseteq \gamma_- \circ U_-^\#.$$

Then for any input program, the algorithms (section 4.3) based on such $F^\#$ compute a sound approximation of the concrete semantics of the program.

Memory and Environment Abstract Domains An example of a memory abstract domain $M^\#$ is a Galois connected CPO

$$\rho(M) \xrightleftharpoons[\alpha_M]{\gamma_M} M^\#.$$

Example 8.6 (A memory abstract domain) An example of such an $M^\#$ domain is

$$M^\# = (X \times I^\#) \rightarrow V^\#,$$

given that $V^\#$ is a Galois connected CPO. The set X is the finite set of variables and parameters in the input program to analyze.

An example of an environment abstract domain $E^\#$ is a Galois connected CPO

$$\rho(E) \xrightleftharpoons[\alpha_E]{\gamma_E} E^\#.$$

Example 8.7 (An abstract environment domain) An example of such an $E^\#$ domain is

$$E^\# = X \rightarrow I^\#.$$

An example value abstract domain $V^\#$ is a Galois connected CPO

$$\rho(V) \xrightleftharpoons[\alpha_V]{\gamma_V} V^\#.$$

Example 8.8 (An abstract value domain) An example $V^\#$ domain is achieved by abstracting kind-wise a set of values (integers, labels, and/or function names). We abstract its set of integers into an abstract integer, its set of labels into an abstract label, and a set of function names into an abstract function name:

$$\wp(\mathbb{Z} \cup \mathbb{L} \cup \mathbb{F}) \xleftarrow[\alpha_Z]{\gamma_Z} \mathbb{Z}^\sharp \times \mathbb{L}^\sharp \times \mathbb{F}^\sharp,$$

where the abstract integers, labels, and locations are Galois connected CPOs:

$$\wp(\mathbb{Z}) \xleftarrow[\alpha_Z]{\gamma_Z} \mathbb{Z}^\sharp \quad \wp(\mathbb{L}) \xleftarrow[\alpha_L]{\gamma_L} \mathbb{L}^\sharp \quad \wp(\mathbb{F}) \xleftarrow[\alpha_F]{\gamma_F} \mathbb{F}$$

Note that the variables X and the function names F are finite sets for a program; thus, the powersets of those sets are already finite, eligible to be used as abstract domains in static analysis.

Abstract Transition $\hookrightarrow^\#$ The abstract state transition relation for the function call and return statements are shown in figure 8.7. Basically, the definition of $\hookrightarrow^\#$ is the same as that of \hookrightarrow except that $\hookrightarrow^\#$ uses abstract semantic operators for their concrete correspondents.

$$\begin{aligned}
 E_0(E_1) : \langle l, M^\sharp, \sigma^\sharp, \kappa^\sharp, \phi^\sharp \rangle &\hookrightarrow^\# \langle \text{body}(f), \\
 &\quad \text{bind}_x^\sharp(M^\sharp, \phi'^\sharp, v^\sharp), \quad \text{parameter binding} \\
 &\quad \text{new-env}_x^\sharp(\sigma^\sharp, \phi'^\sharp), \quad \text{new environment} \\
 &\quad \text{push-context}^\sharp(\kappa^\sharp, \text{next}(l), \sigma^\sharp), \quad \text{new continuation} \\
 &\quad \phi'^\sharp \rangle \\
 &\quad \text{where } f \in \text{eval}_{E_0}^\sharp(M^\sharp, \sigma^\sharp) \\
 &\quad x = \text{param}(f) \\
 &\quad v^\sharp = \text{eval}_{E_1}^\sharp(M^\sharp, \sigma^\sharp) \\
 &\quad \phi'^\sharp = \text{tick}^\sharp(\phi^\sharp) \\
 \text{return} : \langle l, M^\sharp, \sigma^\sharp, \kappa^\sharp, \phi^\sharp \rangle &\hookrightarrow^\# \langle l', M^\sharp, \sigma'^\sharp, \kappa'^\sharp, \phi^\sharp \rangle \\
 &\quad \text{where } \langle l^\sharp, \sigma'^\sharp, \kappa'^\sharp \rangle = \text{pop-context}^\sharp(\kappa^\sharp) \text{ and } l' \in l^\sharp
 \end{aligned}$$

Figure 8.7

Abstract transition relation $\hookrightarrow^\#$ for function call and return

The abstract semantic operators are homomorphic to their concrete correspondent except that they are defined over abstract domains:

$$\begin{aligned}
bind_x^\# &: M^\# \times I^\# \times V^\# \rightarrow M^\# \\
new-env_x^\# &: E^\# \times I^\# \rightarrow E^\# \\
push-context^\# &: K^\# \times L^\# \times E^\# \rightarrow K^\# \\
pop-context^\# &: K^\# \rightarrow L^\# \times E^\# \times K^\# \\
tick^\# &: I^\# \rightarrow I^\# \\
eval_E^\# &: M^\# \times E^\# \rightarrow V^\# \\
update_x^\# &: M^\# \times V^\# \times E^\# \rightarrow M^\# \\
filter_B^\# &: M^\# \times E^\# \rightarrow M^\#
\end{aligned}$$

Theorem 8.2 (Safety of $\hookrightarrow^\#$ for the language with functions) Consider the concrete one-step transition of section 8.2.1 and the abstract transition of section 8.2.2. If the semantic operators satisfy the soundness properties

$$\begin{aligned}
\wp(bind_x) \circ \times \circ (\gamma_M, \gamma_I, \gamma_V) &\subseteq \gamma_M \circ bind_x^\# \\
\wp(new-env_x) \circ \times \circ (\gamma_E, \gamma_I) &\subseteq \gamma_E \circ new-env_x^\# \\
\wp(push-context) \circ \times \circ (\gamma_K, \gamma_L, \gamma_E) &\subseteq \gamma_K \circ push-context^\# \\
\wp(pop-context) \circ \gamma_K &\subseteq \times \circ (\gamma_L, \gamma_E, \gamma_K) \circ pop-context^\# \\
\wp(tick) \circ \gamma_I &\subseteq \gamma_I \circ tick^\# \\
\wp(eval_E) \circ \times \circ (\gamma_M, \gamma_E) &\subseteq \gamma_V \circ eval_E^\# \\
\wp(update_x) \circ \times \circ (\gamma_M, \gamma_V, \gamma_E) &\subseteq \gamma_M \circ update_x^\# \\
\wp(filter_B) \circ \times \circ (\gamma_M, \gamma_E) &\subseteq \gamma_M \circ filter_B^\#
\end{aligned}$$

then $\check{\wp}(\rightarrow) \circ \gamma \sqsubseteq \check{\wp}(\gamma) \circ \rightarrow^\#$. (The \times is the Cartesian product operator of multiple sets.)

The proof is similar to the proof of theorem 4.4 (appendix B.3). Since the definition of $\hookrightarrow^\#$ has the same structure as that of \hookrightarrow except that it uses abstract versions for concrete semantic operators, the soundness of $\hookrightarrow^\#$ naturally follows from the soundness of the abstract semantic operators.

Varying the Context Sensitivity In a concrete execution, every call to a function may happen in a unique machine state. The global variables may have unique values at each call time, and in case of recursive calls the stacked instances of the formal parameter may be different at each call.

If the static analysis abstracts, for each function, all the machine states at the function's multiple calls into a single abstract state, then the analysis is called *context insensitive*. If the static analysis abstracts the multiple call contexts into multiple abstract contexts to distinguish some differences of the call contexts, then the analysis is called *context sensitive*.

The context sensitivity boils down to how we abstract the instance domain I :

$$\wp(I) \xrightleftharpoons[\alpha_I]{\eta} I^\#$$

If $I^\#$ is a singleton set (i.e., if we do not differentiate the parameter instances in abstract semantics), then follows the context insensitivity.

On the other hand, by using multiple elements for $I^\#$, some kind of context sensitivity emerges. For example, suppose that $I^\#$ uses the set C_{site} of every call sites of the input program:

$$I^\# = \wp(C_{site})$$

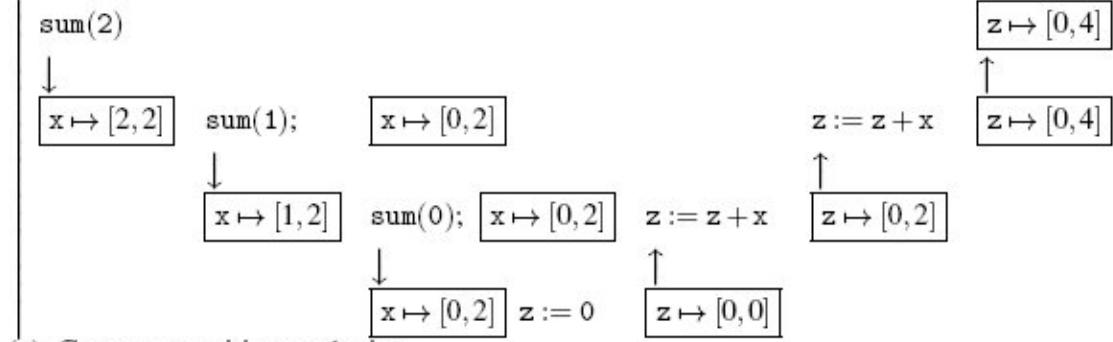
Then multiple abstract instances of each parameter, each of which corresponds to each call site, emerge.

A more elaborate context sensitivity is possible if we use, say, abstract *call strings* [57] for $I^\#$. A call string is an ordered sequence of function names that abstracts the continuation at the moment of a call. The larger the maximum length of the abstract call strings in

(a) Example program:

```
sum(x) = if(x = 0) {z := 0; return} else {sum(x - 1); z := z + x; return}
sum(2)
```

(b) Context-insensitive analysis:



(c) Context-sensitive analysis:

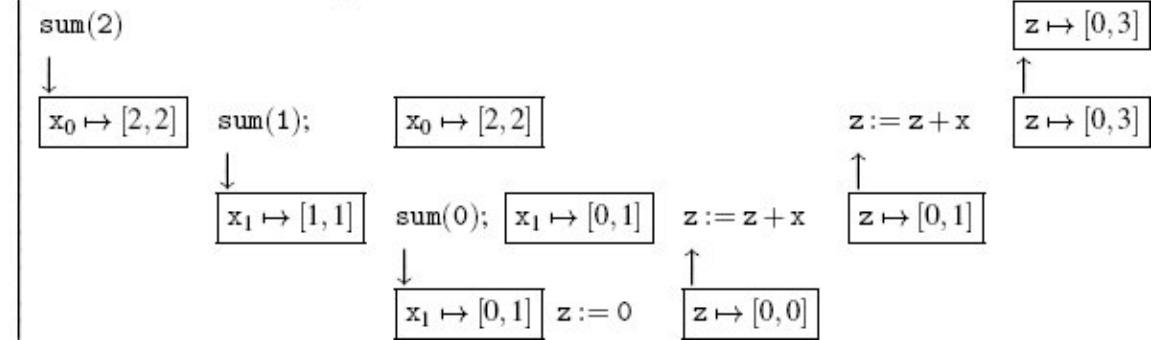


Figure 8.8

Context-insensitive and -sensitive analyses of an example program

$I^{\#}$, the more varied abstract instances are used for a single parameter, hence follows a more elaborate context-sensitive analysis. The precision improvement from the context sensitivity comes with a cost. We can selectively apply the context sensitivity only to those calls that eventually contribute to the improvement of the analysis precision [86].

Example 8.9 (Analyses with different context sensitivities) The example program in figure 8.8 is the same as in example 8.5. Suppose we use the integer intervals for abstract integers. Figure 8.8(b, c) show the abstract memory entries at each program point during analysis. The down arrows are for calls and the up arrows are for returns. Each horizontal line shows a part of the program to analyze by each call to `sum`.

- *Figure 8.8(b) shows the abstract memory entries at each program point during the context-insensitive analysis. Recall that context insensitivity means that we use a single abstract location for all the instances of each function's parameter. That is, the abstract instance domain $I^\#$ is a singleton set. In this abstraction, the abstract value [0,2] of the abstract location x at the beginning of the function body subsumes all the values passed to x . The z variable at the end of the function body also has [0,4] that covers all the values stored there.*
- *Figure 8.8(c) shows the abstract memory entries at each program point during a context-sensitive analysis. Recall that context sensitivity boils down to using multiple abstract locations for the instances of each parameter. Suppose that we abstract the instances by the call sites. Thus, for the example program we have two abstract instances of x : one x_0 for the initial call of $\text{sum}(2)$ and the other x_1 for the recursive calls of $\text{sum}(1)$ and $\text{sum}(0)$.*

Note that the first instance x_0 of x for the initial call to sum has [2,2]. The second instance x_1 has [0,1] that covers all the values bound to x by the two recursive calls. These two call-site-sensitive abstract locations for x result in a precise analysis for the final abstract value of z in the end. At the finish of the two recursive calls, z contains [0,1], not [0,2] as in the context-insensitive case, because it involves only x_1 . At the finish of the initial call, z contains [0,3], having the (weak update) effect of the assignment of the sum of [0,1] (z) and [2,2] (the value of x_0).

8.3 Abstractions for Data Structures

The previous sections presented a general approach to the design of static analysis for programming languages with pointers and with functions. This section and the next one take a complementary step and study abstractions that are specific to families of data structures and functions. Their purpose is to provide readers with a good intuition of what kind of abstraction should be used in any given practical case. On the other hand, they provide a higher-level view of the analysis algorithms.

In this section, we study abstractions for data structures:

- arrays (section 8.3.1);
- buffers and strings (section 8.3.2);
- statically allocated pointer structures (section 8.3.3); and
- dynamically allocated pointer structures (section 8.3.4).

8.3.1 Arrays

Arrays are one of the most common data structures. An array reserves a contiguous memory region of fixed size, an element of which can be accessed using an integer index. Therefore, vectors and matrices are naturally

represented as arrays. Similarly, arrays are quite adapted to the storage of tables of data. For instance, we encounter such structures in all sorts of applications ranging from user-level databases (each entry corresponds to one array cell) to operating system internal states (each process corresponds to a cell in a process table array). There are even some programming languages designed around the notion of arrays, such as spreadsheet environments (Excel and similar) and array programming languages (APL, Fortran, and similar).

Extension of the Language Syntax and Semantics To illustrate the issues inherent in the abstraction of arrays, we consider a minimal extension of the language of chapter 3 with arrays of integer values and of arbitrary length. While real-world languages feature arrays of other data types, including structures, arrays of scalars are enough to illustrate the main concepts.

We reuse and extend the notations of chapter 3, although we adopt a much less formal point of view here, as we focus on the abstraction and do not intend to formalize static analyses. We assume that each variable either stores a scalar value in V or consists of an array that stores elements of V . Array elements are indexed by integers ranging from 0 to $n - 1$, where n is the length of the array:

$$M = X \rightarrow \left(V \cup \left(\bigcup_{n>0} [0, n-1] \rightarrow V \right) \right)$$

To account for all typical array operations, we need to add to the language basic constructions to create, read, and write into arrays:

- **Creation:** We let the statement **create** (t, n) create a new array, the length of which is the value stored in variable n , and store it into variable t ; the contents of array cells is unknown (after creation, each cell may store any value).
- **Fread:** We let the statement $x := t[n]$ read the contents of the cell of index n in array t , if this cell exists, and copy its value into the variable x ; otherwise, it causes a run-time error.
- **Write:** We let the statement $t[n] := x$ write the contents of the variable x into the cell of index n of the array t , if this cell exists; otherwise, it causes a run-time error.

Note that array creation fails when n does not store a strictly positive integer. In the case of an array cell read or write, the correct evaluation is possible only when the index is valid, that is, if it lies within the bounds of the array. Otherwise, the evaluation results in a *run-time error* that terminates the

execution of the program (section 3.1.2). This semantics is close to that of the Java programming language. However, the semantics of C leaves the behavior of such invalid accesses undefined, which means that they may cause abrupt crashes or corrupt other memory cells.

As an example, figure 8.9 displays two versions of an array creation and initialization routine. The version of figure 8.9(a) presents a correct implementation, whereas the version of figure 8.9(b) is incorrect and contains a common “off by one” index bug (it starts at 1 and ends at n). As a consequence, under the assumption that n initially contains a strictly positive integer, the first program will create and initialize a new array of length n , whereas the second program will crash.

Semantic Properties Related to Arrays Array-manipulating programs are often hard to get correct and are prone to several kinds of problems. One common issue is the existence of errors related to attempts to access a cell out of the bounds of the array. This is the case of the program of figure 8.9(b).

Another common issue is the breakage of more complex invariants that are *global* to the array. For instance, fixing the loop exit condition of the program of figure 8.9(b) will prevent a run-time error but will not make the program fully correct; indeed, the resulting code would still omit initializing the value of the first cell (at index 0).

| | |
|---|---|
| <pre>create(t,n); i := 0; while(i ≤ n - 1){ t[i] := v; i := i + 1 }</pre> | <pre>create(t,n); i := 1; while(i ≤ n){ t[i] := v; i := i + 1 }</pre> |
| (a) Correct code | (b) Incorrect code |

Figure 8.9

Examples of array creation and initialization programs (we assume n is a predefined integer variable that stores a positive value, and v is a scalar variable that stores the value that should be used for initialization)

There are many other challenging properties related to arrays. For instance, a sorting routine needs to ensure several global properties: at the end, the array should not only be sorted, but also contain the same values as the initial array. Moreover, an initialization or sorting routine typically treats the input array cell by cell, which means that intermediate states satisfy even more complex invariants. As an example, if we observe the state of the program of figure 8.9(a) after a few iterations, we note that the first cells of the array are initialized while the others may still contain any value.

In the following paragraphs, we discuss a few existing abstractions to reason about such semantic properties. These are representative of general array abstract domains.

Abstractions to Reason on the Correctness of Array Indexes Array cell indexes are numerical values; hence, the verification of array indexes boils down to an abstract interpretation using a numerical abstract domain such as those presented in section 3.2. When constraints that bound several variables are required, a relational abstract domain should be employed. As an example, the programs of figure 8.9 contain accesses to an array of length n , the value of which is not statically known; thus, a relational domain such as octagons or convex polyhedra is needed here. Assuming either of these two domains, a static analysis similar to those presented in chapters 3 and 4 would produce the following results:

- for the program of figure 8.9(a), $0 \leq i \leq n - 1$ inside the loop, so that all accesses are correct;
- for the program of figure 8.9(b), $1 \leq i \leq n$ inside the loop, which does not allow proving the absence of run-time errors (and as observed above, this program will indeed cause an out-of-bound access error).

If the length of the array was equal to a known, fixed integer value instead, interval constraints would suffice.

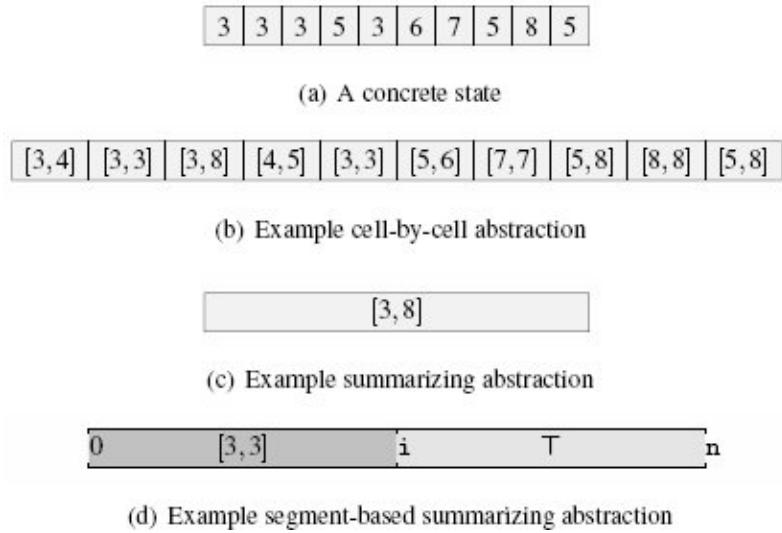


Figure 8.10

Array contents abstractions: several abstractions of an array t

Abstraction of Array Contents and Summarization Let us now turn to the inference of properties related to array contents. A reference concrete state is given in figure 8.10(a).

A basic approach attaches one abstract constraint to each cell. This is quite a straightforward generalization of the analyses presented in chapters 3 and 4. This *cell-by-cell* abstraction is depicted in figure 8.10(b), assuming each variable and array cell is abstracted by an interval. This abstraction is able to carry very precise information; for instance, figure 8.10(b) expresses that the contents of $t[1]$ is 3. On the other hand, some other cells may be abstracted in a less precise way, just like any scalar variable; this is the case of $t[0]$, which may store either 3 or 4 according to the abstraction. Unfortunately, it is far from solving the problem of array contents abstraction. First, the cell-by-cell abstraction does not rely on any regularity inside the array and thus overlooks any chance to use a more compact logical statement such as “all cells contain value v .” This is very costly when the array is large. Second, it cannot cope with arrays the length of which is not known statically, which happens in both programs in figure 8.9. Last, while the analysis algorithm seem quite similar to

those used in previous chapters for programs with a set of variables, it actually requires identifying the cells that may be modified by each statement.

Another more compact way to describe the contents of an array is to let a single abstract symbol represent the set of all the values stored in the array. We call this special abstract symbol a *summary* since it summarizes the contents of the whole array. This technique is illustrated in figure 8.10(c): this example abstraction expresses that all the cells in the array store a value in interval [3,8]. Such a constraint is not only expressive but also compact, and it can be expressed whatever the length of the array.

Both abstractions require special operations to handle array creation and access constructions. The analysis of an array creation is straightforward, as it boils down to the definition of an abstract state as shown in figure 8.10. The analysis of a read joins the information that may be read for all the cells in the index range; for instance, if i is in [2,4], the analysis infers that $t[i]$ is in [3,7] in figure 8.10(b) and in [3,8] in figure 8.10(c). The analysis of a write operation is more complex, as also discussed in section 8.1.2:

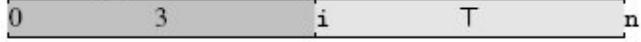
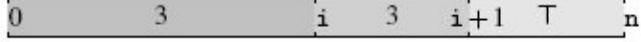
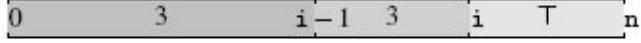
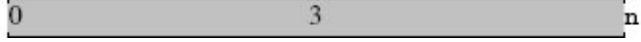
- When the information on the index designates a single cell that is precisely known, the analysis of the write operation is similar to that of a write into a program variable. We call such a case a *strong update*. For instance, if $i = 2$, the analysis of $t[i] := x$ with the cell-by-cell abstraction (figure 8.10(b)) copies the information about x into the cell of index 2.
- When the index may designate several distinct cells, or when updating a summary, the analysis needs to be conservative and to account both for cases where a cell is modified and for cases where it is not. We call such a case a *weak update*. Using the abstraction of figure 8.10(b), if i is known to be in the range [0,1], and no other information is available about i , the analysis of $t[i] := 5$ needs to carry out a weak update and replace the information about *both* of these two cells with a less precise range [3,5]. Indeed, the analysis should account for the case where $t[0]$ is modified and $t[1]$ is left intact, and for the symmetric case as well. Using the abstraction of figure 8.10(c), any write operation leads to a weak update; for instance, the analysis of $t[3] := 9$ should replace the information about all the cells with range [3,9].

Abstraction Based on Array Partitioning Weak updates present a real challenge for analyses that rely on summaries, so we show another family of array abstractions that avoid this issue. As observed above, summarization is essential to describe array regions of unknown and/or of unbounded size. We also remarked the importance of strong updates to infer precise invariants. Array abstractions based on *dynamic array partitioning* [48, 55, 29] combine these two ingredients and allow proving properties such as correct initialization (as in figure 8.9(a)) or sortedness. They partition the array into segments as in figure 8.10(d), so as to abstract each block of cells separately: in this figure, we see that the array can be divided into two zones, one that comprises cells with an index ranging from 0 to $i - 1$, and one that comprises the other cells. Moreover, we also observe that all the cells of the first zone are known to contain the value 3.

Array-partitioning abstract domains rely on sophisticated algorithms since the analysis needs to compute not only content predicates but also partitions of arrays into segments. To demonstrate these algorithms, we consider the program of figure 8.9(a), with $v = 3$. Figure 8.11 shows the invariants computed at each program point, and we discuss the main analysis steps based on this figure. First, the analysis of the array creation command generates a single segment that encompasses all the cells in the array and does not tie any content information to them. The analysis of the assignment command $t[i] := 3$ inside the loop splits this segment so as to materialize the cell that is modified and to carry out a strong update. A widening operator generalizes the abstract states observed at the loop head and synthesizes two segments: the first segment includes all the cells already initialized, whereas the second one describes all the other cells (it is the same as the abstract state shown in figure 8.10(d)). The invariants shown in figure 8.11 show the effect of the two assignment commands: the analysis of the assignment to the array cell $t[i]$ creates a new segment of length 1, that describes exactly the cell that is modified (hence the strong update), whereas the analysis of the assignment to i causes the recomputation of the segment bounds. Finally, the loop exit abstract state shows that all the cells of the array are initialized. A similar algorithm would allow proving the correctness of a sorting program, provided more expressive content predicates are used.

```

create(t,n);
i := 0;

while(i ≤ n − 1){

    t[i] := 3;

    i := i + 1

}


```

Figure 8.11

Static analysis based on array partitioning

8.3.2 Buffers and Strings

Buffers rely on a representation similar to arrays and with some specific algorithms to add (and, possibly, to remove) pieces of data. A common implementation of a buffer consists of an array with a separate variable that indicates the first free position. Each array cell stores an atomic piece of data. When cells store characters, we call such structures *string buffers*. The end of a string buffer is marked by a special character (e.g., ASCII code $0x0$) that we denote by ϕ in the following. String buffers are common and have been reported as the source of many security issues; thus, they are a very important static analysis target.

As an example, figure 8.12 displays three string buffers; moreover, $b0$ represents string “abc,” $b1$ stands for string “de,” and so on.

| | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--------|--|---|---|---|---|---|---|---|---|--------|--|---|---|---|---|---|---|--------|--|---|---|---|---|---|---|---|---|
| $s_0:$ | <table border="1"> <tr> <td>a</td><td>b</td><td>c</td><td>∅</td><td>u</td><td>v</td><td>u</td><td>v</td></tr> </table> | a | b | c | ∅ | u | v | u | v | $s_1:$ | <table border="1"> <tr> <td>a</td><td>d</td><td>∅</td><td>w</td><td>w</td><td>u</td></tr> </table> | a | d | ∅ | w | w | u | $s_2:$ | <table border="1"> <tr> <td>a</td><td>b</td><td>c</td><td>a</td><td>d</td><td>∅</td><td>u</td><td>v</td></tr> </table> | a | b | c | a | d | ∅ | u | v |
| a | b | c | ∅ | u | v | u | v | | | | | | | | | | | | | | | | | | | | |
| a | d | ∅ | w | w | u | | | | | | | | | | | | | | | | | | | | | | |
| a | b | c | a | d | ∅ | u | v | | | | | | | | | | | | | | | | | | | | |

Figure 8.12

A few string buffers

Extension of the Language Syntax and Semantics We use the same language as in section 8.3.1 (with commands for the creation of a buffer, and for the read/write of a buffer cell), with additional high-level operations, such as the following:

- **Initialization to a given string:** the operation $s := "..."$ stores a constant string specified as a sequence of characters into the buffer s and adds ϕ at the end, if the buffer is long enough; otherwise, it causes a run-time error.
- **Append:** the operation $\text{append}(s, t)$ copies the characters of s until the first ϕ (included) from the position of the first in ϕ t ; if s or t contains no ϕ , or if it attempts to write beyond the end of t , it causes a run-time error.

For instance, the state in figure 8.12 can be observed after the execution of the sequence of operations $s_0 := "abc"; s_1 := "ad"; s_2 := "abc"; \text{append}(s_1, s_2)$. The two operations mentioned above are typically implemented into library functions, which use basic array operations.

Semantic Properties Related to Buffers and Strings Operations on string buffers are potentially dangerous in many ways. First, initialization and append may fail when the target buffer is not large enough. While our language extension assumes that programs will terminate with a run-time error in such a case, real-world implementations (e.g., C string libraries) may just carry out the write operation and overwrite space beyond the buffer area. Such a corruption of the memory state is often even worse than an abrupt crash since it makes it very hard to predict the behavior of the program. In some cases, buffer overflows may allow a malicious user to take over the execution of a program.

Second, a read operation (using the usual array read command $s[i]$) beyond the bounds of a string buffer also represents a serious security concern. Indeed, real-world implementations typically just read the value stored at the

corresponding address, even if it is sensitive data that should not be leaked. This issue is actually the root cause of the Heartbleed vulnerability in OpenSSL implementation [37].

Last, content properties sometimes also need to be enforced. This is the case when strings should contain valid data descriptions, such as a valid number followed by a currency symbol or a valid URL.

Abstraction to Reason on Correctness of String Buffer Operations The verification of absence of run-time errors related to buffer overflows boils down to checking numerical relations [102, 36]. Given a buffer s , we denote its length by $\text{len}(s)$ (so that valid positions in the buffer range from 0 to $\text{len}(s) - 1$), and $\text{zero}(s)$ is the position of the first ϕ in s , if any (if there is none, we let $\text{zero}(s) = \text{len}(s)$). Then a string buffer is well-formed if and only if it contains at least one occurrence of ϕ , that is, if $\text{zero}(s) < \text{len}(s)$. As an example, in figure 8.12, $\text{len}(t_0) = 8$ and $\text{zero}(t_0) = 3$.

An initialization $s := "..."$ succeeds if and only if $\text{len}(s)$ is strictly greater than n , where n is the length of the string in the right-hand side; afterward it contains a ϕ at position n . Moreover, $\text{append}(s, t)$ succeeds if and only if $\text{zero}(t) + \text{zero}(s) < \text{len}(t)$, and afterward the position of the first ϕ in t can also be computed precisely. For these reasons, it is possible to reason over the correctness of string buffer operations using a numerical static analysis, which computes constraints over the program variables and the $\text{len}(\cdot)$ and $\text{zero}(\cdot)$ metavariables describing buffers. This approach reduces the problem of verifying the correctness of buffer operations to the computation of numerical invariants so as to verify constraints over buffer lengths and sizes [102, 36]. Thus, static analyses similar to those shown in chapters 3 and 4 apply. The use of a relational domain for the representation of the numerical constraints is preferable since the verification of an **append** operation requires constraints that bind several variables. The verification of programs using (non-string) buffers typically requires similar abstractions and analysis techniques.

To illustrate this technique, we present in figure 8.13 the results of the analysis of a short series of commands that operate on two buffers s_0 and s_1 . The numerical constraints inferred by a forward static analysis using the $\text{len}(\cdot)$ and $\text{zero}(\cdot)$ numeric metavariables are shown between commands. We remark that the assignment into s_1 and the **append** operation can be proved safe using these invariants. Moreover, the analysis proves that $\text{zero}(s_1) < n$ at the

exit point of the program, which means that the content of `s1` is guaranteed to be a well-formed buffer, with a zero before its final position.

Abstractions to Reason over String Contents There exist many abstractions to describe the actual contents of strings viewed as character sequences. As an example, abstractions based on occurrence counts based on the Parikh vector [89], describe properties such as “the number of occurrences of character ‘*a*’ is less than 3” or “character ‘*z*’ does not occur at all.” Abstractions based on regular expressions [80] can describe properties such as the well-formedness of a URL. Abstractions based on grammars can express, for example, that a string describes a well-parenthesized expression.

8.3.3 Pointers

Most programming languages feature pointers in one form or another. Essentially a pointer allows referring to a stored object via its actual numeric address (which depends on the program execution) rather than via a fixed name. It is thus possible to share, pass, modify, and utilize data structures in a very flexible manner. As an example, passing a parameter by reference allows a function to easily access or modify a large data area. Even purely functional languages make an extensive use of pointers, with closures and shared immutable objects. Pointers greatly enhance expressiveness but often make programming more challenging and error prone. Therefore, pointers are also notoriously important yet difficult to deal with when doing static analysis.

```

create(s0,5);
create(s1,n);
s0 := "cqfd";
    len(s0) = 5  $\wedge$  len(s1) = n  $\wedge$  zero(s0) = 4  $\wedge$  zero(s1) = + $\infty$   $\wedge$  i < n
s1[i] = &;

    len(s0) = 5  $\wedge$  len(s1) = n  $\wedge$  zero(s0) = 4  $\wedge$  zero(s1) = i  $\wedge$  i < n
if(i + 4 < n){
    len(s0) = 5  $\wedge$  len(s1) = n  $\wedge$  zero(s0) = 4  $\wedge$  zero(s1) = i  $\wedge$  i + 4 < n
    append(s0,s1);
    len(s0) = 5  $\wedge$  len(s1) = n  $\wedge$  zero(s0) = 4  $\wedge$  zero(s1) = i + 4  $\wedge$  i + 4 < n
}
    len(s0) = 5  $\wedge$  len(s1) = n  $\wedge$  zero(s0) = 4  $\wedge$  zero(s1) < n

```

Figure 8.13

Analysis of a sequence of string buffer operations (we assume that $0 \leq i < n$ at the entry point)

This subsection discusses issues relevant to the manipulation of pointer values whereas subsection 8.3.4 studies dynamic memory allocation. This section also introduces abstractions that may be used interchangeably in the analysis formalized in section 8.1.

Extension of the Language Syntax and Semantics In chapters 3 and 4, we let the set of values, V consist of scalars (integer or floating-point numbers or labels). Variables of pointer types store an *address*. This changes not only the definition of the set of values but also the definition of memory states. Indeed, it requires V to also include a set of addresses A and memory states to map addresses in A into the contents of the corresponding cells. Since dynamic memory allocation is considered later in section 8.3.4, we assume here that addresses simply correspond to variables: a variable name x is stored at an address $\&x$. Although we do not make explicit use of a type system, we will consider only programs that do not lead to mismatches such as comparing a pointer with an integer or multiplying a pointer by a scalar.

There are two main operations that are specific to pointers:

- **Computation of an address:** the assignment $p := \&x$ stores the address of variable x into variable p .
- **Dereference:** the expression $*p$ denotes the memory location the address of which is the value of p , if p contains a valid address, and fails with a run-time error otherwise.

```

x := 0; y := 0;
x := 0;
if(?) {
    p := &x;
}
*p := 8;
(a) Unsafe dereference

if(?) {
    p := &x;
} else {
    p := &y;
}
*p := 8;
(b) Weak update

```

Figure 8.14

Two programs manipulating pointers

It can be used in either side of an assignment; indeed, $y := *p$ stores into y the value read inside that cell, whereas $*p := y$ stores the value of y into that cell.

Figure 8.14 displays two programs that use a pointer variable. We let **?** denote a condition that non-deterministically evaluates to **true** or to **false**. According to our semantics, the fragment presented in figure 8.14(a) may cause a run-time error, if the condition evaluates to **false** and p does not get assigned a correct address. On the other hand, the program of figure 8.14(b) is memory safe. Although these programs are quite contrived, they will help us illustrate a few salient issues related to the analysis of programs with pointers.

Semantic Properties Related to Pointers We remarked that pointer dereference may cause a run-time error when the value of the pointer is not a valid address. In real programming languages such as C, incorrect pointer operations as in figure 8.14(a) may have undefined behaviors and cause an unspecified memory location to be modified. Thus, a first application of static analysis related to pointers is to check the *memory safety*. Intuitively, it boils down to checking the absence of null, dangling, or invalid pointer dereferences.

A second application is to support client program analyses or transformations. As an example, the program of figure 8.14(b) performs a numerical assignment via a pointer; this means that the value of the numerical variables at the end of the execution can be known only if we have information about the value of the pointer variable p . This situation is actually fairly common: whenever a program manipulates pointers, the effect of a read/write memory operation is not so obvious, as it is possible to modify a memory cell without referring explicitly to it (provided we have stored its address in a pointer variable). Thus, many static analysis tools (or even compiler optimizers) have to compute pointer information beforehand.

In the following paragraphs, we highlight two families of abstractions for pointers that can be used for both of these goals.

Abstractions of Pointer Values The most intuitive approach is to use a non-relational abstraction of pointer values that attempts to describe the value that each pointer variable may take separately from the other variables. It is very similar to the other non-relational abstractions presented in section 3.2.2, except that the abstraction for values describes sets of addresses.

A basic such abstraction consists of *points-to sets*: the abstraction of a pointer boils down to a set of addresses it may point to [104, 58]. Special care must be taken for invalid pointer values. Thus, a points-to set may include special symbols that denote the null pointer or an invalid pointer value. We note **0x0** for the null pointer and **0x?** for an invalid pointer. We also write T for a points-to set that may contain any possible address, for instance, when a pointer variable has not been properly initialized. To illustrate the results that an analysis based on points-to sets may compute, we consider the two programs of figure 8.14:

- After the condition in the program of figure 8.14(a), the points-to set for p is T , which does not allow verifying the final assignment; as observed above, this statement may indeed cause a run-time error, so it is unsurprising that the analysis catches this issue.
- At the end of the program of figure 8.14(b), the points-to set for p is $\{\&x, \&y\}$; this information is actually sufficient to verify that the final assignment does not cause an error since it proves that p may point only to a valid address.

The computation of such points-to sets proceeds by standard forward abstract interpretation using points-to sets as a value abstraction. Except for the value

abstraction, it is very similar to the analyses presented in chapters 3 and 4.

Points-to sets provide only a rather naive abstraction and cannot cope with programs where the number of possible addresses is high or unbounded, since that would mean points-to sets may also become very large. An alternative technique drops all information about points-to sets that are too big; for instance, *k-limiting* turns any points-to set that contains more than k elements into a special abstract value \top . More advanced abstractions able to represent unbounded sets of concrete addresses are presented in section 8.3.4.

Abstractions of Aliasing Relations Another family of abstractions for pointer values consists of *aliasing* relations across pointers [73, 33]. We say that two pointers are aliased if and only if they store the same address. Abstracting the aliasing relation allows expressing relational properties between pointers. As an example, it is possible to state that two pointer variables p and q store the same address, even though this address is only known to be either $\&x$ or $\&y$. Due to this, points-to set abstractions can be viewed as abstractions of the aliasing relation, but the opposite is generally not true.

Pointer Analysis and Weak Updates Analyses based on abstractions for points-to sets or aliasing relations rely on the principles presented in chapters 3 and 4. An important remark is that, due to the nature of pointers, the analysis of read/write operations may need to account for cases where memory cells cannot be determined precisely. The very same issue led to *weak updates* in sections 8.1.2 and 8.3.1. Likewise, weak updates may also arise in pointer analyses. As an example, we discuss the analysis of the last statement in the program of figure 8.14(b). The complete results are shown in figure 8.15. As we have noted earlier, the most precise information that can be computed about pointer p is that it may point either to $\&x$ or to $\&y$. Because of this, it is impossible to precisely say what the value of x is after the assignment $*p := 8$. In fact, the analysis needs to account for this statement by carrying out a weak update, which means that it needs to consider cases where either x is modified and y is unchanged, or x is unchanged and y is modified.

```

x := 0; y := 0;
x = 0 ∧ y = 0 ∧ p ↦ T
if(?) {
    x = 0 ∧ y = 0 ∧ p ↦ T
    p := &x;
    x = 0 ∧ y = 0 ∧ p ↦ {&x}
}else{
    x = 0 ∧ y = 0 ∧ p ↦ T
    p := &y;
    x = 0 ∧ y = 0 ∧ p ↦ {&y}
}
    x = 0 ∧ y = 0 ∧ p ↦ {&x, &y}
*p := 8;
x ∈ [0, 8] ∧ y ∈ [0, 8] ∧ p ↦ T

```

Figure 8.15

Points-to analysis

8.3.4 Dynamic Heap Allocation

While section 8.3.3 discusses a few analyses for some programs that manipulate pointers, it omits important language features that are often used together with pointers. In particular, *dynamic memory allocation* allows creating, using, and disposing of memory cells in a very flexible way, and without defining new variables. It is often associated to pointers in order to build data structures of dynamic size. Some languages like C require programmers to explicitly carry out both allocation and deallocation. Other languages like Java or ML let programmers allocate new objects but provide automatic memory deallocation. In all cases, reasoning over programs with dynamic memory allocation is not trivial since the memory footprint of the program, namely, the set of memory cells that it may read or write, is not known statically and varies depending on the execution traces. In static analysis terms, it means that the analysis should infer not only content properties but also the memory footprint. By contrast, section 8.3.3 assumes a

fixed, finite footprint. In the following, we study the analysis of programs that construct and use dynamic data structures.

Extension of the Language Syntax and Semantics We extend the language of chapter 3. The main change to the target language that is required to support dynamic memory allocation is to let the set of memory cells evolve during the execution of programs. So far, we have let it be fixed by a pre-defined set of variables X . In section 8.3.3, we already introduced a set of addresses A , although we let it be equal to the set of addresses of program variables. We now need to relax this constraint. Instead, we require A to contain the addresses of the variables, as well as an infinite but countable set of heap addresses. Then a memory state is not only a function from variables to values but also a partial function from addresses (including the addresses of the program variables) to values:

$$M = A \rightarrow \text{part } V$$

Note that a memory state is only a partial function: first, the addresses that are used vary during the execution; second, this set is not expected to ever cover all elements of A (which is assumed infinite). Furthermore, we assume in this subsection that dynamically allocated memory cells are grouped in blocks made of series of contiguous fields (this would actually require a slightly more complex definition of M , although we do not formalize it here).

Due to this modification to the definition of memory states, the formal semantics of section 3.1 would need be thoroughly updated. The updates are quite systematic, though: a variable read or write now relates to an arbitrary cell and not necessarily to a variable.

Besides this change and the addition of the pointer operations of section 8.3.3, we need to add specific operations to allocate or free memory and access block fields. The added operations are slightly more realistic than those in section 8.1:

- **Allocation of a block:** When n is an integer, the statement $p := \mathbf{alloc}(n)$ creates a fresh block made of n fields and stores its address into the pointer variable p . Thus, it makes the set of valid addresses in the current memory state grow. For the sake of simplicity, we assume the values of the fresh block fields are chosen non-deterministically at run-time.
- **Deallocation of a block:** The statement $\mathbf{free}(p)$ frees the block that p points to.

- **Block field access:** If p is a pointer variable that points to a memory block, and if i is a valid field of that block (i.e., if it is between 0 and the length of that block), then $p->i$ denotes the field of index i in the block; this expression can be used either to read the value of a field or to write into a field.

In the following, we consider a few programs that rely on dynamic memory allocation to construct *singly linked list* structures, similar to that shown in figure 8.16(a). A singly linked list consists of a chain of blocks that have two fields: the first field stores either the address of the next element or the special **0x0** pointer value that denotes the end of the list; the second field stores another value (e.g., a scalar). In figure 8.16(a), we let dynamic blocks be denoted with light gray rectangles. These structures are useful to store sets of records the number of which is not known prior to the execution of the program.

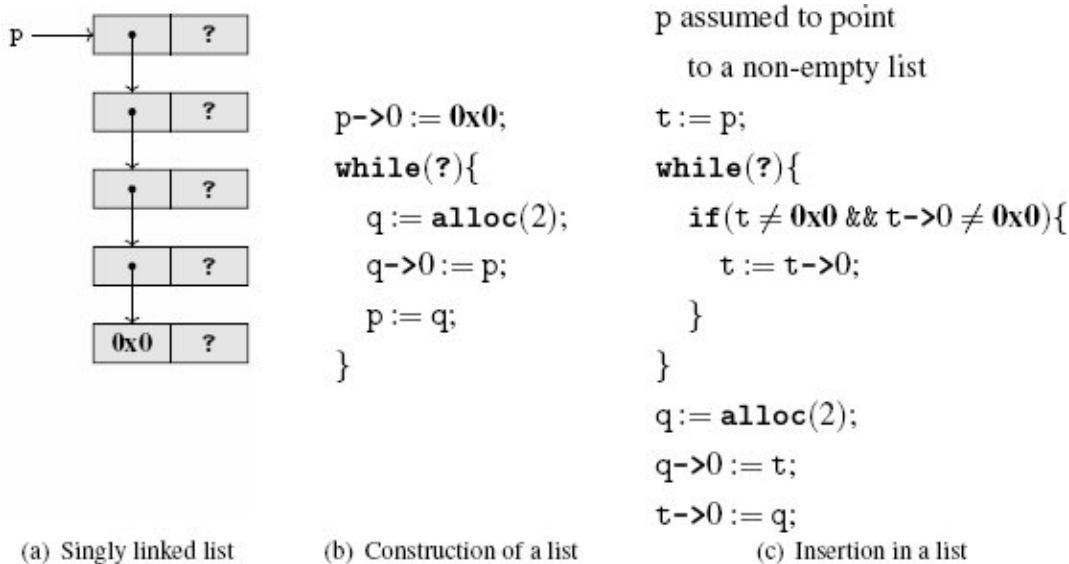


Figure 8.16

Programs with dynamic memory allocation

Figure 8.16 also presents a couple of list-manipulating programs. For brevity, we show only operations on the fields that store list pointer links. The program of figure 8.16(b) constructs a list of any possible length. To do this, it allocates one block per element. In a real-world implementation, the length of

the list would depend on some program input, and the data fields would also be modified. The program of figure 8.16(c) inserts a new element in a list. In a complete program, it would determine the position based on the contents of each block (e.g., so as to preserve sortedness).

Semantic Properties Related to Dynamic Heap Allocation The properties of interest mentioned in section 8.3.3 are still relevant here. In particular, memory safety is a major concern, as the dereference of a null or invalid pointer will cause a run-time error, and since dynamic structures typically require complex series of pointer updates. Additionally, when a block is deallocated, all pointers to it become dangling, so dereferencing them will cause an error. Thus, memory safety is even harder to ensure in presence of dynamic memory allocation.

However, an important class of semantic properties become relevant here: dynamic structures such as the list shown in figure 8.16(a) should also satisfy structural invariants, which are easily broken. As an example, the singly linked list structures introduced above consist of acyclic sequences of blocks chained to one another, and such that the last block is marked with a null pointer field. Breaking this property may cause either run-time errors (e.g., a dangling pointer dereference if a block is deallocated before the previous link is updated) or non-termination (e.g., a function that traverses the list and stops when it encounters the null pointer will not terminate when applied to a cyclic structure). Therefore, it is often interesting to verify the correctness of the *shape* of the data structures. The notion of shape property refers to the compliance with a global structural property. For instance, we may want to verify that the program of figure 8.16(b) constructs well-formed lists and that the program of figure 8.16(c) never breaks structural correctness.

Last, for the same reasons as noted in section 8.3.3, pointer or shape information may be required to reason over other aspects of programs, such as numeric computations, if these numeric computations depend on the memory layout. In that case, it is necessary to combine (e.g., using the notion of product presented in section 5.1.2) one of the abstractions that we present below with a numeric abstract domain. For brevity, we do not comment on this last family of target properties and focus on analyses for programs with pointers and dynamic allocation to establish either memory safety or shape properties.

Abstractions Based on Allocation Sites The pointer abstractions shown in section 8.3.3 assume a finite set of addresses; thus, they do not directly apply here. Indeed, they are defined under the assumption that the set of possible addresses

is known and finite (e.g., it is exactly the set of the variables of the program). However, dynamic memory allocation generates fresh blocks in a way that depends on the execution of the program.

To circumvent this limitation, a common approach groups memory blocks that are allocated dynamically into a finite set of abstract blocks. For instance, such an abstract block can be characterized by the location in the source code at which its elements have been allocated. This abstraction is called *allocation site* based, which also is the abstraction used in example 8.4 of section 8.1. Then an abstract state consists of a finite set of abstract addresses, including some *summary addresses*, which account for a possibly unbounded number of concrete addresses. Indeed, while the content of each variable is still described precisely, the content of a block that is dynamically allocated at a point visited several times is summarized. The notion of summarization employed here is quite similar to the one introduced in section 8.3.1. Such summaries usually incur a loss of precision compared to abstract addresses that describe a single concrete cell. Indeed, the weak update phenomenon described in sections 8.1 and 8.3.1 also occurs here.

As an example, we describe the results of an analysis based on allocation site abstraction for the program of figure 8.16(b). Both pointer variables p and q are described by a single abstract address. Moreover, all list blocks are allocated at the same point (in the first line of the loop body); therefore, they are all described by a single abstract address, which we denote by a . Let us now assume a points-to set abstraction of pointer values. Then the following abstract state describes all memories observed at the loop head:

$$\left\{ \begin{array}{l} p \mapsto \{0x0, a\} \\ q \mapsto \{0x?, a\} \\ a \rightarrow 0 \mapsto \{0x0, a\} \end{array} \right.$$

We note that q may either be dangling (due to being still undefined) or point to some location described by the summary address a . Similarly the content of p is either the null pointer or an address described by a . Last, all pointer fields in the list also store either the null pointer or the address of another element of the list. However, we observe that this abstract state is quite coarse and does not characterize a well-formed singly linked list structure: while singly linked lists can be described by these constraints, many other (incorrect) configurations would also be described by it, including cyclic structures, or structures that are

not connected. For this reason, we discuss shape abstractions later, which address this issue. On the other hand, the information this abstraction allows computing is often sufficiently precise to verify basic memory safety (absence of pointer dereference errors).

Regarding to the static analysis algorithms, the main difference from section 8.3.3 is that a summary abstract address may account for many concrete addresses; thus, an update to a field of a dynamically allocated block will require a weak update, which accounts both for the case where a memory cell is updated and for the case where it is not updated.

Abstraction of the Shapes of Data Structures *Shape abstractions* aim at preserving information about the shape or form of data structures, by summarizing regions in a dynamic way (i.e., that depends on the program points) and by attaching content information to summaries. For instance, it can express that a region stores a well-formed singly linked list as a whole. Compared to the basic allocation site abstraction mentioned above, this approach enables more precise program reasoning, although the abstract predicates and analysis algorithms are significantly more complex.

To understand the definition of summary structural predicates, we focus on the case of singly linked list segments: a segment is a contiguous fragment of a list. Essentially, a well-formed *singly linked list segment* consists either of an empty region (then the first and last addresses are equal) or of a first list element with a pointer to the tail of the segment. This intuition supports an inductive definition of a summary predicate, as shown in the graphical view of figure 8.17(a): the rounded corner boxes denote **sll** (“singly linked list”) predicates, and can be defined recursively, using basic list blocks. Essentially, the **sll** predicate describes any empty or non-empty list segment. We do not give a mathematical definition for the **sll** predicates and use the high-level graphical representation shown in figure 8.17(a).

Based on this graphical representation, we show two abstract states in figure 8.17(b). The abstract state on the top describes memory states with a pointer variable **p** that points to a well-formed singly linked list of length at least one. The code shown in figure 8.16(b) constructs well-formed lists. Thus, this abstract state correctly describes all these states. The abstract state in the bottom describes memory states with two pointer variables **p** and **t**, and such that **p** points to a well-formed singly linked list and **t** points to one of the elements of this list.

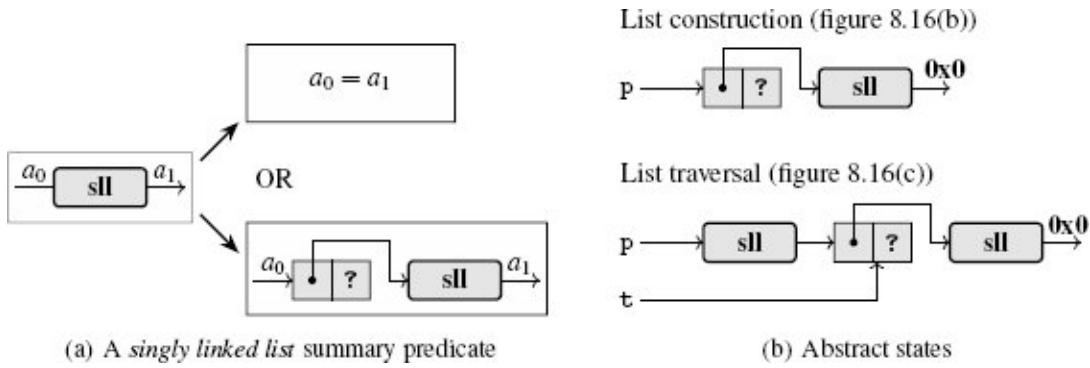


Figure 8.17

Shape abstraction based on the summarization of list segments

This abstraction relies on the summary predicates, in conjunction with a more conventional pointer abstraction. There exists several ways to describe these summaries in the literature, such as reachability recursive predicates in three-valued logic [99], and separation logic [95] with inductive predicates [35, 9, 19].

Corresponding analysis algorithms deal with summary predicates by going back and forth between summarized forms and more concrete forms (as shown in figure 8.17(a)) as in the following examples.

- The analysis of an update into a field of a list element that is part of a summary predicate requires unfolding this summary into a disjunction of cases according to the schema shown in figure 8.17(a), so as to *materialize* all the memory cells that are read or written; for instance, this is necessary when analyzing the list traversal of figure 8.16(c).
- The analysis of a loop needs to generalize memory shapes (with a specialized widening algorithm) that synthesizes summaries, following the schema of figure 8.17(a); for example, the analysis of a loop that constructs a list (as in figure 8.16(b)) requires such a generalization to produce the abstract state shown at the top of figure 8.17(b).

These two algorithms are illustrated in figure 8.18. First of all, let us consider the abstract state in the left-hand side of the picture. The variable p is known to store a pointer to a well-formed singly linked list and is not equal to the null

pointer. Therefore, it is possible to materialize the memory cell that is pointed to by p ; indeed, a direct case analysis over **sll** would produce two disjuncts (either the list is empty, or it is not empty), and the fact that p is a non-null pointer rules out the case of the empty list. As a consequence, materialization produces the abstract state in the right-hand side. Conversely, the abstract state in the right-hand side can be replaced with the abstract state in the left-hand side. This operation makes the structure of the list pointed to by p more abstract; thus, it acts as a generalization analysis step.

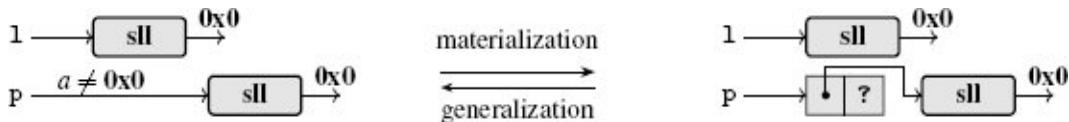


Figure 8.18

Materialization and generalization of shape abstract states

These algorithms allow computing invariants such as the abstract states shown in figure 8.17(b).

To give a general view of the main steps of such a shape analysis, we study the analysis of a list reversal program shown in figure 8.19. The program is assumed to start in a state such that l points to a well-formed singly linked list. Moreover, it traverses this list and updates the pointers between the elements so that their order is eventually reversed. For the sake of clarity, line numbers in the source code are indicated. Local abstract invariants that are computed at the end of the analysis are shown at each source code location in the figure. Each of these abstract states contains summaries that account for unbounded numbers of list elements. Materialization occurs at line 3, inside the body of the loop: at this point, the summary pointed to by p needs to be unfolded, as shown in figure 8.17(a). The rest of the analysis of the loop body is actually straightforward, as each step boils down to the update of a pointer that is already precisely exposed. As usual, the computation of the abstract state at loop head requires widening iterations (which are not shown). This operation generalizes the abstract states before the loop and at the end of the loop body:

- All information about n is dropped (as it is not initialized before the loop).

- The information about l is weakened into an **sll** predicate (l is the null pointer before the loop and points to a non-empty list at the end of the loop body; thus, it points to some sort of well-formed linked list in both cases).

As a side note, we remark that the notion of summarizing abstract predicates and the algorithms to navigate between the summary predicates and more precise expressions have some similarities with the array-partitioning abstraction of section 8.3.1. Indeed, both families of analyses split summaries so as to perform updates, and synthesize new summaries as part of the loop invariant computation process based on widening.

8.4 Abstraction for Control Flow Structures

This section focuses on language extensions that make the control flow more complex. Essentially, these require us to extend the notion of control states (chapter 3) and to enhance static analyses accordingly.

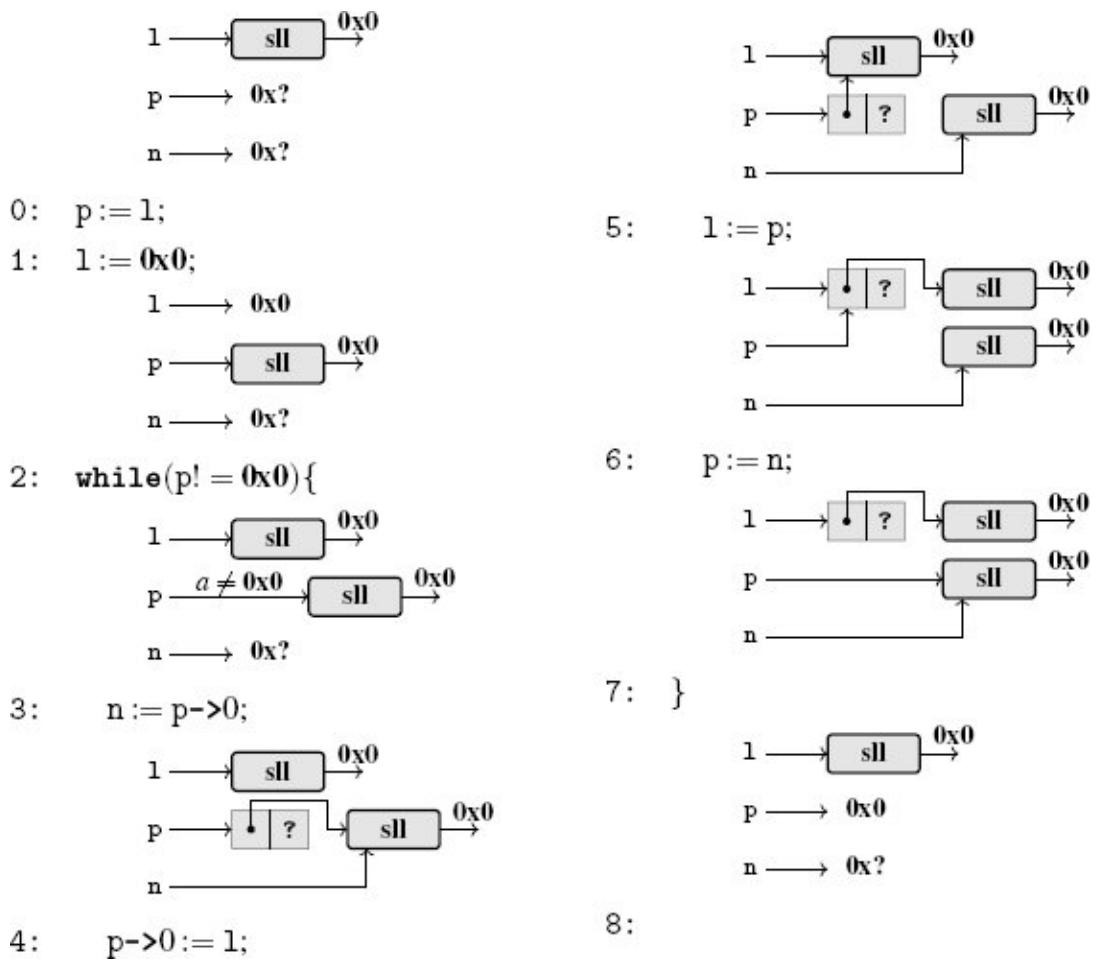


Figure 8.19

Shape analysis of a list reversal, with the assumption that `l` initially points to a well-formed singly linked list

8.4.1 Functions and Procedures

Functions and procedures make code modular and allow splitting the implementation effort more easily and reasoning over different pieces of a same program in a separate manner. Therefore, they play a fundamental role in programming and are present in one form or another in all programming languages. On the other hand, they make the execution flow a lot more complex to understand as there may be infinitely many radically different ways to reach a given program point. The flows of data are also more intricate due to

the passing of values as arguments and to the references into the call stack. Furthermore, higher-order functions allow parameterizing a function by another function, which complicates control flow even more.

In the following, we discuss some of the main issues that are brought up by functions and procedures, as well as some of the most common abstractions and analysis techniques. As in the previous paragraphs, exhaustivity is out of our reach; thus, we focus on the most salient aspects. For instance, we deliberately leave out higher-order functions to better focus on the call stack abstraction.

Extension of the Language Syntax and Semantics We will consider a slightly different language from the one in section 8.2. As in the previous section, we first set up a minimal language for our study, by extending the basic language of chapter 3. Since functions with parameters and possible return values would require a significantly more complex syntax, we simply add basic procedures, and we assume the language supports both global and local variables.

In practice, this leads to a comparable level of expressiveness, but with lighter syntax since both parameters and return values can be eliminated using copies in global variables. Therefore, we consider only procedures without argument and without a return value.

A *procedure declaration* thus comprises only a finite set of local variables and a command (often a sequence of commands) that stands for the body of the procedure. We denote the declaration of a procedure p with local variables x_0, \dots, x_n and with body C by

```
proc p local[ $x_0, \dots, x_n$ ]{C}
```

The commands in the body of the procedure may access both its local variables and the global variables. For the sake of consistency, we also let the keyword **global** prefix the declarations of global variables in the following.

A *procedure call* command is simply defined by the name of the procedure to be called (in the following, we refer it as the *callee*). We denote a call to procedure p by

```
call p;
```

A program consists of a finite set of procedure declarations and of a *main body*, which is the equivalent of the main function of a C or Java program.

Figure 8.20 presents two programs with procedures. The program of figure 8.20(a) is quite contrived and mainly aims at showing how procedures complicate static analysis. Indeed, we observe that the behavior of function h depends on the caller procedure (f or g). The program of figure 8.20(b) is inspired by a general dichotomic search algorithm, and illustrates recursion.

We now define the semantics of procedures. The semantics presented below is lighter than that in section 8.2, as its only goal is to show relevant abstractions. To make the definition lighter, we set it up in two steps: first, we assume that procedures have no local variables, and we focus on the control part; second, we consider local variables. A control state describes the sequence of ongoing nested function calls. Each function call in this sequence is defined by the name of the procedure that is currently running, by the name of the procedure that called it, and by the program location at which this call occurred. Therefore, it consists of a *stack* of procedure names, also called *call stack*. The call stack is, in the semantics jargon, called *continuation* in section 8.2.

```
global[a,b]
```

```
proc h[]{
    if(a > 0){
        b := 0;
    }else{
        b := b + 1;
    }
}
```

```
proc f[]{
    a := 1;
    call h
}
proc g[]{
    a := -1;
    call h
}
call f;
call g;
```

(a) A few procedures

```
global[a,b]
```

```
proc p local[x]{
    input(x);
    if(x > 0){
        a := (a + b)/2;
    }else if(x < 0){
        b := (a + b)/2;
    }
    call p;
}
input(a);
input(b);
call p;
```

(b) A recursive search

Figure 8.20

Example programs with procedures

When the program starts, the call stack is empty. Then, whenever a procedure p is called, p is added on top of this stack, and when the end of the current procedure is reached, the execution continues at the program point defined by the procedure and call site at the top of the call stack. By definition, each local variable is created at the time of a call to a procedure and disappears when this procedure returns. In fact, when a procedure is called recursively (like p in figure 8.20(b)), several instances of a local variable may be created. Therefore, the aforementioned stack should also contain the local variables attached to each procedure call. To summarize, the call stack is a sequence of 3-tuples (p, l, m_p) , where

- p is a procedure name;
- l is the location at which it was called; and

- m_p is a partial memory state that defines the value of the local variables of p .

Such a 3-tuple is called an *activation record* (in the compiler jargon). Then a *state* should comprise not only the current program location and the current value of global variables, but also the call stack. Using this notion of state leads to a trivial extension of the semantics exposed in chapters 3 and 4.

Semantic Properties Related to Functions and Procedures Regardless of the target semantic property to verify, the existence of procedures requires an additional effort in the analysis design. Therefore, all the semantic properties mentioned in the previous chapters are relevant in our study of the analysis of programming languages with procedures. Due to this, we study in the following paragraphs general static analyses to over-approximate the set of reachable states of a program with procedures. However, we can also mention several semantic properties of interest, which are specific to programs with procedures.

First, the call stack (the *continuation*) that is now part of program states also needs to be stored in memory, and most implementations provide a fixed amount of stack space: when this space is exhausted, the execution may crash (with a stack overflow error), or adjacent memory cells may be corrupted. Thus, it is often useful to compute statically an over-approximation of the stack space that a program may consume. In practice, such analysis of the stack requires a reachable-state static analysis that reasons over the low-level structure of activation records at the assembly level [68]. If we design this stack analysis in section 8.2.2, the design boils down to the definition of the abstract continuation domain $K^\#$.

Second, it is often interesting to compute and verify that the effect of a function matches some input-output specification. As an example, one may want to prove that any call to the procedure p defined in figure 8.20(b) either does not return or returns in a state where the new values of a and b are in the range $[\min(a,b),\max(a,b)]$ of the old values of these variables. We call this a *relational property*, as it relates input and output states. Another example is the typing of functional languages. The type of a function characterizes both the arguments that it may take and the results that it may produce, and possibly the relations between both (in the case of polymorphic types). Types are very valuable to developers (they help catch many kinds of programming errors) and compilers (they guide the choice of data representation formats). We do not

study these families of properties in this chapter (typing is discussed in chapter 10).

Fully Context-Sensitive Abstraction As we remarked above, the main change in the definition of the language is the addition of the call stack. A first technique to analyze programs with procedures simply keeps all the stack information fully precise. In other words, it does not abstract the call stack at all. We call this approach a *fully context-sensitive abstraction*, as it keeps all information about call contexts. It can be viewed as an application of the sensitive abstraction method described in section 5.1.3, where the analysis maps each possible context to abstract information that can be observed in that context. It is also known as the *call string* approach [100] since it relies on call strings to describe contexts.

As an example, a fully context-sensitive analysis of the program shown in figure 8.20(a) should distinguish the following contexts:

- the empty context (when the program executes the main command);
- the context corresponding to a call to f (there is only one call site to f , located in the main command);
- the context corresponding to a call to g (similar to the previous one);
- the context of a call to h from f (so that the call stack contains triples corresponding to f and h in that order); and
- the context of a call to h from g (so that the call stack contains triples corresponding to g and h in that order).

As a consequence, such an analysis keeps precise information about each call to function h . This is the main advantage of fully context-sensitive abstractions.

On the other hand, keeping information about each possible context may be costly, or even impossible. In the settings of section 8.2.2, the full context sensitivity corresponds to not abstracting the instance domain $\wp(I)$. In the case of the recursive procedure shown in figure 8.20(b), there are infinitely many call contexts since a call context boils down to a sequence of calls to p ; thus, it is characterized by the recursion depth. Therefore, a fully context-sensitive abstraction that distinguishes all contexts requires abstracting elements that cannot be represented. Even without recursive functions, fully context-sensitive abstractions may be too costly due to a too high number of contexts, which means that the analysis will require too much memory or time to explore all possible contexts.

From the analysis implementation point of view, this approach is quite simple to set up, as it assumes that a control state is a pair of a call context and a location in the program text: whenever a call is encountered, the analysis should extend the call context with the new call, and when a procedure return point is reached, the analysis should pop the topmost element from the call context and use it to continue the analysis of the caller. These analysis operations are very similar to the way concrete program executions progress.

Context-Insensitive Abstraction Another approach completely abstracts contexts away. It is called *context-insensitive abstraction*. This abstraction essentially collects all states observed in a same function, before applying any other abstractions (e.g., the value abstractions described in chapter 3). In the setting of section 8.2.2, a context-insensitive abstraction is obtained by abstracting away the instance domain $\wp(I)$.

If we consider the example code shown in figure 8.20(a), then the context-insensitive abstraction does not distinguish between the states that are reached when h is called from f and the states that are reached when h is called from g . This implies that such states will not be distinguished and that overall analysis results are likely to be much less precise than with the fully context-sensitive abstraction presented in the previous paragraph. Another comparison between the context-sensitive abstraction and the context-insensitive abstraction can be found in example 8.9 (see figure 8.8).

The obvious advantage of context-insensitive abstraction is a lower cost than the context-sensitive approach. Even when a program has recursive procedures, the context-insensitive abstraction can produce rather compact results since it yields only one piece of abstract information per procedure in the program. As an example, a context-insensitive abstraction of the executions of the program of figure 8.20(b) will produce a single abstract state at the entry into procedure p . On the other hand, we should expect analyses based on context-insensitive abstraction to produce less precise results than a fully context-sensitive one.

The easiest way to view a context-insensitive analysis is to start with an analysis that iterates over a control flow graph with additional edges for procedure calls and returns [94]. For each call command, we have two edges: one from the point before the call site to the procedure entry and the other from the procedure exit to the point right after the call command. Note that in general the exit point of a procedure has several outgoing edges; as an

example, in the case of the program of figure 8.20(a), there are two edges from the last control point of `h` (one per call site).

Partially Context-Sensitive Abstraction Very often the fully context-sensitive abstraction is too heavy and the context-insensitive one leads to results that are too imprecise. To cope with such cases, a whole range of intermediate abstractions can be used that retain some but not all of the context information. We call them *partially context-insensitive* since they preserve only part of the context information.

The most common way to achieve such a partially context-insensitive abstraction preserves only the topmost k activation records in addition to the topmost one, where k is an integer bound fixed before the analysis; moreover, it discards the rest of the call stack altogether. This approach is generally called k -CFA [101]. For instance, let us assume that $k = 1$, and consider the two examples of figure 8.20. We first discuss the program of figure 8.20(a). Using partially context-sensitive abstract up to depth 1, the analysis distinguishes the states that reach `h` after a call from `f` from those that reach `h` after a call from `g`. It thus remedies the imprecision observed with the context-insensitive abstraction. In the case of the code of figure 8.20(b), recursion prevents fully context-sensitive abstraction from producing a finite representation. However, with partially context-sensitive abstraction up to depth 1, not only this issue does not occur, but also two sorts of calls to procedure `p` are differentiated: on one hand, the call from the main command body, and on the other hand, the recursive calls that stem from `p` itself. These two examples show how partial context sensitivity can provide decent precision while limiting the cost. Because of this, many static analysis tools let the user control to what extent the analysis should be sensitive to contexts.

In the analysis implementation point of view, partial context sensitivity can easily be described as an iteration over a graph, where nodes describe abstract contexts (e.g., contexts up to depth k) and a program location.

Abstraction of the Call Stack Contents The previous paragraphs focused on the abstraction of the call stack from the context point of view, as it is often the most important choice to make when reasoning over programs with procedures. However, the contents of the stack also need to be abstracted. From the memory point of view, the call stack is a stack data structure. Thus, the techniques discussed in section 8.3 apply. In particular, the summarization and shape abstraction approaches presented in section 8.3.4 can be used to describe

the contents of local variables stored in the call stack. When using a fully sensitive abstraction of call contexts, another option is to keep a fully precise representation of the layout of the call stack; for instance, we can apply a non-relational abstraction of the value of program variables and associate an abstract value to each local variable.

Relational Abstraction, and Procedure Summaries The previous paragraphs focused on the abstraction of the states that can be observed at various points during the execution of programs with procedures. Another approach aims at computing the *effect* of each procedure out of any context. For instance, a procedure that increments a global variable has the effect of modifying the value of that variable. This method is also called the *functional approach* [100], as it relies on abstractions of the functional behavior of procedures. It has been implemented in many analyses [62, 92].

The abstraction needed to represent the effect of procedures depends on what the form of program states and on the nature of the effect properties that are researched (numeric, Boolean, or related to pointers). As mentioned above, function types are one instance. We can also define other forms of summaries that characterize the relationship between values before and after a procedure is run. For instance, the procedure of figure 8.20(b) satisfies the following relation, where we note a^{in} (resp., a^{out}) for the value of a before (resp., after) the execution of the body of p :

$$\left(\begin{array}{l} a^{in} \leq b^{in} \\ \wedge \quad a^{in} \leq a^{out} \leq b^{out} \leq b^{in} \end{array} \right) \vee \left(\begin{array}{l} a^{in} \geq b^{in} \\ \wedge \quad a^{in} \geq a^{out} \geq b^{out} \geq b^{in} \end{array} \right)$$

This summary consists of a disjunctions of constraints that can be described using a relational abstract domain, such as convex polyhedra (definition 3.9), and that convey the fact that p shrinks the range between a and b .

Such summaries are interesting for several reasons. First, they provide information about what a procedure or function does, independently from its calling context. This implies that they are well suited to verify functional properties of functions in a library. Second, procedure summaries may also help accelerate a static analysis that computes an over-approximation for reachable states, by avoiding the need to analyze a procedure in many call contexts [17]; indeed, once an abstract summary is available, the analysis can derive an abstract post-condition for a call from an abstract pre-condition. While computing a summary may be expensive, this cost can be amortized

over a great number of calls. For additional details on such *modular analysis* approaches, we refer the reader to section 5.4, where we cover this approach in a general setting.

8.4.2 Parallelism

Hardware and software architectures are increasingly parallel. Modern processors have an increasing number of cores and often support hyperthreading. Software often combines several tasks that should be performed in concurrent threads; for instance, an operating system needs to manage many resources (memory, access storage devices and networks, etc.) and to run in the same time as user processes.

Unfortunately, concurrency complicates significantly any reasoning about program semantics, and static analysis in particular. First, concurrency makes it harder to describe how program executions progress since the executions of the parallel components are *interleaved* [72]. Second, parallel programs may suffer defects that are irrelevant in the case of sequential programs and thus motivate verification tools. As an example, concurrent access to resources and variables may cause undesirable *races*. Synchronization mechanisms that are supposed to prevent races may cause parallel programs to reach a *deadlock* state, which means that the execution cannot progress any further. We discuss these issues in this subsection.

Hardware-level concurrency brings its own set of issues with the complexity induced by weak memory models, where actual read/write operations may be reordered due to the cache hierarchy. We do not study these issues in this book.

This section provides a high-level description of the main issues related to parallel programs and sketches a few important analysis techniques.

Extension of the Language Syntax and Semantics To include parallelism into our small language, we add a command that executes two commands asynchronously. We note the command that runs C_0 and C_1 in parallel as follows:

$$\{C_0 \parallel C_1\}$$

Using the common terminology, we call *thread commands* (or *threads*) the commands C_0 and C_1 . Instead of augmenting our simple language with locks or other synchronization mechanisms, we let global variables simulate them. Figure 8.21 displays two parallel programs. Each of these programs is

composed of initialization assignments followed by the parallel execution of two thread commands. For clarity, the definitions of the thread commands are shown separately. Besides the parallel composition, all the constructions used here are standard. Although these programs are contrived, they show several key issues raised by the analysis of parallel programs.

Because of the parallel execution of commands, the format of the semantics needs to be amended. Indeed, a program state should capture the configuration of the machine at a given point in the execution of a program. Thus, when several thread commands are running in parallel, a state should specify the status of each of these threads. As a consequence, a state should be made of a set L of program points in different thread commands, and a memory state m . Furthermore, we assume that a program execution step describes either a step in any of the active thread commands, or the beginning of a parallel section:

| | |
|---|--|
| $l_0 : \quad x := 0;$ | $l_0 : \quad x := 0;$ |
| $l_1 : \quad y := 0;$ | $l_1 : \quad \{C_0 \parallel C_1\}$ |
| $l_2 : \quad \{C_0 \parallel C_1\}$ | where $C_0 : \quad l_0^0 : \quad y := 1;$ |
| | $l_1^0 : \quad \text{while}(x < 1)\{\}$ |
| where $C_0 : \quad l_0^0 : \quad \text{while}(x = 0)\{\}$ | $l_2^0 : \quad z := 2;$ |
| $l_1^0 : \quad y := 1$ | $l_3^0 : \quad \text{if}(y > 0)\{x := 2\}$ |
| $l_2^0 :$ | $l_4^0 :$ |
| and $C_1 : \quad l_0^1 : \quad \text{while}(y = 0)\{\}$ | and $C_1 : \quad l_0^1 : \quad y := 0;$ |
| $l_1^1 : \quad x := 1$ | $l_1^1 : \quad z := 1;$ |
| $l_2^1 :$ | $l_2^1 : \quad x := 1;$ |
| | $l_3^1 :$ |

Figure 8.21

Examples of parallel programs

- If $(l, m) \hookrightarrow (l', m')$ (where \hookrightarrow is the transition relation defined in chapter 4), then $(L \uplus \{l\}, m) \hookrightarrow (L \uplus \{l'\}, m')$.

- If l is the control state preceding a parallel command $\{C_0 \parallel C_1\}$, and l_0^0, l_0^1 are the entry control states of C_0, C_1 , then $(L \uplus \{l\}, m) \hookrightarrow (L \uplus \{l_0^0, l_0^1\}, m)$.

This execution model assume that steps described by \hookrightarrow , are atomic and that they cannot be decomposed into smaller steps, which can be executed in parallel. As an example, figure 8.22 presents all the nonlooping transition steps of the parallel program of figure 8.21(b).

Semantic Properties Related to Parallelism First, we discuss several semantic properties specific to parallel programs, which are particularly relevant in practice. As mentioned above, reaching a deadlock state is a very serious issue, as it means that none of the parallel components will progress from that point. As an example, an obvious case of deadlock is shown in figure 8.21(a), since both variables x, y are initially equal to 0, which prevents both C_0 and C_1 from progressing. Searching for the absence of deadlocks boils down to a reachability problem, where the goal is to prove that some undesirable configurations cannot be reached. In the case of the program of figure 8.21(a), blocking configurations are of the form $(\{l_0^0, l_0^1\}, m)$, where $m(x) \leq 0$ and $m(y) \leq 0$. Such a state is obviously reachable, as both variables are initially set to 0.

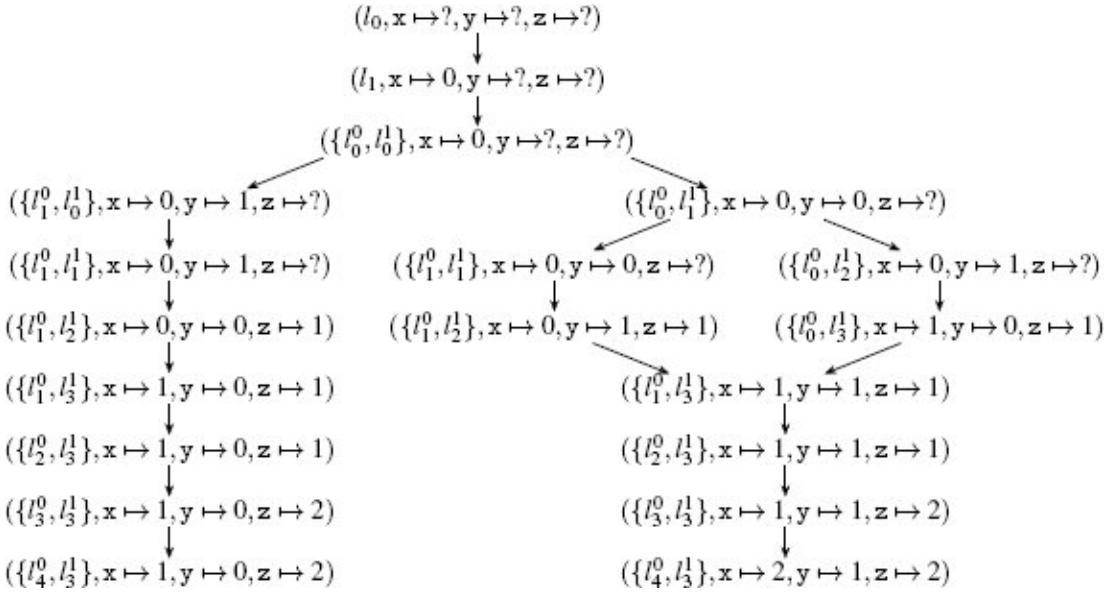


Figure 8.22

Parallel executions: nonlooping execution steps in the program of figure 8.21(b) (for clarity, looping transitions are omitted)

Data races correspond to pairs of read or write actions to a same location, which may occur in any order. They are sources of non-determinism, and while some data races are admissible (or even desirable), others could cause unwanted or erratic behaviors. We can identify two interesting static analysis problems related to data races: one focuses on computing a super-set of the possible data races (either to prove that there are none, or to verify that they are all admissible with respect to the specification), and the second attempts only to compute a superset of the possible effects of races on the semantics of programs (e.g., to determine if some numerical constraints hold). As an example, the program of figure 8.21(b) has several races: first, the two assignments to y may happen in any order; second, the same goes for the assignments to x in C_0 and C_1 . As a consequence, the final values of variables x and y depend on the execution order. On the other hand, the assignments to z are unambiguously sequentialized since C_0 cannot exit the loop until C_1 updates x to 1. One may be interested in proving that the final value of z is indeed 2, which implicitly requires reasoning over races.

Fairness is a semantic property that is most relevant for a specific subclass of parallel programs. The most common example is the case of a scheduler, which is a thread run in parallel with other thread commands, and which should manage the access to a resource (computation time, memory, output device, etc.) by the other thread commands. Intuitively, a scheduler is said to be fair if it will not cause any program to starve on the resource that it manages by never letting the program access the resource.

Last, all the semantic properties of interest for sequential programs are also relevant to parallel programs. For instance, distributed computing systems (e.g., distributed databases) utilize sophisticated data structures; thus, all the structures and related properties mentioned in section 8.3 are relevant here. However, in addition to the problems related to the nature of the data structure, we also have to face issues related to concurrent accesses, including race conditions. To verify such properties, we need to address the second analysis problem mentioned in the paragraph on data races, namely, the sound over-approximation of the effect of all the possible races.

Abstractions and Analysis Techniques for Parallelism To discuss the main issues related to the analysis of parallel programs, we study the extension of static analyses aimed at numerical properties such as those presented in chapters 3 and 4. We assume an abstraction of values is fixed (e.g., based on the domain of intervals introduced in example 3.7), and we study two important families of analysis techniques for parallel programs:

- an approach based on *global iteration* over all threads; and
- a technique that composes and iterates simpler analyses that are *local to specific threads*.

Global Iteration over All Threads When considering parallel programs, a first major issue stems from the fact that a state in a parallel program comprises a state for each of the components that are running in parallel. This implies that adapting the iteration techniques presented in chapter 3 (iteration over a syntax tree) and chapter 4 (iteration over a control flow graph) requires using the extended semantics mentioned above, so as to account for *all* possible parallel interleavings. In particular, a worklist algorithm as shown in chapter 4 can be applied immediately to the extended relation \hookrightarrow . As an example, in the case of figure 8.21(b), this technique can discover that it is not possible to reach the exit location of C_0 before reaching the exit location of C_1 , and also that the final value of z is necessarily 2. While this approach retains a lot of

information about the program history, it is also very costly. Indeed, if the program is composed of only two commands run in parallel, the number of control states is proportional to the product of the sizes of these commands, so that even the estimation of the control flow is quadratic, hence not tractable. Obviously, the situation is even worse when considering a program made of many components ran in parallel. Moreover, this approach does not really exploit the modularity induced by the construction of programs from parallel threads, since it analyzes all the threads simultaneously.

Iteration of Thread Local Analyses An alternative approach is to analyze threads separately. However, threads may interact, so this approach requires information about the shared variables, which may be read or written in distinct threads [87]. This is similar to rely-guarantee proof methods [64, 84] for parallel programs, where precise information on interactions needs to be set up before reasoning over each separate thread. It is possible to let a static analysis automate this proof method, by analyzing each thread separately and letting the analysis of a thread compute an over-approximation of the write operations over variables that other threads may read or write [72, 83]. In the case of the program shown in figure 8.21(b), such shared variables include x , y , and z . When the analysis of C_0 reveals a possible write into variable x , the other threads need to be reanalyzed under the assumption that x may have changed at any time. In turn, their analysis may reveal new writes into shared variables, causing other threads (including C_0) to be reanalyzed. When the global interactions that are discovered are stable, and the analysis of each thread is stable, a sound over-approximation of the executions of the whole program is produced. Let us unfold the first steps in the analysis of the program of figure 8.21(b):

1. Initially, x is equal to 0.
2. The analysis of C_0 shows that only its first line may run (since $x = 0$), so that y becomes equal to 1.
3. The analysis of C_1 shows that y is set to 0 *before or after* C_0 sets it to 1, so we may assume that $y \in [0, 1]$; moreover, x is set to 1.
4. At this stage, C_0 needs to be analyzed again, as C_1 may have modified x and y ; this time, the analysis progresses beyond the loop and shows possible new values for z and x .
5. At this point, it is necessary to analyze C_1 again.

To summarize, this approach boils down to iterating a global computation over all the thread commands, each step of which consists of a local post-fixpoint computation over each thread command. Such an analysis is generally much faster than the approach based on tuples of program location, but it may also be less precise at first, as it may fail to take into account that some sequences of actions are impossible.

There are several ways to make the approach based on separate analyses of each threads more precise. One technique is to use a relational domain (section 3.2.3) for the approximation of the values that the shared variables may take. This allows indirectly ruling out certain control states. For instance, in the program of figure 8.21(b), the assignment $z := 2$ may be executed only in states where x is already equal to 1, which means that the thread command C_1 has reached its exit point; therefore, it allows showing that when the execution of the parallel composition terminates, z is equal to 2.

9 Classes of Semantic Properties and Verification by Static Analysis

Goal of This Chapter In section 1.3.1, we introduced several families of semantic properties that are interesting targets for verification, including safety, liveness, and information flow properties. In the subsequent chapters, we focused on state properties, even though many other kinds of semantic properties are important in practice. Indeed, state properties not only encompass many semantic properties of great interest but also provide the most simple starting point for the definition of static analyses. The goal of this chapter is to discuss techniques and abstractions dedicated to other semantic properties. Unlike previous chapters, we do not detail the definition of the abstractions; instead, we focus on the underlying intuitions and provide bibliographic references to the interested reader. In particular, we carefully consider *what* needs to be abstracted (sets of states, executions, or others) and how sound abstractions can be derived, while still reusing the basic techniques presented more thoroughly in the previous chapters.

Recommended Reading: [S] (Master), [D] and [U] (2nd reading, depending on need) Engineers and students may skip this chapter on first reading and revisit it when dealing with the verification of classes of properties that are not covered in the previous chapters. Teachers will also find in this chapter examples to illustrate in classes and practical sessions.

Chapter Outline In this chapter, we consider two families of properties. We discuss trace properties in section 9.1. These include not only safety properties but also liveness properties (as explained in section 1.3.1). Then section 9.2 presents a few properties based on information flow, including some classes of security properties. In each case, we provide examples of properties and of abstractions that allow tackling them.

9.1 Trace Properties

We call *trace property* a semantic property P that can be defined by the data of a set of program execution traces (or, for short, traces), which are exactly the executions that satisfy P . As an example, the absence of run-time errors can be described by the set T_{Ok} of executions that do not terminate in an error state. A program p is free of run-time errors if and only if the set $\llbracket p \rrbracket$ of all its executions traces as defined in its operational semantics (chapter 2) is included in T_{Ok} :

$$\llbracket p \rrbracket \subseteq T_{\text{Ok}}$$

By contrast, a *state property* can be defined a set of states S that are considered admissible, and holds for a program p if and only if the reachable states of p all belong to S .

As an example of trace property, let us consider a program that is supposed to sort in place an array of scalars (e.g., using an array-based implementation of merge sort or insertion sort). A correct implementation (in the sense of *total correctness*) of such a sorting algorithm should meet the conjunction of several basic properties:

- it should not fail with a run-time error;
- it should terminate;
- it should return a sorted array; and
- it should return an array that contains the same elements of its argument, with the same multiplicity (if there are duplicates).

An execution that violates this correctness specification is a trace that ends in an error state or that reaches the program exit with an array that is not sorted, or that does not contain the same elements as in the beginning, or that does not terminate. Thus, this total correctness property is a trace property.

As a program p satisfies a trace property T if and only if $\llbracket p \rrbracket \subseteq T$, trace properties have a few interesting monotonicity features:

1. If two programs p_0 and p_1 are such that p_1 satisfies a trace property T and $\llbracket p \rrbracket \subseteq \llbracket p_1 \rrbracket$, or, in other words, if p_1 has more behaviors than p_0 , then p_0 also satisfies trace property T ;
2. If a program p satisfies a trace property T_0 , and if T_1 is a second trace property such that $T_0 \subseteq T_1$, then p also satisfies T_1 ; in other

words, this means that T_0 is logically stronger than T_0 .

9.1.1 Safety

As observed in chapter 1, *safety properties* essentially state that some given behavior observable in finite time never occurs. Safety properties actually account for a large part of the semantic properties that programmers care for, and we have studied several important examples throughout the previous chapters.

A Few Families of Safety Properties As a first example, we have remarked previously that the absence of run-time errors is a safety property. In fact, any *state property* (i.e., semantic property that asserts that some set of states should not be reachable) is a safety property. The absence of failure due to a run-time error is a state property, hence a safety property. In the case of the aforementioned array-sorting program, reaching the exit point of the program with an array that is not sorted can also be described by a set of states the programmer expects not to observe. Thus, the fact that the program should always produce a sorted array is a state property, hence a safety property.

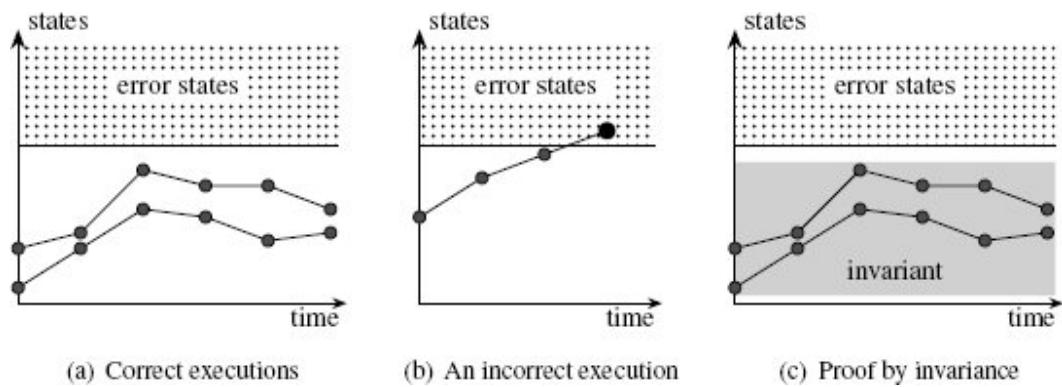


Figure 9.1

Example of state (hence also safety) property: absence of errors

As an example, figure 9.1 describes graphically a generic state property, which expresses that a set of *error states* should not be encountered during any program execution. Two correct executions are shown in figure 9.1(a), whereas figure 9.1(b) displays an erroneous execution. As we can see in

figure 9.1(b), showing that a program does not satisfy this property boils down to exhibiting at least one finite error trace.

However, not all safety properties are state properties. As an example, let us consider an array-sorting program. The partial correctness of this program is essentially the conjunction of two properties: first, the output array should be sorted; second, it should contain the same elements as the input array. Both properties are safety properties since they can be described by sets of program executions and because it is possible to find a finite counter-example execution for any program that violates them (we simply need to consider an execution where the program returns an incorrect result). To verify the second property (preservation of the set of elements stored in the array), one should check that the elements present in the array when an execution reaches the program exit point exactly match those in the initial state of that execution. This preservation property is naturally expressed as a relation between pairs of states formed by the initial and final states of each execution. On the other hand, it cannot be captured by the data of a set of states; indeed, if one observes only the final states, it is impossible to determine whether the set of elements initially present in the array was preserved (an execution leading to given observed final states may have either preserved the set of elements in the array or modified it). In other words, one needs to observe not only the final states but also some of the executions that led to them in order to decide whether the program preserves the elements stored in the array. Moreover, we also observe that this property fits the definition of safety since a counter-example is necessarily a finite execution that goes from the entry point to the exit point (and that modifies the elements present in the array).

Proving Safety Properties by Static Analysis Let us now study the verification of safety properties using static analysis techniques. In previous chapters (e.g., in chapters 3 and 4), we have discussed static analyses that compute a superset of the set of all the reachable states. Such analyses are naturally adapted to the verification of state properties; indeed, since they return an over-approximation of the set of reachable states, their result may be used to verify that certain states (e.g., error states, or states at the exit point of the program, and where the array is not sorted) are unreachable.

This is actually an instance of a more general verification principle. In general, the verification of a state property P boils down to the discovery of an invariant that entails P . Figure 9.1(c) illustrates this principle in the case of

the state property defined by a set of error states that should never be reached during the execution of a program. To prove it, one simply needs to identify a set of states that defines an *invariant* (i.e., the program always starts from a state in the invariant, and any computation step starting from a state within the invariant also leads to another state that belongs to the invariant) and is disjoint with the set of error states. From the static analysis point of view, this means that proving the state property P essentially involves computing an invariant, by abstract interpretation of the program to verify.

In fact, general safety properties (and not just state properties) can be reasoned over in a similar manner, using some kind of reachability analysis. To show this, we consider the preservation of the elements stored in an array by a sorting program. As noted above, this is a trace property but not a state property. Despite this, we demonstrate that it is amenable to reachability analysis techniques that are very similar to those studied in the previous chapters. In those chapters, we let program states be pairs made of a control state and a memory state, and we let abstract states describe sets of memory states. Let us slightly extend this notion of states so as to account for the changes of the set of elements stored in the array:

- We enrich each concrete state with a function from values to scalars that collects all the values that were added or removed in the array with the multiplicity describing the number of times a value was added or removed.
- We enrich abstract states with symbolic information that describes the number of times each element was added to or removed from the array. More precisely, we let a symbolic variable X stand for the multiset of elements initially present in the array, and we let an abstract state be a symbolic formula that describes the modifications to X . As an example, $X \cup \{x\} \setminus \{a[i]\}$ expresses that the value $a[i]$ was removed once and that the value x was added once.

Although we do not make the formalization of the abstraction relation fully explicit, we observe that this notion of states slightly extends the usual one, with some information describing the execution traces that led to each state. Moreover, it is now very simple to express the preservation of the elements in the array both in concrete terms and in abstract terms:

- In an initial state, the function describing the updates to the array should describe the identity (no addition, no removal).

- In the final state, the information attached to the exit state should also describe the identity transformation; namely, it should boil down to X .

Note that we do not need to observe all the information in the execution traces to verify the property of interest; we only need to reason over a rather coarse abstraction of the program history. Moreover, families of trace abstractions were already studied in section 5.1.3, in the context of trace partitioning.

This example summarizes well how safety properties that are not state properties can be tackled with analysis techniques based on reachability that compute invariants expressed in terms of execution traces (and not just states).

9.1.2 Liveness

As remarked in chapter 1, a *liveness property* is a trace property that can be refuted only using infinite counter-examples, that is, that states a program will never exhibit some specific behavior that can be defined only with infinite traces. The following paragraphs present important examples of liveness properties and discuss how static analysis can be used to prove them automatically.

A Few Families of Liveness Properties The canonical example of a liveness property is *termination*. Indeed, a program violates the termination property if and only if there exists one execution that does not terminate, which is by definition an infinite execution. In other words, it should have no infinite execution. On the other hand, observing an isolated finite execution provides no information on the termination behavior since it could either reach the program exit point at the next step and terminate or loop forever. Not only is termination the most common liveness property, but it also carries a lot of intuition about liveness properties in general; thus, we discuss it more in detail below.

We can cite many other useful program properties that are liveness. Let us consider a program that is supposed to run forever, with at least one integer variable x and the following specification: the values that x takes during the execution should not be ultimately forever of the same sign. Another way to state this is that, for each execution, and at each time there must exist a future instant where x is positive, and another future instant where x is negative. This property is similar to the *fairness* of a scheduler that manages a resource in a system running two processes: the scheduler should decide at each instant which process gets the exclusive right to use the resource, without starving

either of the two processes. If we consider the infinite executions, it is very easy to identify the traces that are admissible and those that are not. On the other hand, a given finite execution provides no information on whether or not the program that produces it satisfies the property: even if x always has the same sign over a given finite trace, we cannot rule out that infinitely many sign alternations may occur in the future.

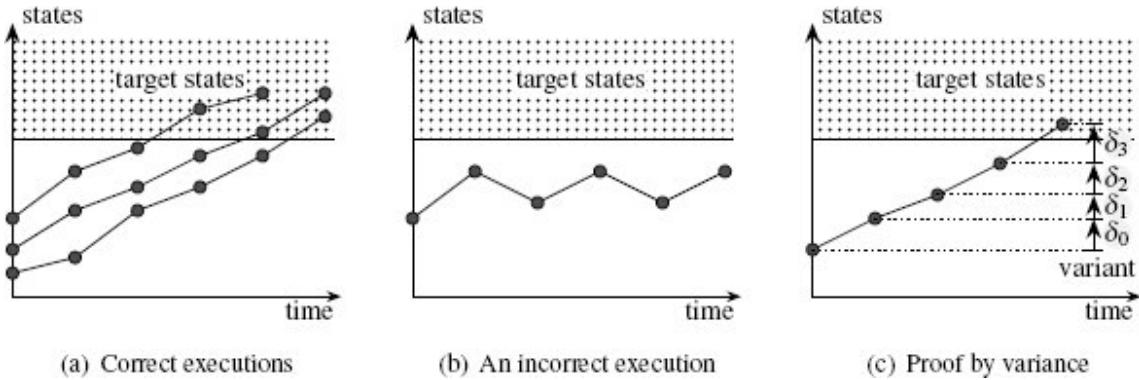


Figure 9.2

Example of liveness property: eventually reaching a target state

Proving Liveness Properties by Static Analysis We now discuss how it is possible to reason over liveness properties using static analysis techniques. While the techniques discussed in the previous chapters naturally compute an over-approximation for reachable states (or, with some small additional work, for reachable traces, as discussed in section 9.1), they do not immediately allow proving that certain *infinite* executions do not occur. However, in the following, we show that it is actually possible to reason over liveness properties using similar abstractions.

Before this, we discuss how paper proofs of liveness properties are usually done. We consider a graphical example shown in figure 9.2. In this example, we are interested in proving that all executions eventually reach a set of target states represented by the dotted area at the top. We observe that termination fits this view; indeed, proving that a program always terminates is equivalent to showing that all its executions eventually reach a final state. In figure 9.2(a), we show a few program executions that are all correct, as they all reach the set of target states after a finite number of steps. In figure 9.2(b), we

present an execution trace that alternates forever between two given states and never reaches the set of target states; thus, it violates the property. The usual way to deal with the proof of such properties is to exhibit a *variant*, a quantity that evolves toward the set of target states and guarantees any execution will eventually reach that set. Usually, it is a value that is strictly decreasing for some well-founded order relation. A common example of variant is an expression of integer type that always takes a positive value, and is strictly decreasing, at each program execution step. A case where such a variant can be found is illustrated in figure 9.2(c). Therefore, we show how a variant can be computed automatically by static analysis.

| | |
|--|--|
| $i := n_0;$ $i := n;$ $r := 1;$ $\text{while}(i > 0)\{$ $\quad r := r * i;$ $\quad i := i - 1;$ $\}$ | $r := 1;$ $\underline{c} := 2;$ $\text{while}(i > 0)\{$ $\quad r := r * i;$ $\quad i := i - 1;$ $\quad \underline{c} := \underline{c} + 3;$ $\}$ |
| (a) A factorial computation program | (b) An instrumented program |

Figure 9.3

Termination example

To do that, we restrict ourselves to the case of termination, and we illustrate the discussion with the program shown in figure 9.3(a) that computes the factorial of n into variable r . In the following, we extend states with an additional integer component (in addition to memory states and control states) that describes the number of execution steps observed since the beginning of the execution of the program. We let variable c store the value of this “step counter.” To maintain the consistency of the step counter, we make the following semantic assumptions:

- When a program execution starts, c is equal to zero.
- Each program execution step increments c by one.

Essentially, termination states that, over a given execution, this value should not grow unbounded. However, this does not mean that it cannot take arbitrarily high values for a given program. As an example, if we consider the program of figure 9.3(a), and if we let k be a positive integer, then there exists at least one execution that makes more than k steps: we simply need to start from a state such that $n = k$, so that the program will run for $3k \geq k$ execution steps (the condition test and two assignments are evaluated for each iteration in the loop), since it will iterate the loop k times.

Nevertheless, this observation allows deriving a bound on the number of steps performed by the program; indeed, the number of iterations is a factor of the value of n . The value of n does not change along the execution of the program; thus, we write n_0 for the initial value of n for the sake of clarity. During an execution, the value of c is thus smaller than $3n_0 + 2$, which fully expresses the argument that a paper termination proof would rely on. The important point is that the inequality $c \leq 3n_0 + 2$ actually defines a state property (assuming we consider the states extended with the c counter) and can be proved using an invariant. Indeed, we observe that, at loop head, we always have $0 \leq i \leq n_0$ and $0 \leq c \leq 2 + 3(n_0 - i)$. This invariant provides the aforementioned upper bound for c . Furthermore, it can be computed by the static analysis techniques introduced in chapters 3 and 4, and using a numerical domain such as convex polyhedra [30] applied to the instrumented version of the program of figure 9.3(a), which we show in figure 9.3(b) (for clarity, we group the assignments to c so as to account the effect of each basic block).

We can derive from this example a more general technique to tackle termination verification using static analysis techniques:

1. Augment states with a program counter variable.
2. Augment each state with a copy of the initial values of the other program variables.
3. Use standard static analysis to compute relations between this step counter and the current and initial values of the numerical variables.

This example can be viewed as a simplified instance of more general and advanced static analysis techniques to prove termination and other liveness properties [23, 107], which are able to infer *ranking functions*.

9.1.3 General Trace Properties

The previous two subsections considered two interesting categories of trace properties, namely, safety and liveness properties. We now consider trace properties that do not necessarily fit either of these two categories. This includes the total correctness of the array-sorting program that was introduced at the beginning of section 9.1. As total correctness does not fall either in the safety category or in the liveness category, it cannot be tackled directly using the static analysis techniques shown in sections 9.1.1 and 9.1.2.

Fortunately, there exists a general theorem [3] that states that any trace property T can be decomposed as the conjunction of a safety property T_{safe} and a liveness property T_{live} :

$$T = T_{\text{safe}} \wedge T_{\text{live}}$$

In terms of verification and static analysis, this entails that the verification of T can be decomposed into two (possibly independent) analyses:

- one for the verification of T_{safe} , using techniques described in section 9.1.1; and
- one for the verification of T_{live} , using techniques described in section 9.1.2.

Often both analyses may take advantage of certain common facts about the program executions, so that the two analyses may better be done together, but this is not a requirement, and one may even opt to use two independent tools, respectively, dedicated to safety and liveness.

This approach based on the decomposition of the verification task into safety and liveness components is an instance of a long-known program proof method. In particular, the *Floyd proof method* [44] also relies on such a decomposition and uses invariant assertions that capture both invariance properties to deal with the safety part, and the existence of a well-founded decreasing entity to deal with the liveness part.

As an example, in the case of the array sorting program and of its total correctness specification, T_{live} states that the program should terminate, whereas T_{safe} corresponds to the three other items mentioned in the beginning of section 9.1: absence of errors, production of a sorted array, and preservation of the elements initially stored in the array.

9.2 Beyond Trace Properties: Information Flows and Other Properties

The goal of this section is twofold. First, we show that some classes of very important semantic properties (including properties related to security) fall out of the traces properties studied in section 9.1. Second, we highlight a few techniques to attack such properties.

Definition of Information Flows In most of this section, we look at *information flow properties*. By definition, an information flow occurs when the observation of the value of one variable gives information about the value of another. This sort of property is actually useful in many contexts.

A most notable application is *security*. Indeed, let us imagine a shared application that deals with secret information, such as an Internet banking website. Then all customers should be able to watch the general public information and the information relative to their own account such as their balance, but not the balance of other customers: even a partial leak of information (such as “user A is richer than user B”) would be perceived as a major breach into privacy. Similar concerns arise in many other kinds of systems as well, such as a multiuser operating system that manages permissions across users, and make sure that each user may read and write only into her/his own private space (except for the superuser who is granted stronger permissions). More generally, such security properties aim at proving that information that should be accessible only with a high privilege will not flow into variables accessible even with low privilege [32]. In this case, it is often interesting to prove the absence of information flows. In the abstract non-interference setup [45], the information that should not be leaked and the observation of program behaviors by attackers can be formalized as abstractions in the sense of definition 3.2.

Another application is *program understanding*. When maintaining programs, developers often need to know what part of a program to look at in order to understand how a given feature works, so as to fix it or augment it with additional functionalities. Then, to understand it, they need to figure out what other parts of the program may have an impact on the variables related to this feature. Such a relation between a program statement and a variable of interest is called a *dependence* and corresponds to a form of information flow, where the observed values of the variable may be different if that program statement receives different inputs [71, 42, 16]. Program investigation tools such as slicers [109] need to compute an over-approximation of the components of a program for which there exists an information flow. Besides

program understanding, dependence information is also often useful to carry out code optimizations (e.g., independent computations may be parallelized) or to enable client static analyses (e.g., to determine program fragments that may be ignored).

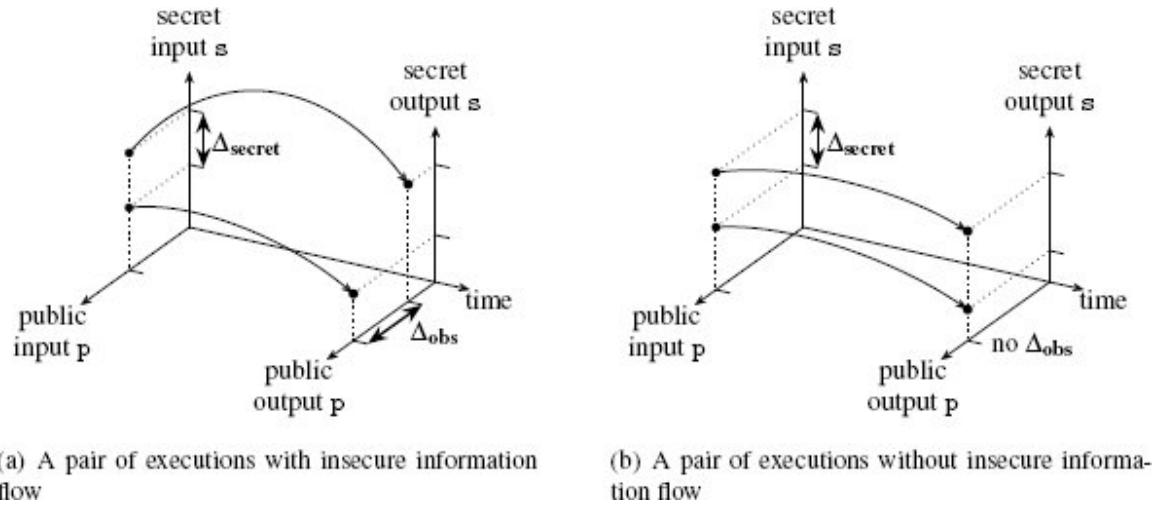


Figure 9.4

Information flows

To make information flow properties intuitive, we study a couple of examples from a security point of view. These are quite contrived (although representative) so that we can easily represent their inputs and outputs. We assume a program that operates on two variables p (as “public variable”) and s (as “secret variable”), and we look for the existence of flows between the secret variable into the public one. The absence of flows can be described as follows:

The observable (related to p) produced by two executions

the initial states of which differ only by the value of s

should be the same.

To make this property more intuitive, we provide graphical examples in figure 9.4. For the sake of simplicity, we assume that we are considering only

deterministic programs. We use three dimensions since one accounts for time (from left to right) and two account for the values of the two variables (the value of the secret variable is described by the upward axis). Using these conventions, the pair of executions shown in figure 9.4(a) correspond to a case where some private information is leaked. Indeed, the initial states differ only in the value of the secret variable s ; however, the public observation in the output states differ since p takes different values for these two executions. This means that an attacker may derive from the observation of the output the initial value of the secret variable. On the other hand, figure 9.4(b) displays a case where no private information is leaked. Indeed, the final values of the public variable are the same for the two executions, even though the initial values of the secret variable differ. To summarize, a program may leak private information if and only if there exists a pair of executions such as those shown in figure 9.4(a). This intuitive view generalizes to non-deterministic programs easily. Indeed, in this case, we simply need to reason over *sets* of initial/final values for p and s .

$$C_0 ::= p := [0, 1]; \quad C_1 ::= p := s * [0, 1]; \quad C_2 ::= p := \text{rand}([0, 1]) - s;$$

Figure 9.5

Comparing information flows

Characterization of Information Flows We now show that information flow is not a trace property, and why it is difficult to reason about. In this paragraph, we assume that variables s and p may take only values 0 and 1. We consider two programs C_0 and C_1 , which are defined in figure 9.5. Essentially, C_0 writes a non-deterministically chosen value into p , whereas C_1 writes the product of a non-deterministically chosen value with the secret variable into p . Finally, C_2 chooses a value in $[0, 1]$ in a non-deterministic manner, subtracts s from this value, and stores the result into p . These programs have the following input-output tables (we show only the output values of the public variable p , as the output value of s plays no role in the existence of an information flow); note that each initial state may correspond to several possible output values (i.e., the executions may be non-deterministic):

| | |
|---|---|
| $C_0 : \begin{array}{l} (p : 0, s : 0) \mapsto (p : \{0, 1\}) \\ (p : 0, s : 1) \mapsto (p : \{0, 1\}) \\ (p : 1, s : 0) \mapsto (p : \{0, 1\}) \\ (p : 1, s : 1) \mapsto (p : \{0, 1\}) \end{array}$ | $C_1 : \begin{array}{l} (p : 0, s : 0) \mapsto (p : \{0\}) \\ (p : 0, s : 1) \mapsto (p : \{0, 1\}) \\ (p : 1, s : 0) \mapsto (p : \{0\}) \\ (p : 1, s : 1) \mapsto (p : \{0, 1\}) \end{array}$ |
|---|---|

Obviously, observing the value produced for variable p by C_0 provides no information on the initial value of s . On the other hand, C_1 has a more complex behavior. For some values for the secret input it always returns 0, whereas for others it non-deterministically returns states where p may be either 0 or 1. Looking at the table, if we run C_1 with from some input state and observe that p is equal to 1 in the end, we know that s was necessarily equal to 1 in the initial state. That means that there is a flow of information from the initial value of s into the final value of p (note that when the final value of p is 0, we do not learn anything about s). In other words, if p is public and s stores a piece of secret information, the program C_1 is not secure. Furthermore, we note that the program C_2 defined in figure 9.5 has the same semantics as C_0 and is thus secure, even though the computation of p syntactically reads s .

These two examples also demonstrate why information flows are *not* trace properties in the sense of section 9.1. Indeed, let us consider the property to “be secure,” namely, that there is no information flow from s to p . We have shown that C_0 is secure whereas C_1 is not. However, we can also see that C_0 has more behaviors than C_1 . As remarked in section 9.1, this means that, if the absence of information flow were a trace property (i.e., if it could be defined as a set of “correct traces”), and since C_0 satisfies it and has more behaviors, C_1 should also satisfy it. However, C_1 does not satisfy information flow, so the absence of information flows is not a trace property. The fundamental reason is that trace properties are defined by quantifying once over execution (i.e., with statements that write down as “for each execution, some property holds”), whereas the absence of information flow requires a quantification over a *pair* of executions (namely, “for each pair of executions starting from initial states that differ only in the value of s , the final observations of p are the same”). The term *hyperproperty* [22] is often employed to describe a semantic property that requires to quantify about a number of executions that

may be greater than 1. Therefore, hyperproperties include not only trace properties but also many semantic properties that cannot be defined as a trace property. In general, a hyperproperty is defined by a *set of sets of traces*.

This need to quantify over several executions also makes reasoning about information flow difficult. In the previous chapters, we have extensively used over-approximation of program behaviors, so as to design automatic static analyses. In particular, when a program fragment C is hard to analyze but has no impact on the analysis goal, we would like to safely ignore it and over-approximate its behaviors in a coarse manner. When considering trace properties, the simplest and most imprecise over-approximation is simply to consider that the fragment C may have any possible behavior, just like the C_0 fragment above. However, this approach would be *incorrect* and *unsound* here since the absence of an information flow is not a trace property.

As a consequence, instead of directly over-approximating the set of program traces or states, we need to utilize other methods. We present three common approaches in the following paragraphs. To do that, we consider the absence of information flow, and we let it be defined by a set \mathcal{I} of sets of executions, which contains all the semantics such that there is no information flow from s to p . Intuitively, the absence of information flow is characterized as the set of all the sets of traces that are secure, in the sense that they satisfy the statement “for each pair of execution starting from initial states that differ only in the value of s , the final observations of p are the same.”

Abstraction of Sets of Executions A first approach is to actually search for a trace property T_{safe} that implies the hyperproperty of interest and to attempt to verify this property by the static analysis techniques presented in this book. Formally, T_{safe} should be such that:

1. T_{safe} defines a safety property (thus, it is a set of execution traces);
and
2. for all programs p , if $\llbracket p \rrbracket \subseteq T_{\text{safe}}$, then $\llbracket p \rrbracket \in \mathcal{I}$.

As an example, many static analyses to compute information flows or dependences attempt to approximate a kind of *taint* property that can intuitively be defined as follows:

- In the case of an assignment $x := E$, we say that the taints of the variables that occur in E may flow into x .

- In the case of a program with a condition $\mathbf{if}(E)\{...x := ...\}$, we say that the taints of the variables that occur in E may flow into x .

Intuitively, this taint property captures relations between variables computed using (directly or indirectly) the content of other variables. As such, it describes a stronger property than the flow property that we are interested in. For instance, if we consider $\mathbf{if}(y > 0)\{x := 1\} \mathbf{else}\{x := 1\}$, then there is no information flow from y to x since x gets assigned to 1 whatever the value of y . However, the computation of x reads y since it needs to evaluate the condition, so there is a taint relation between these two variables even though x is 1 regardless of y . The same goes for $x := 0^*y$. However, the advantage of this approach is that the taint property is actually a safety property and can be attacked with regular abstractions based on set of states or traces. To summarize, this approach proceeds as follows:

- It replaces the target hyperproperty that does not fit the traces property definition with another property that does, and that is stronger than it (in other words, this step introduces some steep imprecision that the second step cannot recover from, as shown in the above examples).
- It performs a static analysis to establish this stronger trace property, using techniques seen in section 9.1, and chapters 3 and 4.

We can apply this technique to the three example programs mentioned above: C_0 is accepted as secure since p receives no taint from s ; C_1 is (rightfully) rejected as potentially insecure as there exists such taint; and finally, C_2 is also rejected since the computation of p syntactically reads the value of the secrete variable s . This last case shows the limit of the taint approach in terms of precision; indeed, the failure to accept C_2 stems from the imprecision in the first step.

Static Analysis Based on Self-Composition As defined above, the absence of information flow from s to p expresses a property over pairs of execution. In order to adapt techniques that can reason about one execution, one can encode in a single execution the property over the pair of executions. This approach is known as *self-composition* [7, 31]. It reduces the verification that a program p is free of information flows to the verification of another program p' , which essentially performs two executions of p in a single run, which start with

states that differ only in the value of the secret variable. When the programs are deterministic, p' can be derived from p by duplicating variables and adding assertions. The property that should be verified on p' simply asserts that the public values obtained for each execution should be the same, so that it is not possible to infer any knowledge about the secret variable. It is thus a state property.

Abstraction of Sets of Sets of Executions A second approach is to directly compute over-approximation of the *set of set of executions* $\{ \llbracket p \rrbracket \}$ and to use it to establish the absence of information flow (or, more generally, of the hyperproperty of interest). Formally, this approach lifts analysis techniques to sets of sets of states and attempts to compute P such that

1. it over-approximates the program behaviors in the sense that $\{ \llbracket p \rrbracket \} \subseteq P$; and
2. it implies the absence of information flows, that is, that $P \subseteq \mathcal{I}$.

Overall, the static analysis relies on the principles presented in chapters 3 or 4, even though it builds on more sophisticated abstract domains [6] because an abstract element should over-approximate a set of sets of executions (and not just a set of executions). The advantage of this approach is that it does not introduce the irrecoverable imprecision that the previous approach introduces in its very first step.

As an example, we consider the following abstraction of sets of sets of executions, for the programs C_0 , C_1 , and C_2 :

- We let the abstract element a_0 stand for the set of sets of traces where p may take, regardless of s , any value at the end, whatever the input condition.
- We let the abstract element $a_1 = \top$ represent any set of sets of traces.

Then we may design an analysis that returns a_0 both for C_0 and C_2 , thus accepting them, whereas C_1 is still rejected. In this case, this approach is thus more precise than the taint approach.

Other Families of Properties To conclude this section, we would like to point out that there are other interesting properties besides information flows (and the related security and dependence properties) that can be represented only as sets of sets of executions. As an example, some such properties quantify over

many executions, or even over all executions, such as *average* execution time. Indeed, if we would like to establish that average execution time of some system is less than a given bound δ (assuming that all its execution traces have the same probability of occurring), we need to consider all its executions. As we have shown for information flows, we can actually either try to consider a stronger trace property instead, or seek for an abstraction that can describe sets of sets of traces. As an example of the first approach, we can cite worst-case execution time. Worst-case execution time analyses aim at computing an upper bound for the time elapsed during any execution of the program [40]. If such an analysis establishes that all executions take less than δ , then the average execution time is also lower than that value.

10 Specialized Static Analysis Frameworks

Goal of This Chapter We discuss three specialized frameworks that are less general but simpler than abstract interpretation.

Specialized frameworks are analogous to domain-specific programming languages as opposed to general-purpose ones. For a limited set of target languages and properties, these specialized frameworks are simple to use yet powerful enough. They can be practical alternatives to the general abstract interpretation framework when the target languages and properties are good fits for them. The burden of soundness proof can be reduced and the special algorithms can outperform the general worklist-based fixpoint iteration algorithms.

Recommended Reading [S], [D] This chapter is targeted at students and developers who would like to explore specialized static analysis techniques that can be simple yet powerful enough for specific cases in hand. This chapter is also a reminder for readers who are already familiar with the specialized static analysis techniques of limitations, if any, and how these techniques can be seen from the general abstract interpretation point of view. This chapter is intended more as a survey of the specialized frameworks than an in-depth coverage.

Chapter Outline: Three Specialized Static Analysis Frameworks, Theirs Uses, and Their Limitations We present three classes of specialized static analysis frameworks. We discuss their uses and their limitations from the perspective of the general frameworks presented in chapters 3 and 4.

1. In section 10.1, we present a specialized framework that has long been used in optimizing compilers. We present an example called *data flow analysis* and discuss its limitations.
2. In section 10.2, we discuss another class of specialized framework in which a static analysis computes a monotonic

closure of facts. We present two examples of this framework and discuss their limitations.

3. In section 10.3, we discuss final class of specialized framework where a static analysis constructs a proof about programs. We present an example called *static typing* and discuss its limitations.

10.1 Static Analysis by Equations

From the equations point of view, static analysis comprises equation setup and equation resolution. For an input program, a set of equations captures all the executions of the program. A solution of this set of equations is the analysis result.

Capturing the dynamics of a program by a set of equations is indeed implicit in the general frameworks of chapters 3 and 4. There, the concrete semantics of a program (and usually, its finitely computable abstract version) is defined as a fixpoint of the program's semantic function. Note that a function f 's fixpoint c , such that $c = f(c)$, is nothing but a solution of the equation $x = f(x)$.

However, the general frameworks of chapters 3 and 4 are sometimes overkill. When the target programming language is simple enough, it is rather straightforward to set up correct equations for programs without formally defining an abstract semantics that soundly approximates the concrete semantics of the target language [2].

10.1.1 Data-Flow Analysis

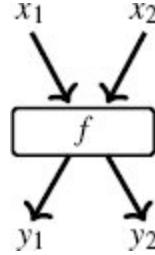
The target programming language is simple enough to be suitable for this approach to static analysis when the execution order (control flow) of a program is fixed and explicit from the program text before the execution and when the execution of each construct is simple.

The input programs of such languages are represented by control-flow graphs, and the analysis focuses on the flow of data over the fixed control-flow graphs. By this reason, this approach to static analysis is called *data-flow analysis*.

Program as a Graph Suppose a simple imperative language with assignment, sequence, if-branch, while-loop, no pointers, and no function calls. The

control flows of programs are always fixed and clear from the program source code. The control flow of a program is represented as a directed control-flow graph. The nodes are the atomic statements or conditions of the program. The directed edges determine the execution order (control flow) between the statements. An if-branch statement has two branches in the graph: one with the condition expression node followed by the true branch statements, and the other with the node for the negation of the condition expression followed by the false branch statements. Similarly for the while-loop statement.

Equations The equations are about the states that flow at each edge of the graph. For each node of the form (without loss of generality, we consider two incoming and outgoing edges, respectively) we have the following:



The edges describe the control flow, and each node f is a state transformation function for the corresponding statement of the program. The function f simulates the statement's semantics: it transforms the incoming machine state (pre-state) into the resulting machine state (post-state). The x_i are two incoming pre-states, and y_i are the post-states that flow out along the outgoing edges. Then the equations are set up as follows:

$$\begin{aligned} y_1 &= f(x_1 \sqcup x_2) \\ y_2 &= f(x_1 \sqcup x_2) \end{aligned}$$

The value space that the unknowns (x_i and y_i) range over is a lattice whose partial order corresponds to the information subsumption: $a \sqsubseteq b$ means that the set implied by a is also implied by b . An expression of the form $a \sqcup b$ denotes the least upper bound of a and b , the least element that subsumes both a and b . The function f is a monotonic function that describes the operation (semantics) of the node's statement over the lattice of values.

Example 10.1 (Data-flow analysis) Consider the following program:

```

input (x);
while (x <= 99)
{ x := x+1 }

```

The control-flow graph of this program is shown in figure 10.1. The edges are numbered. The empty node that does nothing corresponds to the identity transformation function.

Suppose that we are interested in the value range of variable x on each edge. Suppose that we represent value ranges as integer intervals. A set of equations for the values x_i of x at each program label i would be, for example, as in figure 10.2:

- The equation for x_0 ,

$$x_0 = [-\infty, +\infty]$$

describes that x at edge 0 may be any integer because the input is unknown.

- The equation for x_1 ,

$$x_1 = x_0 \sqcup x_3$$

describes the value of x at edge 1 has x 's value at 0 or 3 (because of the while-loop iterations).

- The equation for x_2 ,

$$x_2 = x_1 \sqcap [-\infty, 99]$$

says the value of x at edge 2 has the value at edge 1 but is less than or equal to 99.

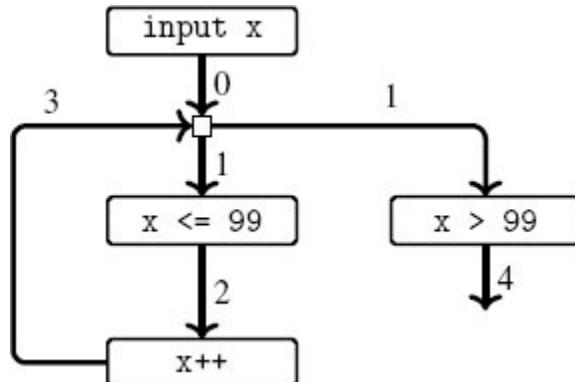


Figure 10.1

An example control-flow graph

$$\begin{aligned}
x_0 &= [-\infty, +\infty] \\
x_1 &= x_0 \sqcup x_3 \\
x_2 &= x_1 \sqcap [-\infty, 99] \\
x_3 &= x_2 \oplus 1 \\
x_4 &= x_1 \sqcap [100, +\infty]
\end{aligned}$$

Figure 10.2

A set of equations for the program in figure 10.1

- The equation for x_3 ,

$$x_3 = x_2 \oplus 1$$

says the value of x at edge 3 has the value incremented by 1 to the value at edge 2.

- The equation for x_4 ,

$$x_4 = x_1 \sqcap [100, +\infty]$$

says the value of x at line 4 has the value at edge 1 but is greater than 99.

The definitions of the operators (\sqcup , \sqcap , \oplus) in the equations are as follows. The soundness of these definitions is easy to see:

$$\begin{aligned}
[a, b] \sqcup [a', b'] &= [\min(a, a'), \max(b, b')] \\
[a, b] \sqcap [a', b'] &= \text{if } \max(a, a') > \min(b, b') \text{ then } [] \text{ else } [\max(a, a'), \min(b, b')] \\
[a, b] \oplus c &= [a + c, b + c]
\end{aligned}$$

It is obvious that the equations in figure 10.2 with the above definitions of the operators capture all the execution cases of the program. Formally proving this correctness is much ado for the obvious.

Computing a solution of the equations can be done by the general fixpoint iterations such as those in section 4.3 because all operators are monotonic over the lattice space of the intervals. For finite height lattice, these naive iterations can reach a fixpoint in finite time.

$$\begin{aligned}
x_0 &= [-\infty, +\infty] \\
x_1 &= x_0 \sqcup x_3 \\
x_2 &= x_1 \\
x_3 &= x_2 \oplus 1 \\
x_4 &= x_1 \sqcap [100, +\infty]
\end{aligned}$$

Figure 10.3

Another set of sound equations for the program in figure 10.1

When the lattice is of infinite height, as is the interval lattice, the naive fixpoint iterations cannot terminate for some programs. In this case we can use a sound finite computation technique such as the widening operator in chapters 3 and 4 to guarantee the termination and the soundness (over-approximation) of the result.

Limitation Though the above equations are simple, so their correctness is obvious, we cannot rely on the approach for arbitrary languages. First, the dichotomy of control being fixed and data being dynamic does not hold in modern languages. If the target language allows the control flow as values (e.g., jump targets as values, functions as values, or exceptions as values), the control-flow graph a priori is not possible. The control flow can generally not be represented as a static finite graph. Edges are introduced during program execution. The control flows emerge only during static analysis.

Second, the sound transformation functions of each node are not obvious for modern languages. In example 10.1, though the operators \sqcup , \sqcap , and $\oplus 1$ are straightforward for the example program construct at each node, for complicated languages and value domains the sound definitions can be error prone. A framework to check the transformation function's soundness (e.g., theorems 4.4, 8.1, and 8.2) is necessary so that the equation solution is guaranteed sound.

The third issue is about design choices. For a program, the equations are not unique. There are a vast number of ways to set up sound equations that over-approximate all the executions of the program. From this collection of sound equation sets, an analysis chooses one particular way of setting up equations. For example, given the program in example 10.1, different

equation sets may be chosen, whose solutions also cover all the possible executions of the program. One such example is shown in figure 10.3.

The data flow analysis lacks a systematic approach to explore various possible choices of sound equations for a given program. For every choice, though simple, we have to argue the soundness of the equations from scratch.

Meanwhile, the semantics-based general frameworks (chapters 3 and 4) work for an arbitrary programming language once the concrete semantics of the language is defined in the styles supported by the frameworks. From the point of view of data flow analysis, the semantic frameworks guide us on what to check (sections 3.4 and 4.2.3) to set up sound equations from the programs to analyze. In the semantic frameworks, sound and finitely computable static analyses with various accuracies can be systematically defined only if the required properties of the abstract domains and of the abstract semantic function are ensured.

10.2 Static Analysis by Monotonic Closure

From the monotonic-closure point of view, static analysis comprises setting up initial facts and then collecting new facts by a kind of chain reaction. An analysis definition consists of rules for collecting initial facts and rules for inductively generating new facts from existing facts. An analysis in action accumulates facts until no more facts are possible. This collection of facts is an over-approximation of the program behavior.

The initial facts are those that are immediate from the program text. The chain reaction steps generate new facts from existing facts by simulating the program semantics. The analysis terminates when no more facts are generated. For the analysis to terminate, the universe of facts for a given input program must be finite.

For simple target languages and properties, the soundness of the analysis—setting up the initial facts and the chain reactions—is straightforward.

Formally, let R be the set of the chain-reaction rules, and let X_0 be the initial fact set. Let $Facts$ be the set of all possible facts. Notation $X \vdash_R Y$ denotes that new set Y of facts is generated from X by applying all possible rules in R . Then the analysis result is

$$\bigcup_{i \geq 0} Y_i,$$

where

$$\begin{aligned} Y_0 &= X_0, \\ Y_{i+1} &= Y \text{ such that } Y_i \vdash_R Y. \end{aligned}$$

Or, equivalently, the analysis result is the least fixpoint

$$\bigcup_{i \geq 0} \phi^i(\emptyset)$$

of monotonic function $\phi : P(\text{Facts}) \rightarrow P(\text{Facts})$:

$$\phi(X) = X_0 \cup (Y \text{ such that } X \vdash_R Y).$$

10.2.1 Pointer Analysis

Let us consider the following simple C-like imperative language. A program is a collection of assignments.

| | | |
|---------|--------------------|---------------------|
| $P ::=$ | C | program |
| $C ::=$ | | statement |
| | $L := R$ | assignment |
| | $C ; C$ | sequence |
| | while $B C$ | while-loop |
| $L ::=$ | $x *x$ | target to assign to |
| $R ::=$ | $n x *x \&x$ | value to assign |
| B | | Boolean expression |

The left- or right-hand side of an assignment may be a variable(x), the dereference ($*x$) of a variable, or the location ($\&x$) of a variable. Let the semantics of these reference and dereference constructs be the same as in the C language. A variable may store an integer or the location of a variable.

Target Property Suppose that we are interested in the set of locations that each pointer variable may store during program execution. Computing such a set is reduced to collecting all possible “points-to” facts between two variables: which variable can store the location of which variable. We hence represent each points-to fact by a pair of two variables: $a \rightarrow b$ denotes that

variable **a** can point to (can have the address of) variable **b**. The set of such points-to facts is finite for each program because a program has a finite number of variables.

Rules The analysis globally collects the set of possible points-to facts that can happen during the program execution. We start from an initially empty set. We apply the following rules to add new facts to the global set. This collection (hence the analysis) terminates when no more addition is possible. The rule has the form

$$\frac{C \quad i_1 \dots i_k}{j}$$

and dictates that, if the program text has component C , and the current solution set has $i_1 \dots i_k$, then add j to the solution set [58]. The $i_1 \dots i_k$ part can be omitted.

The initial facts that are obvious from the program text are collected by this rule:

$$\frac{x := \&y}{x \rightarrow y}$$

That is, for each assignment statement of the form $x := \&y$, add the fact that x points to y .

The chain-reaction rules are as follows for other cases of assignments:

$$\begin{array}{c} \frac{x := y \quad y \rightarrow z}{x \rightarrow z} \quad \frac{x := *y \quad y \rightarrow z \quad z \rightarrow w}{x \rightarrow w} \\ \\ \frac{*x := y \quad x \rightarrow w \quad y \rightarrow z}{w \rightarrow z} \quad \frac{*x := *y \quad x \rightarrow w \quad y \rightarrow z \quad z \rightarrow v}{w \rightarrow v} \\ \\ \frac{*x := \&y \quad x \rightarrow w}{w \rightarrow y} \end{array}$$

Soundness It is easy to see that the above rules will collect every possible points-to fact of the input program. The easiness comes from the simple semantics of the target language. The six rules follow, in terms of the points-to information, from the semantics of all six cases of the assignment statements. Consider the last rule, for example. The assignment statement $*x := \&y$ stores the location $\&y$ of y to the location that x points to. Thus,

the rule adds $w \rightarrow y$ if x points to w . It is similarly straightforward to see the soundness of other rules.

Given the global set of facts, applying all the possible rules to the set is a monotonic operation; every rule always adds new facts, if any, to the global set. The analysis result is the least fixpoint of this applying-all-rules function.

The over-approximation comes from two sources. First, the rules ignore the conditional execution of the while-loop. Regardless of the while-loop condition, the rules are applied to the assignments in the loop body. Second, we collect the points-to facts into a single global set. This means that a points-to fact from any statement in the input program can trigger a new points-to fact at any statement.

Example 10.2 (Pointer analysis) Consider the following program:

```

x := &a ;
y := &x ;
while B
    *y := &b ;
    *x := *y

```

Initial facts are from the first two assignments:

$$x \rightarrow a, y \rightarrow x$$

From $y \rightarrow x$ and the assignment body of the while-loop, the analysis can apply the last rule to add

$$x \rightarrow b.$$

For the last assignment and the hitherto collected facts, the analysis can apply the second-to-the-last rule and add new facts as follows: from $x \rightarrow a$ and $y \rightarrow x$, the analysis can add $a \rightarrow a$; from $x \rightarrow b$ and $y \rightarrow x$, the analysis can add $b \rightarrow b$; from $x \rightarrow a$, $y \rightarrow x$, and $x \rightarrow b$, the analysis can add $a \rightarrow b$; from $x \rightarrow b$, $y \rightarrow x$, and $x \rightarrow a$, the analysis can add $b \rightarrow a$.

Limitation Note that the above rules do not take the control flow into account. The rules are applied for any assignment statement regardless of where it appears. For example, there is no separate rule for while-loop statement. Regardless of where an assignment appears in the program (e.g., whether an assignment appears within a while-loop body or outside), the analysis blindly collects every possible points-to facts from the collection of the assignment statements in the program.

Similarly, the above analysis is flow-insensitive: the closure rules are oblivious to the statement order in the input program. Regardless of where

the assignment statement appears in the program, whenever the premise (numerators) of a rule holds the analysis applies the rule.

For example, for a sequence of statements

$$x := \&a ; y := x ; x := \&b,$$

the analysis concludes $y \rightarrow a$ and $y \rightarrow b$ because $x \rightarrow a$ and $x \rightarrow b$ even though y has x before $x \rightarrow b$.

This flow insensitivity can indeed be avoided by preprocessing the input program into a *static single assignment* form where each variable is defined only once. Two writes to a single variable and its reads are transformed into writes to two different variables and their distinctive reads.

10.2.2 Higher-Order Control-Flow Analysis

As an another example, consider the following higher-order call-by-value functional language. We uniquely label each subexpression of the program:

$$\begin{array}{ll} P & ::= F \quad \text{program} \\ F & ::= \quad \text{expression} \\ & | \quad x \quad \text{variable} \\ & | \quad \lambda x.E \quad \text{a function with argument } x \text{ and body } E \\ & | \quad E E \quad \text{function application} \\ E & ::= F_l \quad \text{expression } F \text{ with label } l \end{array}$$

A program is defined as an expression that has no free variable. Program executions are defined by the transition steps, each step (\rightarrow) of which is by the *beta reduction* in the call-by-value order. The beta-reduction step is

$$(\lambda x.e) e' \rightarrow \{e'/x\}e,$$

where $\{e'/x\}e$ denotes the expression obtained by replacing x by e' in e . We assume that, during execution, every function's argument is uniquely renamed.

For example, the program

$$(\lambda x.(x(\lambda y.y)))(\lambda z.z)$$

runs as follows:

$$\begin{aligned}
 & (\lambda x.(x(\lambda y.y)))(\lambda z.z) \\
 \rightarrow & \quad (\lambda z.z)(\lambda y.y) \\
 \rightarrow & \quad \lambda y.y
 \end{aligned}$$

During the execution, the first step binds x to $\lambda z.z$ and the second step binds z to $\lambda y.y$.

Target Property Suppose that we are interested in which functions are called for each application expression. Since functions can be passed to parameters of other functions, such inter-functional control flow is not obvious from the program text. For the target property we need to collect which lambda expression can be bound to which argument during program execution.

Rules We let an analysis collect facts about which lambda expression “ $\lambda x.e$ ” a sub-expression may evaluate to. Hence, we represent each fact by a pair $L \ni R$, meaning “ L can have value R ,” where L is either an expression label or a variable, and R is an expression label, a variable, or a lambda expression:

$$\begin{aligned}
 L & ::= l \mid x \quad \text{expression label or variable} \\
 R & ::= l \mid x \mid v \\
 v & ::= \lambda x.E
 \end{aligned}$$

For each program the set of the pairs is finite because a program has a finite number of expressions (l), variables (x), and lambda expressions (v).

From a global set that is initially empty, we apply the following rules to add facts of the form $L \ni R$ to the set. The addition (the analysis) terminates when no more addition is possible. Each rule has the form

$$\frac{C \quad i_1 \dots i_m}{j_1 \dots j_n}$$

and dictates that, if the program text has component C , and if the hitherto collected facts include $i_1 \dots i_m$, then add $j_1 \dots j_n$ to the global set. The $i_1 \dots i_m$ part may be empty.

The initial fact setup rules collect facts that are obvious from the program text only:

$$\frac{(\lambda x.E)_l \quad (x)_l}{l \ni \lambda x.E} \quad l \ni x$$

A lambda expression is a constant whose value is itself, and a variable expression contains the value of the variable. The value of a variable will be later collected when the variable as the parameter of a function is bound to the actual parameter value at the applications of the function.

The propagation rules that collect new facts by simulating expression evaluations are

$$\frac{(E_{l_1} E_{l_2})_l \quad l_1 \ni \lambda x.E_{l_3} \quad l_2 \ni v}{l \ni l_3 \quad x \ni v} \quad \frac{l_1 \ni l_2 \quad l_2 \ni v}{l_1 \ni v}.$$

As in the points-to analysis case, it is easy to see that the above rules will collect every possible value of expressions and variables. For an application expression $(E_{l_1} E_{l_2})_l$, if we knew which function can be called $l_1 \ni \lambda x.E_{l_3}$ with which parameter $l_2 \ni v$, we add two new facts: the first $l \ni l_3$ about the result of the function application and the other $x \ni v$ about the parameter binding. The second rule propagates the actual value along the chains of collected facts. Formally proving the soundness seems again much ado for the obvious.

Example 10.3 (Control-flow analysis) Consider the following program. Every lambda expression is identified by its unique parameter name. Each expression of the program is labeled from 0 to 7.

$$(\lambda x. (\underbrace{x_5 (\lambda y. y_6)}_2)) (\underbrace{\lambda z. z_7}_4)$$

$\overbrace{\hspace{10em}}^1$

$\overbrace{\hspace{10em}}^0$

During the execution, variable x (expression numbered 5) will be bound to $\lambda z.z$, and variable z (expression numbered 7) to $\lambda y.y$.

Let's see how this information is collected from the above rules. The initial facts are collected from the lambda expressions 1, 3, and 4 and variable expressions 5, 6, and 7:

$$\{1 \ni \lambda x.(x(\lambda y.y)), 3 \ni \lambda y.y, 4 \ni \lambda z.z, 5 \ni x, 6 \ni y, 7 \ni z\}$$

- From application expression 0, we add $x \ni 4$ (parameter binding) and $0 \ni 2$ (application result) to the above set.
- Then by the last propagation rule from $x \ni 4$ and $4 \ni \lambda z.z$, we add $x \ni \lambda z.z$, and then from $5 \ni x, 5 \ni \lambda z.z$ to the above set.
- Then from application expression 2, we add $z \ni 3$ (parameter binding) and $2 \ni 7$ (application result) to the above set.

- Then by the last propagation rule, we add to the above set $z \ni \lambda y . y$; then, from $7 \ni z$, we add $7 \ni \lambda y . y$, then $2 \ni \lambda y . y$, and then $0 \ni \lambda y . y$.

Limitation The above analysis uses a crude abstraction for the function values. Note that a function value in the concrete semantics is a pair made of the function code and a table (called *environment*) that determines the values of the function's free variables. The above analysis completely abstracts away the environment part. In the concrete semantics, a function expression ($\lambda x.E$) in a program may evaluate into distinct values at different contexts (e.g., when the function's free variable is the parameter of a function that is multiply called with different actual parameters). The above analysis collects only the function code part, with no distinction for the values of the function's free variables.

To employ some degree of context sensitivity in the abstraction of the function values, the above analysis needs an overhaul, whose design and soundness assurance will be facilitated by semantic frameworks such as those in chapters 3 and 4 (see sections 8.2 and 8.4.1).

10.3 Static Analysis by Proof Construction

From the proof construction point of view, static analysis is a proof construction in a finite proof system. A *finite proof system* is a finite set of inference rules for a predefined set of judgments. The soundness of the analysis corresponds to the soundness of the proof system. The soundness property of the proof system expresses that, if the input program is provable, then the program satisfies the proven judgment. Thus, for a sound proof system, if a program violates a target judgment, then proving the judgment fails for the program.

10.3.1 Type Inference

Let us consider a proof system whose judgment is about the types [91] of program expressions. The set of types corresponds to an abstract domain of a static analysis. A type over-approximates the values of an expression. This proof system is called *type system*, and the proof construction is called *static type inference* or *static typing*.

Let us consider the following higher-order language with the call-by-value evaluation:

| | |
|---------------|----------------------|
| $P ::= E$ | program |
| $E ::=$ | expression |
| n | integer |
| x | variable |
| $\lambda x.E$ | function |
| $E E$ | function application |

In type systems, the judgment that says expression E has type τ is written as

$$\Gamma \vdash E : \tau,$$

where Γ is a set of type assumptions for the free variables in E , and τ is a type. The judgment means that the evaluation of e with its free variables having values of the types as assumed in Γ is type-safe (runs without a type error) and returns a value of type τ if it terminates.

$$\frac{}{\Gamma \vdash n : \text{int}} \quad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma + x : \tau_1 \vdash E : \tau_2}{\Gamma \vdash \lambda x.E : \tau_1 \rightarrow \tau_2} \quad \frac{\Gamma \vdash E_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash E_2 : \tau_1}{\Gamma \vdash E_1 E_2 : \tau_2}$$

Figure 10.4

Proof rules of simple types

Simple Type Inference The above language has two kinds of values, namely, integers and functions. Hence, we use the following types:

$$\tau ::= \text{int} \mid \tau \rightarrow \tau$$

Type int describes integers, and $\tau_1 \rightarrow \tau_2$ the type-safe functions from τ_1 to τ_2 (functions that, if given a value of type τ_1 , run without a type error and return a value of type τ_2 if they terminate).

We say that a program runs without a type error whenever, during the evaluation of the application expression “ $E_1 E_2$ ” in the program, the value of E_1 is always a function value, not an integer.

The type environment Γ is a finite map from variables to types. Let us write each entry as $x : \tau$. The notation $\Gamma + x : \tau$ denotes the same function as Γ except that its entry for x is τ .

The simple type system is shown in figure 10.4. Each proof rule

$$\frac{J_1 \dots J_k}{J}$$

is read as follows: whenever all the premises (J_1, \dots, J_k) are provable, then the conclusion (J) is provable. A rule with an empty set of premises is an axiom; that is, its denominator is always provable.

Example 10.4 (Static type inference) For example, program

$$(\lambda x . x 1)(\lambda y . y)$$

is typed *int* because we can prove

$$\emptyset \vdash (\lambda x . x 1)(\lambda y . y) : \text{int}$$

as follows:

$$\frac{\begin{array}{c} x : \text{int} \rightarrow \text{int} \in \{x : \text{int} \rightarrow \text{int}\} \\ \{x : \text{int} \rightarrow \text{int}\} \vdash x : \text{int} \rightarrow \text{int} \end{array}}{\{x : \text{int} \rightarrow \text{int}\} \vdash x : \text{int} \rightarrow \text{int}} \quad \frac{\begin{array}{c} x : \text{int} \rightarrow \text{int} \in \{x : \text{int} \rightarrow \text{int}\} \\ \{x : \text{int} \rightarrow \text{int}\} \vdash 1 : \text{int} \end{array}}{\{x : \text{int} \rightarrow \text{int}\} \vdash 1 : \text{int}}$$

$$\frac{\begin{array}{c} \{x : \text{int} \rightarrow \text{int}\} \vdash x 1 : \text{int} \\ \emptyset \vdash \lambda x . x 1 : (\text{int} \rightarrow \text{int}) \rightarrow \text{int} \end{array}}{\emptyset \vdash (\lambda x . x 1)(\lambda y . y) : \text{int}}$$

$$\frac{y : \text{int} \in \{y : \text{int}\}}{\{y : \text{int}\} \vdash y : \text{int}}$$

$$\frac{\{y : \text{int}\} \vdash y : \text{int}}{\emptyset \vdash \lambda y . y : \text{int} \rightarrow \text{int}}$$

Note that each step of the above proof is an instance of one of the proof rules in figure 10.4.

The proof rules of figure 10.4 are sound, as indicated by the following.

Theorem 10.1 (Soundness of the proof rules) Let E be a program, an expression without free variables. If $\emptyset \vdash E : \tau$, then the program runs without a type error and returns a value of type τ if it terminates.

The soundness proof is done by proving two facts, upon defining the program execution in the transitional style as a sequence of steps. The first proof (called *progress lemma*) shows that a typed program, unless it is a value, can always progress one step. The second proof (called *preservation lemma*) shows that a typed program after its one-step progress still has the same type. By these two facts, a typed program runs without a type error and returns a value of its type if it terminates. The readers can refer to [91] for proofs.

Off-line Algorithm The simple type system has a faithful and efficient algorithm for its proof construction. The algorithm is faithful: it succeeds for a judgment if and only if the judgment is provable in the simple type system. The algorithm is efficient: it uses a special operator called *unification* that has no iterative computations (or just a single iteration) as in the fixpoint algorithms of the general frameworks in chapters 3 and 4.

The static typing algorithm can be understood as two steps. First, by scanning the input program, we collect equations about the types of every subexpression of the program. Then we solve the equations by the unification procedure. This off-line algorithm can be made online where we solve the type equations while we scan the program.

The following procedure V collects type equations of the form $\tau \doteq \tau$ from the input program. Types in type equations contain type variables α (unknowns) ($\tau ::= \alpha \mid \text{int} \mid \tau \rightarrow \tau$). Given a program e , an expression without a free variable, the type equations are collected by calling a function $V(\emptyset, E, \alpha)$ with the empty type environment \emptyset and a fresh type variable α . Procedure $V(\Gamma, e, \tau)$ returns a set of type equations that must hold in order for the e expression to have type τ under assumption Γ :

$$\begin{aligned} V(\Gamma, n, \tau) &= \{\tau \doteq \text{int}\} \\ V(\Gamma, x, \tau) &= \{\tau \doteq \Gamma(x)\} \\ V(\Gamma, \lambda x. E, \tau) &= \{\tau \doteq \alpha_1 \rightarrow \alpha_2\} \cup V(\Gamma + x : \alpha_1, E, \alpha_2) \quad (\text{new } \alpha_1, \alpha_2) \\ V(\Gamma, E_1 E_2, \tau) &= V(\Gamma, E_1, \alpha \rightarrow \tau) \cup V(\Gamma, E_2, \alpha) \quad (\text{new } \alpha) \end{aligned}$$

Solving the type equations from V is equivalent to proving the given program in the simple type system, as shown by the following theorem.

Theorem 10.2 (Correctness of type equations) *Let E be a program (an expression without a free variable), and let α be a fresh type variable. S is a solution for the collection $V(\emptyset, E, \alpha)$ of type equations if and only if judgment $\emptyset \vdash E : S\alpha$ is provable.*

Now, the equations are solved by the unification procedure [98]. The unification procedure *unify*, given two types τ_1 and τ_2 of a type equation $\tau_1 \doteq \tau_2$, finds a substitution S (a finite map from type variables to types) whenever possible that makes both sides of the equation literally the same: $S\tau_1 = S\tau_2$. The following *unify* indeed finds the least solution: the most general unifying substitution. The concatenation S_2S_1 of two substitutions denotes a substitution that applies S_1 and then S_2 : $(S_2S_1)\tau = S_2(S_1\tau)$.

Procedure $\text{unify}(\tau_1, \tau_2)$ is defined as follows, to do the first match from the top:

$$\begin{aligned}
\text{unify}(\alpha, \tau) &= \{\alpha \mapsto \tau\} && \text{if } \alpha \notin \tau \\
\text{unify}(\tau, \alpha) &= \{\alpha \mapsto \tau\} && \text{if } \alpha \notin \tau \\
\text{unify}(\tau_1 \rightarrow \tau_2, \tau'_1 \rightarrow \tau'_2) &= \text{let } S_1 = \text{unify}(\tau_1, \tau'_1) \\
&\quad S_2 = \text{unify}(S_1 \tau_2, S_1 \tau'_2) \\
&\quad \text{in } S_2 S_1 \\
\text{unify}(\tau, \tau') &= \text{failure} && \text{other cases}
\end{aligned}$$

Finding a substitution that satisfies all the equations in collection $\{\tau_1 \doteq \tau_2, \dots, \tau_k \doteq \tau_k\}$ is a simple accumulation of the substitution that unifies each type equation:

$$\begin{aligned}
\text{Solve}(\{\tau_1 \doteq \tau_2\}) &= \text{unify}(\tau_1, \tau_2) \\
\text{Solve}(\{\tau_1 \doteq \tau_2\} \cup \text{rest}) &= \text{let } S = \text{unify}(\tau_1, \tau_2) \\
&\quad \text{in } (\text{Solve}(\text{rest}))S,
\end{aligned}$$

where “ $S \text{ rest}$ ” denotes new type equations after applying the S substitution to every type variable in rest .

For a program E the simple type inference is thus

$$\text{Solve}(V(\emptyset, E, \alpha)).$$

Online Algorithm The above off-line algorithm that first collects equations and then solves them can be rephrased as an online algorithm. For a program E the online simple type inference is

$$M(\emptyset, E, \alpha),$$

where

$$\begin{aligned}
M(\Gamma, n, \tau) &= \text{unify}(\tau, \text{int}) \\
M(\Gamma, x, \tau) &= \text{unify}(\tau, \Gamma(x)) \\
M(\Gamma, \lambda x. E, \tau) &= \text{let } S_1 = \text{unify}(\tau, \alpha_1 \rightarrow \alpha_2) \quad (\text{new } \alpha_1, \alpha_2) \\
&\quad S_2 = M(S_1 \Gamma + x : S_1 \alpha_1, E, S_1 \alpha_2) \\
&\quad \text{in } S_2 S_1 \\
M(\Gamma, E_1 E_2, \tau) &= \text{let } S_1 = M(\Gamma, E_1, \alpha \rightarrow \tau) \quad (\text{new } \alpha) \\
&\quad S_2 = M(S_1 \Gamma, E_2, S_1 \alpha) \\
&\quad \text{in } S_2 S_1.
\end{aligned}$$

The above algorithm is easy to follow since the call to $M(\Gamma, E, \tau)$ computes solutions in order for expression E to have type τ under type assumption Γ for the free variables of E .

Running the M algorithm is equivalent to constructing a proof in the type system, as follows.

Theorem 10.3 (Correctness of the M algorithm) *Let E be a program (an expression without a free variable), and let α be a fresh type variable. If $M(\emptyset, E, \alpha) = S$, then $\vdash E : S\alpha$. Conversely, if $\emptyset \vdash E : \tau$, then $M(\emptyset \vdash E, \alpha) = S_1$ and $\tau = (S_2 S_1)\alpha$ for an additional substitution S_2 .*

Polymorphic Type Inference Recall that, from the static analysis point of view, the type inference of the sound simple type system over-approximates the set of values of expressions of the input program. From the soundness point of view, if the analysis succeeds in typing the input program, then we can safely conclude that the program will run without a type error. On the other hand, if the analysis fails to type the input program, that means we don't know whether the program will have a type error or not.

The accuracy of the simple type system can be improved by controlling the degree of the over-approximation so that the number of fail cases for type-safe programs should be reduced.

A more accurate system than the simple type system is one called *polymorphic* type system. The polymorphic type system reduces the failure cases for type-safe programs but is still sound: when the type inference succeeds, the input program will run without a type error.

The set of polymorphic types is a finer abstract domain than that of the simple types. The abstract domain is refined because there exist elements (polymorphic types) that represent special sets of values that are not precisely singled out as a separate abstract element in simple types. For example, a polymorphic function type can represent a set of functions that can run regardless of the types of arguments. For example, a polymorphic type, written as

$$\forall \alpha . \alpha \rightarrow \text{int},$$

denotes a set of functions that, whatever the type of the actual argument, run without a type error and return an integer if they terminate.

One polymorphic type inference system is called *let-polymorphic* type system [91]. This polymorphic type system, which is common in ML-like

programming languages, has, as for the simple type system, sound and complete algorithms [74] using the unification procedure. The let-polymorphic type system always computes *principal types* among multiple polymorphic type candidates for each expression. From the static analysis point of view, when principal types exist, it means the existence of best abstractions in their abstract domains.

During typing, a function that can behave independently of its argument types has a generalized, polymorphic types. At different call of the polymorphic function, the polymorphic type of the function is instantiated into an appropriate type. For a function, the polymorphic generalization, if possible, and the distinct instantiations at different calls to the function are analogous to context-sensitive analysis that analyzes functions differently at different call sites.

Limitation For target programming languages that lack a sound static type system, we have to invent it. We have to design a finite proof system and prove its soundness. Then we have to find its algorithm and prove its soundness. The burden grows if the algorithm has to solve some constraints that turn out to be unsolvable by the unification procedure.

For languages like ML that already has a sound let-polymorphic static type system, we can use its sound type system to carry an extra entity (elements of extra abstract domains) of interest other than just the conventional type [90]. However, again, if the system derives some constraints that are unsolvable by the unification procedure, it is our burden to invent a new algorithm and prove its soundness from scratch.

11 Summary and Perspectives

We hope that, at this point, readers have been able to build up a solid knowledge of the foundations of static analysis. While we cannot cover the topic in an exhaustive manner, we consider that the very first steps one takes in a new field are often the hardest, and we intend this book to help readers at that stage. This quick conclusion summarizes the main concepts, provides some perspectives on how to continue the learning of static analysis, and discusses some static analysis challenges ahead.

Summary The design, the choice, and the configuration of a static analysis may seem hard at first, but they should be guided with a general methodology.

First, the starting point of any formal static analysis has to be a clear *concrete semantics*, which expresses in a mathematical way how programs run. The importance of this semantics is crucial, as it fixes the foundations of the design and proof of any static analysis. The semantics has to reflect perfectly well how target programs are intended to be run.

Second, the careful definition of the *target property* is also a very important step. Not only does it formalize what one intends to compute, prove, or verify, but it also guides many choices in the design of the analysis. First of all, it should be possible to express it with respect to the concrete semantics used as a reference. Moreover, it should also specify what approximation may be done during the analysis.

Third, the choice of an *abstraction* to use for the analysis is crucial. In the vast design space of an abstraction, if the target programs of the analysis are restricted, one has to use the restricted class as a beacon to zoom in on an abstraction that is effective, at least for the programming idioms of those target programs. The abstractions of the semantic domains and the semantic functions mostly dictate the effectiveness of the resulting analysis. In

general, one has to be well aware of which conservative approximations are made by the abstraction, so as to understand the limitations of the resulting analysis and to be able to remedy them. The abstraction fixes the logical properties that the analysis will be able to use to check the target property and to compute all the required invariants. It also defines their representation and greatly impacts the automatic reasoning algorithms. The abstraction should in general always be expressive enough to write down a paper proof of the property of interest. It should also be as lightweight as possible to enable the quick computation of abstract invariants for the set of target programs.

Finally, static analysis algorithms such as the implementation of the transfer functions, global worklist iterations, and widening operations define the way the analysis proceeds. Many choices are left at this stage, too, for example, the choice of the order of works in the work list and the choice of widening operations. In the implementation, one has to engineer the algorithms one step further to strike a cost-effective balance, minding the soundness that is prescribed in the analysis design.

Perspectives At this point, we would like to suggest a few routes for readers to pursue their journey through the land of static analysis.

First, many topics were not covered in-depth in this book, as they would often require a book of their own to fully address them. Thus, we provide bibliographic references for many such topics. These references should be used as an entry point toward more specialized content. A good reference would be [25], which provides a deep insight in the foundations of abstract interpretation.

Second, readers who are interested in the implementation or in the practical use of static analyzers should consider starting up their own implementation or playing with some of the many existing open-source static analysis tools. While this is a time-consuming activity, it is also often a very rewarding way to acquire solid knowledge and experience in the field.

Finally, readers interested in research should also see plenty of opportunities, as static analysis is still a topic full of open problems. There are still few general and solid results on the precision or on the scalability of static analysis tools. Emerging programming languages sometimes lack a

clean semantics or provide very expressive and complex features, so they challenge existing standard static analysis techniques. Hybrid systems combine discrete and continuous computations, which makes them hard to reason about. Machine learning techniques based on probabilistic or connectionist approaches sometimes lack a clear semantics or generate software at low-level; hence, they challenge language-centric static analysis techniques. Security properties are often hard to prove or even formalize; hence, their verification by static analysis remains very challenging.

A Reference for Mathematical Notions and Notations

This appendix is meant to be used as a reference for elements of mathematics and general notations. It does not substitute for a course in discrete mathematics and aims only to make this book self-contained.

A.1 Sets

In this book, we use *sets* abundantly, as in standard sets theory. Informally, a set is a collection of elements that could be finite (as the set of 32-bit machine integers) or infinite (as the set of non-negative natural integers, denoted by \mathbb{N} , or the set of integers, denoted by \mathbb{Z}). When x is an element of a set E , we write $x \in E$. Moreover, when all elements of a set E belong to a set F , we say that E is included into F , and we write $E \subseteq F$.

We use the following notations to construct sets explicitly:

- \emptyset designates the *empty set*;
- $\{x_0, \dots, x_n\}$ designates the set that contains exactly the elements x_0, \dots, x_n ;
- $\{x \in E \mid P(x)\}$ is the set of all the elements of E that satisfy the property expressed by the formula P .

Given two sets E and F , we define their *union* $E \cup F$ as the set that collects all the elements that belong to either E or F ; using the above notation, it boils down to $\{x \mid x \in E \text{ or } x \in F\}$. Similarly, their *intersection* is the set of the elements that belong to both E and F and is denoted by $E \cap F$. The set difference $E \setminus F$ is the set of all elements in E that do not appear in F . Last, when E and F are disjoint (their intersection is empty), we also denote their union by $E \uplus F$ (in other words, when we refer to $E \uplus F$, we implicitly state that E and F are disjoint).

The *Cartesian product* of two sets E and F is denoted by $E \times F$ and collects the pairs made of an element of E and of an element of F ; such a pair is denoted by (x, y) (where $x \in E$ and $y \in F$). It generalizes to arbitrary numbers of sets.

Sets can be also used as elements; thus, we can also construct sets of sets, which is very often done in static analysis. Given a set E , the set of all the subsets of E is called the *powerset* of E and is denoted by $\wp(E)$. As an example, the powerset of $\{0, 1\}$ is made of the four elements \emptyset , $\{0\}$, $\{1\}$, and $\{0, 1\}$.

Union and intersection generalize to sets of sets. For instance, we often use $\bigcup \{E \in F \mid P(E)\}$ to join all the subsets of F that satisfy a property P . Similarly, the following notation designates a union of sets indexed by natural integers:

$$\bigcup_{i \in \mathbb{N}} E_i$$

A.2 Logical Connectives

The following connectives are used to construct logical formulas:

- If P and Q are logical propositions, we denote the conjunction of P and Q by $P \wedge Q$, namely, a proposition that holds if and only if both P and Q hold.
- Similarly, we denote the disjunction of P and Q by $P \vee Q$.
- Last, $P \Rightarrow Q$ stands for the logical implication that states that if P holds, then so does Q .

Moreover, quantifiers express that some proposition holds for some or all the elements of a set. If E is a set, and if P is a logical proposition parameterized by an element, then $\forall x \in E, P(x)$ is the proposition that states that $P(x)$ holds for *all* the elements of E . Moreover, $\exists x \in E, P(x)$ is the proposition that states that there exists at least one element of E that satisfies P (it implies that E is not empty). When the set over which we quantify is obvious, it is sometimes omitted.

A.3 Definitions and Proofs by Induction

In this book, we often consider inductively defined data types (e.g., to describe expressions, or commands...) or inductively defined properties (e.g., the semantics of a loop or of a transition system). As an example, in chapter 3 we define expressions by induction. More precisely, we define them using the grammar $E ::= n \mid x \mid E \odot E$, which means that an expression is either a constant, or a variable, or a binary operator applied to a pair of expressions.

Such inductive definitions naturally call for proofs also done by induction; thus, we recall the principle underlying such proofs. First, let us consider the case of Peano integers (i.e., non-negative mathematical integers). We let P be a unary predicate over integers, and write $P(n)$ when P holds over integer n . The *integer induction* proof principle reduces the proof that all integers satisfy P (i.e., $\forall n \in \mathbb{N}, P(n)$) to the proof of the two following properties:

1. $P(0)$ holds;
2. for all integers n , if $P(n)$ holds then so does $P(n + 1)$.

Intuitively, to justify $P(3)$, one simply needs to remark that $P(0)$ holds by the first point and to apply the second point to 0 to derive $P(1)$, then to 1 to derive $P(2)$, and finally to 2 to derive $P(3)$. The above induction principle generalizes this intuition to all integers. In fact, once the above two points are proved, we can immediately derive that P holds for any integer.

This principle generalizes to all data types defined inductively, such as program expressions as defined above: to prove a property over expressions, one simply needs to do so for constant values, for variables, and then for expressions made of a binary operator, under the assumption that the property holds for the sub-expressions.

A.4 Functions

A *function* is a mathematical object that captures a computation by associating each element of a set (its domain) to an element of another set (its codomain). When f is a function with domain E and co-domain F , we denote it by $f : E \rightarrow F$, and we write $f : x \mapsto e$ (or $f : (x \in E) \mapsto e$), where the expression e defines how the image of x is computed. For instance, the function $f : \mathbb{Z} \rightarrow \mathbb{Z}, x \mapsto x + 1$ maps each integer to its successor. The identity function over a set E is the function noted id that maps each element to itself.

Given two functions $f : E \rightarrow F$ and $F \rightarrow G$, we write $g \circ f$ for their *composition*, which is computed by applying f first and then g . If $f : E \rightarrow E$, we write f^n for the composition of f with itself n times (if $n = 0$, it is the identity function).

A function $f : \mathbb{N} \rightarrow E$ (i.e., that maps integers to elements of the set E) is called a sequence of elements of E . It is often denoted by $(f(n))_{n \in \mathbb{N}}$.

We let $B = \{\text{true}, \text{false}\}$ denote the set of Booleans. Then, given a set E , a function $f : E \rightarrow B$ uniquely defines a set of elements of E . This set can be defined as $\{x \in E \mid f(x) = \text{true}\}$. This principle also defines an extension of sets called *multisets*. Intuitively, a multiset is a set where each element is annotated by a positive integer, which describes the number of occurrences of that element. A multiset M the elements of which are taken from some set E boils down to a function $f : E \rightarrow \mathbb{N}$: if x is an element of E , x belongs to M if and only if $f(x) > 0$; furthermore, $f(x)$ denotes the number of occurrences of x in M .

A.5 Order Relations and Ordered Sets

In this section, we assume that a set E is given. Intuitively, an *order relation* over E is a mathematical object that specifies for which pairs x, y of elements of E we can say that x is “smaller” than y for some given notion of “smaller.” Formally, an order relation is a binary relation \leq over E that is reflexive (\leq is *reflexive* if and only if $\forall x \in E, x \leq x$), transitive (\leq is *transitive* if and only if $\forall x, y, z \in E, x \leq y$ and $y \leq z \Rightarrow x \leq z$), and anti-symmetric (\leq is *anti-symmetric* if and only if $\forall x, y \in E, x \leq y$ and $y \leq x \Rightarrow x = y$). The ordering over natural integers is an obvious example of order relation. The lexicographic order used to sort words in a dictionary or index provides a more complex example of order relation. Another classic example is set inclusion, which defines an order relation over the powerset of any set.

An order relation is *total* when any pair of elements can be compared (i.e., if $\forall x, y \in E, x \leq y \vee y \leq x$). While it is the case for the standard order over integers, it is often not the case in this book. As a matter of fact, inclusion is not a total order since some pairs of elements such as $\{0\}, \{1\}$ cannot be compared. A *chain* is a subset of E that is a total ordering. As an example, $\emptyset, \{0\}, \{0, 2\}$ form a chain for the inclusion order.

In the following, we assume that \leq defines an order relation over E . If there exists an element $x_0 \in E$ that is smaller than any other element of E for \leq , we call x_0 the *infimum*; to distinguish it, we usually denote it by \perp (read “bottom”). Similarly, the *supremum*, if it exists, is the element that is greater than any other element; it is usually denoted by \top (read “top”).

Let x and y be two elements of E . When it exists, we call *least upper bound* of x and y the element $x \sqcup y$ that is greater than x and y and is smaller than any other element with that property. While the existence of the least upper bound does not always hold, when it exists it is unique. Dually, the *greatest lower bound*, if it exists, is the element $x \sqcap y$ that is smaller than x and y and is greater than any other element with that property. These bounds generalize to families of elements. In the case of the powerset, the least upper bound is simply the set union, and the greatest lower bound is the set intersection.

We say that an ordered set E is a *lattice* when it has an infimum and a supremum, and when each pair of elements has both a least upper bound and a greatest lower bound. Moreover, when any subset of E has both a least upper bound and a greatest lower bound, it is called a *complete lattice*. As an example, the powerset of any set is a complete lattice.

Additionally, we say that E is a *complete partial order* (for short, CPO) if it has an infimum and is such that any chain of elements of E has a least upper bound in E .

Last, let us assume two ordered sets E and F (for simplicity we denote the order relations over both sets by \leqslant) and a function $f : E \rightarrow F$. We say that f is *monotone* if and only if, for all $x, y \in E$ such that $x \leqslant y$, we have $f(x) \leqslant f(y)$. A stronger property is *continuity*: assuming that E and F are CPOs, we say that f is continuous if and only if the image of any chain G of E by f has a least upper bound, that is, such that $\sqcup\{f(x) \mid x \in G\} = f(\sqcup G)$. It is very easy to show that, when a function is continuous, it is also monotone. We say that f is *extensive* if and only if, for all $x \in E$, we have $x \leqslant f(x)$.

A.6 Operators over Ordered Structures and Fixpoints

In this section, we assume a set E , an order relation \leqslant and a function $f : E \rightarrow E$.

We call *fixpoint* of f an element x such that $f(x) = x$. Moreover, when it exists, the *least fixpoint* of f is the fixpoint of f that is smaller for \leqslant than any other fixpoint of f ; it is denoted by $\mathbf{lfp} f$. Fixpoints are useful to define interesting elements and occur naturally in static analysis.

In general, the existence of a fixpoint, let alone, of a least fixpoint is not guaranteed. However, there exist theorems under which one can establish the existence of a least fixpoint, and even provide a constructive definition, that is naturally very close to an algorithm to compute it. In this book, we use the following such theorem:

Theorem A.1 (Kleene's fixpoint theorem) *If f is continuous, and if E is a CPO with infimum \perp , then f has a least fixpoint, that can be expressed as follows:*

$$\mathbf{lfp} f = \bigcup_{n \in \mathbb{N}} f^n(\perp)$$

This theorem underlines the strong link between a least fixpoint and an inductive definition/computation.

For the interested reader, we now show the proof of this theorem. In the next few paragraphs, we assume that E is a CPO.

Proof of Kleene's Fixpoint Theorem First, we need to justify the existence of the least upper bound in the right-hand side of the equality. As \perp is the infimum of E , we have $\perp \leqslant f(\perp)$. Since f is continuous, it is also monotone; thus, we can prove by induction that, for all $n \in \mathbb{N}$, we have $f^n(\perp) \leqslant f^{n+1}(\perp)$. Therefore $\{f^n(\perp) \mid n \in \mathbb{N}\}$ forms a chain. As E is a CPO, it has a least upper bound, which we denote by X .

Since the set of iterates of f from \perp is a chain, we can also apply to it the continuity of f . This allows us to derive that $\{f^{n+1}(\perp) \mid n \in \mathbb{N}\}$ has a least upper bound that is equal to $f(X)$:

$$f(X) = f\left(\bigcup_{n \in \mathbb{N}} f^n(\perp)\right) = \bigcup_{n \in \mathbb{N}} f^{n+1}(\perp) = \perp \sqcup \bigcup_{n \in \mathbb{N}} f^{n+1}(\perp) = \bigcup_{n \in \mathbb{N}} f^n(\perp) = X$$

We have proved that X is a fixpoint for f . To conclude, we simply need to prove that it is the least one, that is, that any fixpoint of f is greater than it. Let us assume that X' is another fixpoint. Then we can prove by induction over n that $f^n(\perp) \leqslant X'$, which is easy: it holds at rank 0 by definition of the infimum, and the property at rank n implies the property at rank $n + 1$ thanks to monotonicity and to the fact that X' is a fixpoint. Thus, by definition of the least upper bound,

$$\bigcup_{n \in \mathbb{N}} f^n(\perp) \preceq X'$$

This concludes the proof.

B Proofs of Soundness

This appendix contains proofs for the theorems of chapters 3 and 4.

B.1 Properties of Galois Connections

In this section, we prove the fundamental properties of Galois connections listed in section 3.2.1.

Theorem B.1 (Properties of Galois connection) *Let (C, \sqsubseteq) and (A, \sqsubseteq) be, respectively, a concrete domain and an abstract domain with an abstraction function α and a concretization function γ that form a Galois connection (definition 3.5)*

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

Then the following properties hold:

- $\forall c \in C, c \sqsubseteq \gamma(\alpha(c))$ (or, equivalently, $\text{id} \sqsubseteq \gamma \circ \alpha$);
- $\forall a \in A, \alpha(\gamma(a)) \sqsubseteq a$ (or, equivalently, $\alpha \circ \gamma \sqsubseteq \text{id}$);
- γ and α are monotone; and
- if both C and A are CPOs (any chain has a least upper bound), then α is continuous.

Proof We assume the existence of the Galois connection and prove items one by one:

- $\text{id} \sqsubseteq \gamma \circ \alpha$: Let c be a concrete element. By the reflexivity of the order relation \sqsubseteq , we have $\alpha(c) \sqsubseteq \alpha(c)$. Thus, by the definition of Galois connections, $c \sqsubseteq \gamma(\alpha(c))$.
- $\alpha \circ \gamma \sqsubseteq \text{id}$: Let a be an abstract element. By the reflexivity of the order relation \sqsubseteq , we have $\gamma(a) \sqsubseteq \gamma(a)$. Thus, by the definition of Galois connections, $\alpha(\gamma(a)) \sqsubseteq a$.
- γ is monotone: Let us assume that c_0, c_1 are concrete elements such that $c_0 \sqsubseteq c_1$. Then, because $\text{id} \sqsubseteq \gamma \circ \alpha$, $c_0 \sqsubseteq c_1 \sqsubseteq \gamma(\alpha(c_1))$. Hence, by the definition of Galois connections, $\alpha(c_0) \sqsubseteq \alpha(c_1)$.
- α is monotone: Let us assume that a_0, a_1 are abstract elements such that $a_0 \sqsubseteq a_1$. Then, because $\alpha \circ \gamma \sqsubseteq \text{id}$, $\alpha(\gamma(a_0)) \sqsubseteq a_1 \sqsubseteq a_1$. Hence, by the definition of Galois connections, $\gamma(a_0) \sqsubseteq \gamma(a_1)$.
- α is continuous: Let S be a chain in C . Since C is a CPO, the least upper bound $\bigcup_{c \in S} c$ exists in C . Since α is monotone, for all elements x in the chain S , we have $\alpha(x) \sqsubseteq \alpha(\bigcup_{c \in S} c)$, and thus $\bigcup_{c \in S} \alpha(c) \sqsubseteq \alpha(\bigcup_{c \in S} c)$. The function $\gamma \circ \alpha$ is monotone, so it transforms a chain into a chain; thus, $\{\gamma(\alpha(c)) \mid c \in S\}$ is a chain and has its least upper bound in CPO C . Moreover, $\text{id} \sqsubseteq \gamma \circ \alpha$; thus,

$$\bigcup_{c \in S} c \sqsubseteq \bigcup_{c \in S} \gamma(\alpha(c)).$$

The function γ is monotone, and for all element c in the chain S , we have $\alpha(c) \sqsubseteq \sqcup_{c \in S} \alpha(c)$ and thus $\gamma(\alpha(c)) \subseteq \gamma(\sqcup_{c \in S} \alpha(c))$. Therefore,

$$\bigcup_{c \in S} \gamma(\alpha(c)) \subseteq \gamma\left(\bigcup_{c \in S} \alpha(c)\right).$$

As we compose the two above inequalities, we get

$$\bigcup_{c \in S} c \subseteq \gamma\left(\bigcup_{c \in S} \alpha(c)\right).$$

By the definition of Galois connections, this entails that $\alpha(\bigcup_{c \in S} c) \sqsubseteq \sqcup_{c \in S} \alpha(c)$. Hence, $\alpha(\bigcup_{c \in S} c) = \sqcup_{c \in S} \alpha(c)$. This concludes the proof.

B.2 Proofs of Soundness for Chapter 3

This section shows how to establish the soundness of the basic abstract interpreter, such as the analysis presented in chapter 3, and the principles of this proof extend to many other static analyzers. Each of the following subsections formalizes the proof of one of the main theorems of chapter 3. All together, these theorems establish the soundness of a non-relational abstract interpreter designed in compositional style (i.e., based on a compositional semantics), under the assumption that the operations of the value domain are sound (as defined in section 3.3).

B.2.1 Soundness of the Abstract Interpretation of Expressions

In this section, we consider theorem 3.2, which we recall first.

Theorem B.2 (Soundness of the abstract interpretation of expressions) *For all expressions E , for all non-relational abstract elements $M^\#$, and for all memory states m such that $m \in \gamma(M^\#)$,*

$$\llbracket E \rrbracket (m) \in \gamma_V(\llbracket E \rrbracket^\#(M^\#))$$

Proof The proof proceeds by induction over the structure of expressions. Indeed, expressions were defined by induction, so we can apply the induction proof principle shown in appendix A.3, to prove this property for all expressions. To do this, we consider each kind of expression and prove each case under the assumption that the property holds for all sub-expressions, as follows.

- Case of constant expressions: We assume that E is the constant expression defined by the value n . Then $\llbracket E \rrbracket (m) = n$, and $\llbracket E \rrbracket^\#(M^\#) = \phi_V(n)$. By definition of the operation ϕ_V of the value abstract domain (as stated in section 3.3.1), $n \in \gamma_V(\phi_V(n))$, which concludes this case.
- Case of expressions made of a variable: We assume that E is the expression made of the reading of variable x . Then $\llbracket E \rrbracket (m) = m(x)$, and $\llbracket E \rrbracket^\#(M^\#) = M^\#(x)$. By assumption, $m \in \gamma(M^\#)$; thus, $m(x) \in \gamma_V(M^\#(x))$, which concludes this case.
- Case of expressions made of a binary operator applied to two sub-expressions: We assume that E is of the form $E_0 \odot E_1$, where E_0 and E_1 are sub-expressions and \odot is a binary operator. We assume that the theorem holds for E_0 and E_1 since we are carrying out the proof by induction over the structure of expressions. Therefore, the inductive

hypothesis entails that, for all $i \in \{0, 1\}$, $\llbracket E_i \rrbracket(m) \in \gamma_V(\llbracket E_i \rrbracket \#(M^\#))$. Then $\llbracket E \rrbracket(m) = f_\odot(\llbracket E_0 \rrbracket(m), \llbracket E_1 \rrbracket(m))$ and $\llbracket E \rrbracket \#(M^\#) = f_\odot^\#(\llbracket E_0 \rrbracket \#(M^\#), \llbracket E_1 \rrbracket \#(M^\#))$. By the induction hypothesis and by definition of the soundness of the operation of the value abstract domain $f_\odot^\#$ (as stated in section 3.3.1), we have $f_\odot(\llbracket E_0 \rrbracket(m), \llbracket E_1 \rrbracket(m)) \in \gamma_V(f_\odot^\#(\llbracket E_0 \rrbracket \#(M^\#), \llbracket E_1 \rrbracket \#(M^\#)))$. This concludes the proof of this case.

Comments This proof actually shows very well how the analysis progresses, and it maintains soundness step-by-step over the course of the analysis of expressions: the inductive cases rely on operations that over-approximate the effect of each concrete program execution step; moreover, the base cases are also designed so as to start with a sound over-approximation. For the representation of constants and for binary operations, we have relied on the soundness of operations in the underlying value abstract domain. This is actually a key feature of the approach followed in chapters 3 and 4: the soundness of these operations can be proved separately for each value domain that we may use in the actual analysis; thus, we get the same benefit in terms of proof modularity as we do for the modularity of the implementation in chapter 7.

Because of this dependency, one may argue that a large part of the proof needs to be done at the value abstract domain level. This is indeed correct, and this part of the proof heavily depends on the representation of the abstract values and on the algorithms to manipulate them. We simply provide a few examples, and general guidelines:

- When the abstract lattice of the value abstract domain is finite, the operations can be proved correct by straightforward case analysis.
- When there are several kinds of abstract values, case analysis is often helpful and/or necessary; for instance, in the case of the interval domain, the soundness of binary operations can be proved by considering separately the case of \perp and non- \perp abstract values.
- Basic cases can often be proved by simple arithmetic arguments, for example, based on interval arithmetics.

B.2.2 Soundness of the Abstract Interpretation of Conditions

We now consider the soundness theorem for the analysis of condition expressions (theorem 3.3).

Theorem B.3 (Soundness of the abstract interpretation of conditions) *For all expressions B , for all non-relational abstract elements $M^\#$, and for all memory states m such that $m \in \gamma(M^\#)$,*

$$\text{if } \llbracket B \rrbracket(m) = \mathbf{true}, \text{ then } m \in \gamma(\mathcal{F}_B^\#(M^\#))$$

Proof Let B be a condition expression. Let $M^\#$ be an abstract state, and let $m \in \gamma(M^\#)$, such that $\llbracket B \rrbracket(m) = \mathbf{true}$. By definition, the operation $\mathcal{F}_B^\#$ of the value abstract domain is assumed to be sound; thus, $\mathcal{F}_B(\gamma(M^\#)) \subseteq \gamma(\mathcal{F}_B^\#(M^\#))$, where $\mathcal{F}_B(M) = \{m \in M \mid \llbracket B \rrbracket(m) = \mathbf{true}\}$. Since $\llbracket B \rrbracket(m) = \mathbf{true}$, m belongs to $\mathcal{F}_B(M)$. This concludes the proof.

Comments The proof heavily relies on the soundness of the filter operation, just like the proof of the soundness of the abstract interpretation of expressions in appendix B.2.1 relies on the operations of the underlying value abstract domain.

B.2.3 Soundness of the Abstract Join Operator

In this section, we show the proof of theorem 3.4, which we recall here.

Theorem B.4 (Soundness of abstract join) *Let M_0^\sharp and M_1^\sharp be two abstract states. Then, the following inclusion holds.*

$$\gamma(M_0^\sharp) \cup \gamma(M_1^\sharp) \subseteq \gamma(M_0^\sharp \sqcup^\# M_1^\sharp)$$

Proof We prove only that $\gamma(M_0^\sharp) \subseteq \gamma(M_0^\sharp \sqcup^\# M_1^\sharp)$, as the inequality $\gamma(M_1^\sharp) \subseteq \gamma(M_0^\sharp \sqcup^\# M_1^\sharp)$ is symmetric and can be proved similarly. Let $m \in \gamma(M_0^\sharp)$. To prove that $m \in \gamma(M_0^\sharp \sqcup^\# M_1^\sharp)$, we need to establish that, for any variable x , we have $m(x) \in \gamma_\gamma((M_0^\sharp \sqcup^\# M_1^\sharp)(x))$. By definition of $\sqcup^\#$, $(M_0^\sharp \sqcup^\# M_1^\sharp)(x) = M_0^\sharp(x) \sqcup_\gamma^\# M_1^\sharp(x)$. The soundness of $\sqcup_\gamma^\#$ guarantees that $m(x) \in \gamma_\gamma(M_0^\sharp(x) \sqcup_\gamma^\# M_1^\sharp(x))$, which concludes the proof.

Comments As in the previous subsections, the soundness of the operation in the non-relational domain follows from that of the operation $\sqcup_\gamma^\#$ in the value abstract domain. This operation can usually be proved sound by case analysis over the elements of the value abstract domain and/or basic arithmetic reasoning.

B.2.4 Abstract Iterates with Widening

In this section, we prove theorem 3.5.

Theorem B.5 (Abstract iterates with widening) *We assume that ∇ is a widening operator over the non-relational abstract A domain and that F^\sharp is a function from A to itself. Then the algorithm below terminates and returns an abstract element M_{\lim}^\sharp :*

```
abs_iter( $F^\sharp, M^\sharp$ ) ::=  
  R  $\leftarrow M^\sharp$ ;  
  repeat   T  $\leftarrow R$ ; R  $\leftarrow R \nabla F^\sharp(R)$ ;  until R = T  
  return  $M_{\lim}^\sharp = T$ ;
```

Let us denote the abstract element that it returns by M_{\lim}^\sharp .

Moreover, if we assume that $F : M \rightarrow M$ is continuous and is such that $F \circ \gamma \subseteq \gamma \circ F^\sharp$ for the pointwise inclusion (which means that, for all abstract element M_0^\sharp , $F \circ \gamma(M_0^\sharp) \subseteq \gamma \circ F^\sharp(M_0^\sharp)$), then the following inclusion holds.

$$\bigcup_{i \geq 0} F^i(\gamma(M^\sharp)) \subseteq \gamma(M_{\lim}^\sharp)$$

Proof We first prove the termination part, which implies the existence of the limit element M_{\lim}^\sharp . Then we prove the soundness of the limit.

The loop in the algorithm is essentially computing the following sequence:

$$\begin{cases} M_0^\sharp &= M^\sharp \\ M_{n+1}^\sharp &= M_n^\sharp \nabla F^\sharp(M_n^\sharp) \end{cases}$$

Indeed, at the n th iteration of the loop body, T stores M_n^\sharp and R stores M_{n+1}^\sharp . By the definition of widening operators (definition 3.11), and since ∇ is assumed to be a widening, this sequence is ultimately stationary. This means that there exists a rank n for which two iterates are equal, so that the loop exit condition holds true. Thus, the algorithm terminates and M_{\lim}^\sharp is well defined.

We now consider soundness. To establish soundness, we first prove the following inclusion.

$$\bigcup_{i=0}^n F^i(\gamma(M_i^\sharp)) \subseteq \gamma(M_{\lim}^\sharp)$$

This can be proved by induction over n :

- If $n = 0$, then $M_0^\sharp = M^\sharp$, so both terms are equal to $\gamma(M^\sharp)$ and the property holds.
- Let us now assume that the property holds at rank n and prove the property at rank $n + 1$. Since F is monotone (because it is continuous), we remark that:

$$\begin{aligned} F^{n+1}(\gamma(M^\sharp)) &= F(F^n(\gamma(M^\sharp))) \\ &\subseteq F\left(\bigcup_{i=0}^n F^i(\gamma(M^\sharp))\right) \quad \text{as } F \text{ is monotone} \\ &\subseteq F(\gamma(M_n^\sharp)) \quad \text{as } F \text{ is monotone and by induction hypothesis} \\ &\subseteq \gamma(F^\sharp(M_n^\sharp)) \quad \text{as } F \circ \gamma \subseteq \gamma \circ F^\sharp \end{aligned}$$

By induction hypothesis, and since ∇ over-approximates set union, we derive that

$$\begin{aligned} \bigcup_{i=0}^{n+1} F^i(\gamma(M_i^\sharp)) &= \bigcup_{i=0}^n F^i(\gamma(M_i^\sharp)) \cup F^{n+1}(\gamma(M^\sharp)) \\ &\subseteq \gamma(M_n^\sharp) \cup \gamma(F^\sharp(M_n^\sharp)) \\ &\subseteq \gamma(M_n^\sharp \nabla F^\sharp(M_n^\sharp)) \\ &= \gamma(M_{n+1}^\sharp) \end{aligned}$$

We have proved above that the sequence of abstract iterates is ultimately stationary, which means that it eventually converges. We let N be the rank at which the limit is reached. The above inequality implies that, for any rank $k \geq N$,

$$\bigcup_{i=0}^k F^i(\gamma(M_i^\sharp)) \subseteq \gamma(M_{\lim}^\sharp)$$

Thus,

$$\bigcup_{i \geq 0} F^i(\gamma(M_i^\sharp)) \subseteq \gamma(M_{\lim}^\sharp)$$

This concludes the proof.

Comments This proof is quite representative of termination and soundness proofs for abstract iteration sequences: first, it establishes the convergence and the existence of the limit, based on the properties of widening; second, it ties the computation performed in the abstract level to the definition of the concrete semantics so as to prove soundness. The second part is actually quite similar to the other soundness proofs shown in this appendix, in the sense that it relies on the soundness of the operations it applies, namely F^\sharp and ∇ .

B.2.5 Soundness of the Abstract Interpretation of Commands

Finally, we prove the main soundness theorem, which states that the analysis of a program will always produce an over-approximation of the behaviors of the program (theorem 3.6).

Theorem B.6 (Soundness) For all commands C and all abstract states $M^\#$, the computation of $\llbracket C \rrbracket_{\mathcal{P}}^\#(M^\#)$ terminates, and

$$\llbracket C \rrbracket_{\mathcal{P}}(\gamma(M^\#)) \subseteq \gamma(\llbracket C \rrbracket_{\mathcal{P}}^\#(M^\#))$$

Proof Program commands are defined inductively, so the proof is also unsurprisingly carried out by induction over the syntax and follows the structure of the definitions of the concrete semantics (figure 3.4) and of the abstract semantics (figure 3.11). The case where $M^\# = \perp$ is handled separately, and the analysis then returns \perp ; this is obviously sound since the concrete semantics also maps $\gamma(\perp) = \emptyset$ to itself.

Let us now assume that $M^\# \neq \perp$ and consider each case of command:

- Case where C is a **skip** statement: Then $\llbracket C \rrbracket_{\mathcal{P}}(\gamma(M^\#)) = \gamma(M^\#) = \gamma(\llbracket C \rrbracket_{\mathcal{P}}^\#(M^\#))$, so the property trivially holds.
- Case where C is a sequence: We assume that the property holds for C_0 and C_1 and prove it for C . Under this assumption, theorem 3.1 applies and proves the property.
- Case where C is an assignment $x := E$: Let $m \in \gamma(M^\#)$. We need to prove that $m[x \mapsto \llbracket E \rrbracket(m)] \in \llbracket x := E \rrbracket_{\mathcal{P}}^\#(M^\#) = M^\#[x \mapsto \llbracket E \rrbracket^\#(M^\#)]$. By the soundness of the analysis of expressions (theorem 3.2), we derive that $\llbracket E \rrbracket(m) \in \gamma_V(\llbracket E \rrbracket(M^\#))$. By definition of γ , that implies the result of the analysis of the assignment is sound.
- Case where C is an input statement **input**(x): This case is similar to that of a standard assignment; indeed, the only difference is that, in the concrete, x may get assigned any value, whereas in the abstract, it gets mapped to \top_V . We observe that \top_V describes any possible value, so that the argument provided for regular assignment commands applies here in the same way.
- Case where C is the condition statement **if**(B)**{** C_0 **}****else****{** C_1 **}**: We assume that the property holds for C_0 and C_1 and prove it for C :

$$\begin{aligned} \llbracket C \rrbracket_{\mathcal{P}}(\gamma(M^\#)) &= \llbracket C_0 \rrbracket_{\mathcal{P}}(\mathcal{F}_B(\gamma(M^\#))) \cup \llbracket C_1 \rrbracket_{\mathcal{P}}(\mathcal{F}_{\neg B}(\gamma(M^\#))) \\ &\subseteq \llbracket C_0 \rrbracket_{\mathcal{P}}(\gamma(\mathcal{F}_B^\#(M^\#))) \cup \llbracket C_1 \rrbracket_{\mathcal{P}}(\gamma(\mathcal{F}_{\neg B}^\#(M^\#))) \\ &\quad \text{by soundness of } \mathcal{F}_B^\# \text{ and } \mathcal{F}_{\neg B}^\# \text{ and by the monotonicity of } \llbracket \cdot \rrbracket_{\mathcal{P}} \text{ and } \cup \\ &\subseteq \gamma(\llbracket C_0 \rrbracket_{\mathcal{P}}^\#(\mathcal{F}_B^\#(M^\#))) \cup \gamma(\llbracket C_1 \rrbracket_{\mathcal{P}}^\#(\mathcal{F}_{\neg B}^\#(M^\#))) \\ &\quad \text{by soundness of } \llbracket C_0 \rrbracket_{\mathcal{P}}^\# \text{ and } \llbracket C_1 \rrbracket_{\mathcal{P}}^\# \text{ (induction hypothesis)} \\ &\subseteq \gamma(\llbracket C_0 \rrbracket_{\mathcal{P}}^\#(\mathcal{F}_B^\#(M^\#)) \sqcup^\# \llbracket C_1 \rrbracket_{\mathcal{P}}^\#(\mathcal{F}_{\neg B}^\#(M^\#))) \\ &\quad \text{by soundness of } \sqcup^\# \\ &= \gamma(\llbracket C \rrbracket^\#(M^\#)) \end{aligned}$$

- Case where C is the loop statement **while**(B)**{** C_0 **}**: We assume that the property holds for C_0 and prove that it also holds for C . To do this, we prove the following inclusion (we remark that it entails the soundness of the analysis of C because we already know that $\mathcal{F}_{\neg B}^\#$ is sound):

$$\bigcup_{i \geq 0} ([C_0]_{\mathcal{P}} \circ \mathcal{F}_B)^i(M) \subseteq \text{abs_iter}([C_0]_{\mathcal{P}}^\# \circ \mathcal{F}_B^\#, M^\#)$$

To prove this inclusion, we need to apply theorem 3.5 (proof in appendix B.2.4) with the following definitions for F and $F^\#$:

$$F = \llbracket C_0 \rrbracket_{\mathcal{P}} \circ \mathcal{F}_B \quad F^\# = \llbracket C_0 \rrbracket_{\mathcal{P}}^\# \circ \mathcal{F}_B^\#$$

First, we establish the assumptions of the theorem:

- First, $\llbracket C_0 \rrbracket_{\mathcal{P}} \circ \mathcal{F}_B \circ \gamma \subseteq \gamma \circ \llbracket C_0 \rrbracket_{\mathcal{P}}^\# \circ \mathcal{F}_B^\#$; indeed, F_B is sound, and so is $\llbracket C_0 \rrbracket_{\mathcal{P}}$ (by induction hypothesis).
 - Second, the function $\llbracket C_0 \rrbracket_P \circ \mathcal{F}_B$ is continuous; indeed, \mathcal{F}_B is continuous (trivial set property), and $\llbracket C_0 \rrbracket_P$ is continuous (this needs to be proved by induction over the syntax of commands; the proof is trivial but a bit long, so we omit it here).
- Thus, theorem 3.5 applies. This concludes the proof.

Comments While the proof of soundness is fairly lengthy, it actually consists of a sequence of quite easy steps and merely encapsulates the previous theorems.

Moreover, the proof of this theorem would remain unchanged if we switched to a relational abstract domain. In that case, only the proofs of correctness of the basic operations would differ. Similarly, if we augment the language with additional constructions, most of this proof could be reused, with additional cases added for the additional language constructions.

B.3 Proofs of Soundness for Chapter 4

We prove theorems 4.2, 4.3, and 4.4. The proofs rely on some properties of Galois connection, which are listed and proven in appendix B.1.

B.3.1 Transitional-Style Static Analysis over Finite-Height Domains

We prove theorem 4.2, which we recall here.

Theorem B.7 (Correct static analysis by $F^\#$) *Given a program, let F and $F^\#$ be defined as in section 4.2.3. If $S^\#$ is of finite height (every chain in $S^\#$ is finite), and if $F^\#$ is monotone or extensive, then*

$$\bigsqcup_{i \geq 0} F^{\#i}(\perp)$$

is finitely computable and over-approximates $\text{lfp } F$:

$$\text{lfp } F \subseteq \gamma(\bigsqcup_{i \geq 0} F^{\#i}(\perp)) \quad \text{or, equivalently,} \quad \alpha(\text{lfp } F) \subseteq \bigsqcup_{i \geq 0} F^{\#i}(\perp).$$

Proof Proof structure is as follows:

$$\begin{array}{lcl} \text{The conditions} & \implies & F \circ \gamma \subseteq \gamma \circ F^\# \\ & \implies & \forall n \geq 0 : F^n \perp \subseteq \gamma(F^{\#n}(\perp)) \end{array} \tag{B.1}$$

$$\implies \text{lfp } F \subseteq \gamma(\bigsqcup_{i \geq 0} F^{\#i}(\perp)). \tag{B.2}$$

$$\implies \text{lfp } F \subseteq \gamma(\bigsqcup_{i \geq 0} F^{\#i}(\perp)). \tag{B.3}$$

First, note that $\bigsqcup_i F^{\#i}(\perp)$ exists in CPO $S^\#$, because that $F^\#$ is monotone or extensive implies the sequence $\perp, F^{\#1}\perp, F^{\#2}\perp, \dots$ is a chain.

(Proof of B.1) From the condition $\check{\rho}(\rightarrow) \circ \gamma \subseteq \gamma \circ \check{\rho}(\rightarrow^\#)$ follows $Step \circ \gamma \subseteq \gamma \circ Step^\#$. Recall that $Step = \check{\rho}(\rightarrow)$ and that $Step^\# = (\text{id}, \cup_M^\#) \circ \pi \circ \check{\rho}(\rightarrow^\#)$. Note that $\check{\rho}(\rightarrow^\#) \subseteq ((\text{id}, \cup_M^\#) \circ \pi) \circ \check{\rho}(\rightarrow^\#)$, because the partial order \sqsubseteq is label-wise, and the $(\text{id}, \cup_M^\#) \circ \pi$ operation partitions the result set $\subseteq \mathbb{L} \times \mathbb{M}^\#$ of $\check{\rho}(\rightarrow^\#)$ by the labels and returns a safe upper bound $(\cup_M^\#)$ of abstract memories in each partition. Hence,

$$\begin{aligned} Step \circ \gamma &= \check{\rho}(\rightarrow) \circ \gamma \subseteq \gamma \circ \check{\rho}(\rightarrow^\#) && \text{by the condition of } \rightarrow^\# \\ &\subseteq \gamma \circ (\text{id}, \cup_M^\#) \circ \pi \circ \check{\rho}(\rightarrow^\#) && \text{by the monotonicity of } \gamma \\ &= \gamma \circ Step^\#. \end{aligned}$$

Then it follows that $F \circ \gamma \subseteq \gamma \circ F^\#$, because

$$\begin{aligned} (\gamma \circ F^\#)X &= \gamma(\alpha I \cup^\# Step^\#(X)) \\ &\supseteq (\gamma \circ \alpha)I \cup (\gamma \circ Step^\#)X && \text{by the condition of } \cup^\# \\ &\supseteq I \cup (Step \circ \gamma)X && \text{by } Step \circ \gamma \subseteq \gamma \circ Step^\# \text{ and } \text{id} \subseteq \gamma \circ \alpha \\ &= (F \circ \gamma)X. \end{aligned}$$

(Proof of B.2) From $F \circ \gamma \subseteq \gamma \circ F^\#$ it follows that

$$\forall n \geq 0 : F^n \perp \subseteq \gamma(F^{\#n} \perp)$$

by induction on n . Basis case $n = 0$ holds since $F^0(\perp) = \perp$. For inductive cases, assume that $F^k(\perp) \subseteq \gamma(F^{\#k}(\perp))$. Then since F is monotone (since F is continuous),

$$\begin{aligned} F(F^k \perp) &\subseteq F(\gamma(F^{\#k} \perp)) && \text{by induction hypothesis} \\ &\subseteq \gamma(F^{\#}(F^{\#k} \perp)) && \text{by } F \circ \gamma \subseteq \gamma \circ F^\#. \end{aligned}$$

(Proof of B.3) From $\forall n \geq 0 : F^n \perp \subseteq \gamma(F^{\#n} \perp)$, we prove the final goal:

$$\bigsqcup_{i \geq 0} F^i \perp \subseteq \gamma(\bigsqcup_{i \geq 0} F^{\#i} \perp)$$

Note that $\bigsqcup_{i \geq 0} F^i(\perp)$ exists because sequence $(F^i(\perp))_{i \geq 0}$ is a chain in CPO $P(S)$, and $\bigsqcup_{i \geq 0} \gamma(F^{\#i} \perp)$ also exists in CPO $S^\#$ because that $(F^{\#i} \perp)_{i \geq 0}$ is a chain and γ is monotone imply that $(\gamma(F^{\#i} \perp))_{i \geq 0}$ is a chain. Thus, from $\forall n \geq 0 : F^n \perp \subseteq \gamma(F^{\#n} \perp)$ it follows that

$$\begin{aligned} \bigsqcup_{i \geq 0} F^i \perp &\subseteq \bigsqcup_{i \geq 0} \gamma(F^{\#i} \perp) \\ &\subseteq \gamma(\bigsqcup_{i \geq 0} F^{\#i} \perp) && \text{by the monotonicity of } \gamma. \end{aligned}$$

This concludes the proof.

B.3.2 Transitional-Style Static Analysis with Widening

We prove theorem 4.3, which we recall here.

Theorem B.8 (Correct static analysis by $F^\#$ and widening operator ∇) Given a program, let F and $F^\#$ be defined as in section 4.2.3. Let ∇ be a widening operator as defined in definition 3.11. Then the following chain $Y_0 \sqsubseteq Y_1 \sqsubseteq \dots$

$$Y_0 = \perp \quad Y_{i+1} = Y_i \nabla F^\#(Y_i)$$

is finite, and its last element Y_{\lim} over-approximates $\mathbf{lfp}F$:

$$\mathbf{lfp}F \subseteq \gamma(Y_{\lim}) \quad \text{or equivalently} \quad \alpha(\mathbf{lfp}F) \sqsubseteq Y_{\lim}.$$

Proof First, the sequence $(Y_i)_{i \geq 0}$ is a chain. The widening operator's first condition $\gamma(a \nabla b) \supseteq \gamma(a) \cup \gamma(b)$ means that

$$\begin{aligned} a \nabla b &\sqsupseteq \alpha(\gamma(a) \cup \gamma(b)) \quad \text{by Galois connection} \\ &\sqsupseteq \alpha(\gamma(a)) \quad \text{by the monotonicity of } \alpha \\ &\sqsupseteq a \quad \text{by id} \sqsubseteq \alpha \circ \gamma; \end{aligned}$$

hence, $a \nabla b \sqsupseteq a$. Thus, $\forall k \geq 0 : Y_k \sqsubseteq Y_{k+1} = Y_k \nabla F^\#(Y_k)$.

Second, by the second condition of the widening operator ∇ , the chain $(Y_i)_{i \geq 0}$ is finitely stationary. Let the last element be Y_{\lim} .

We prove by induction that

$$\forall n \geq 0 : \bigcup_{i=0}^n F^i(\perp) \subseteq \gamma(Y_n), \tag{B.4}$$

which implies, because Y_{\lim} is the biggest element of $(Y_i)_i$, that

$$\forall n \geq 0 : \bigcup_{i=0}^n F^i(\perp) \subseteq \gamma(Y_{\lim});$$

hence,

$$\bigcup_{i \geq 0} F^i(\perp) \subseteq \gamma(Y_{\lim}).$$

(Proof of B.4) Base case $n = 0$ is obvious. For inductive cases, assume that $\bigcup_{i=0}^k F^i(\perp) \subseteq \gamma(Y_k)$.

$$\begin{aligned} \bigcup_{i=0}^{k+1} F^i(\perp) &= \bigcup_{i=0}^k F^i(\perp) \cup F^{k+1}(\perp) \\ &\subseteq \gamma(Y_k) \cup F^{k+1}(\perp) \quad \text{by induction hypothesis} \\ &= \gamma(Y_k) \cup F(F^k(\perp)) \\ &\subseteq \gamma(Y_k) \cup F(\bigcup_{i=0}^k F^i(\perp)) \quad \text{by the monotonicity of } F \\ &\subseteq \gamma(Y_k) \cup F(\gamma(Y_k)) \quad \text{by induction hypothesis and the monotonicity of } F \\ &\subseteq \gamma(Y_k) \cup \gamma(F^\#(Y_k)) \quad \text{by } F \circ \gamma \subseteq \gamma \circ F^\# \\ &\subseteq \gamma(Y_k \nabla F^\#(Y_k)) \quad \text{by the condition of } \nabla \\ &= \gamma(Y_{k+1}). \quad \text{by the definition of } Y_{k+1} \end{aligned}$$

This concludes the proof.

B.3.3 Use Example of the Transitional-Style Static Analysis

We now prove theorem 4.4, which we recall here.

Theorem B.9 (Soundness of $\hookrightarrow^\#$) Consider the one-step transition relation in section 4.4.2 and its abstract version in section 4.4.4. If the semantic operators satisfy the soundness properties

$$\begin{aligned}\wp(eval_E) \circ \gamma_M &\subseteq \gamma \circ eval_E^\# \\ \wp(update_x) \circ \times \circ (\gamma_M, \gamma_W) &\subseteq \gamma_M \circ update_x^\# \\ \wp(filter_B) \circ \gamma_M &\subseteq \gamma_M \circ filter_B^\# \\ \wp(filter_{\neg B}) \circ \gamma_M &\subseteq \gamma_M \circ filter_{\neg B}^\#, \end{aligned}$$

then $\check{\wp}(\hookrightarrow) \circ \gamma \subseteq \gamma \circ \check{\wp}(\hookrightarrow^\#)$. (The \times is the Cartesian product operator of two sets.)

Proof Without loss of generality, let us consider a singleton set $\{(l, M^\#)\} \in S^\#$ and prove for the set that the conclusion holds. We proceed by case analysis for the command at the l label.

Case of “if $B C_1 C_2$ ”:

$$\begin{aligned}& (\check{\wp}(\hookrightarrow) \circ \gamma) \{(l, M^\#)\} \\ = & \check{\wp}(\hookrightarrow) \{(l, m) \mid m \in \gamma_M(M^\#)\} \\ = & \{(\text{nextTrue}(l), m_1) \mid m_1 \in \wp(filter_B)(\gamma_M(M^\#))\} \cup \{(\text{nextFalse}(l), m_2) \mid m_2 \in \wp(filter_{\neg B})(\gamma_M(M^\#))\} \\ \subseteq & \{(\text{nextTrue}(l), m_1) \mid m_1 \in \gamma_M(filter_B^\#(M^\#))\} \cup \{(\text{nextFalse}(l), m_2) \mid m_2 \in \gamma_M(filter_{\neg B}^\#(M^\#))\} \\ & \text{by the conditions of } filter_B^\# \text{ and } filter_{\neg B}^\# \\ = & \gamma \{(\text{nextTrue}(l), filter_B^\#(M^\#)), (\text{nextFalse}(l), filter_{\neg B}^\#(M^\#))\} \\ = & (\gamma \circ \check{\wp}(\hookrightarrow^\#)) \{(l, M^\#)\} \end{aligned}$$

Case of “ $x := E$ ”:

$$\begin{aligned}& (\check{\wp}(\hookrightarrow) \circ \gamma) \{(l, M^\#)\} \\ = & \check{\wp}(\hookrightarrow) \{(l, m) \mid m \in \gamma_M(M^\#)\} \\ = & \{(\text{next}(l), (update_x \circ (\text{id}, eval_E))m) \mid m \in \gamma_M(M^\#)\} \quad \text{where notation } (f, g)x = (f(x), g(x)) \\ \subseteq & \{(\text{next}(l), m) \mid m \in (\wp(update_x) \circ \times \circ (\text{id}, \wp(eval_E)) \circ \gamma_M)M^\#\} \\ = & \{(\text{next}(l), m) \mid m \in (\wp(update_x) \circ \times \circ (\gamma_M, \wp(eval_E) \circ \gamma_M))M^\#\} \\ \subseteq & \{(\text{next}(l), m) \mid m \in (\wp(update_x) \circ \times \circ (\gamma_M, \gamma_W \circ eval_E^\#))M^\#\} \quad \text{by the condition of } eval_E^\# \\ = & \{(\text{next}(l), m) \mid m \in (\wp(update_x) \circ \times \circ (\gamma_M, \gamma_W) \circ (\text{id}, eval_E^\#))M^\#\} \\ \subseteq & \{(\text{next}(l), m) \mid m \in (\gamma_M \circ update_x^\# \circ (\text{id}, eval_E^\#))M^\#\} \quad \text{by the condition of } update_x^\# \\ = & \gamma \{(\text{next}(l), (update_x^\# \circ (\text{id}, eval_E^\#))M^\#)\} \\ = & \gamma \{(\text{next}(l), update_x^\#(M^\#, eval_E^\#(M^\#)))\} \\ = & (\gamma \circ \check{\wp}(\hookrightarrow^\#)) \{(l, M^\#)\}. \end{aligned}$$

Other cases similarly hold. This concludes the proof.

Comments The underlying structure of the proof is fairly simple. Note that the \hookrightarrow and $\hookrightarrow^\#$ functions are compositions of semantic operators, both of which are homomorphic to each other. The only difference is the operators involved: the \hookrightarrow uses concrete operators, and the $\hookrightarrow^\#$ uses their abstract correspondents. The above proof of each case is basically a replay in a different guise of the proof that if every pair of concrete and its abstract operators satisfies the soundness property, then a homomorphic pair of their compositions preserves the soundness property.

The soundness preservation over composition is a common property that most of the abstract interpretations enjoy. For example, this property occurs too in the compositional-style abstract interpretation (theorem 3.1).

Bibliography

- [1] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [3] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.
- [4] ANSI ISO/IEC. *International Standard – Programming Languages – C (ANSI ISO/IEC 9899:2011)*. American National Standards Institute, 2011.
- [5] ANSI/IEEE. *IEEE Standard 745-1985: Standard for Floating-Point Arithmetic*. American National Standards Institute/IEEE, 1985.
- [6] Mounir Assaf, David A. Naumann, Julien Signoles, Eric Totel, and Frédéric Tronel. Hypercollecting semantics and its application to static analysis of information flow. In *Symposium on Principles of Programming Languages (POPL)*, pages 874–887. ACM Press, 2017.
- [7] Gilles Barthe, Pedro R. D’Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Computer Security Foundations Workshop (CSF)*, pages 100–114. IEEE, 2004.
- [8] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL: A tool suite for automatic verification of real-time systems. In *Hybrid Systems*, pages 232–243. Springer, 1995.
- [9] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter O’Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *International Conference on Computer Aided Verification (CAV)*, pages 178–192. Springer, 2007.
- [10] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.
- [11] David L. Bird and Carlos Urias Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.
- [12] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 196–207. ACM Press, 2003.

- [13] François Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal in Functional Programming (JFP)*, 2(4):407–423, 1992.
- [14] Cristiano Calcagno, Dino Distefano, Jérémie Dubreil, Dominik Gabi, Pieter Hooimeijer, Martina Luca, Peter W. O’Hearn, Irene Papakonstantinou, Jim Purbrik, and Dulma Rodriguez. Moving fast with software verification. In *Proceedings of NASA Formal Method – 7th International Symposium*, pages 3–11. Springer, 2015.
- [15] Cristiano Calcagno, Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6):26:1–26:66, 2011.
- [16] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 13–27. ACM Press, 1989.
- [17] Ghila Castelnovo, Mayur Naik, Noam Rinetzky, Mooly Sagiv, and Hongseok Yang. Modularity in lattices: A case study on the correspondence between top-down and bottom-up analysis. In *Static Analysis Symposium (SAS)*, pages 252–274. Springer, 2015.
- [18] Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. Automatically generating features for learning program analysis heuristics for C-like languages. *Proceedings of the ACM on Programming Languages (PACMPL)*, 1(OOPSLA):101:1–101:25, 2017.
- [19] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Symposium on Principles of Programming Languages (POPL)*, pages 247–260. ACM Press, 2008.
- [20] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.
- [21] Edmund C. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
- [22] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *Computer Security Foundations Workshop (CSF)*, pages 51–65. IEEE Computer Society, 2008.
- [23] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that programs eventually do something good. In *Symposium on Principles of Programming Languages (POPL)*, pages 265–276. ACM Press, 2007.
- [24] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
- [25] Patrick Cousot. *Principles of Abstract Interpretation*. MIT Press, forthcoming.
- [26] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM Press, 1977.
- [27] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages (POPL)*, pages 269–282. ACM Press, 1979.
- [28] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2-3):103–179, 1992.
- [29] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Symposium on Principles of*

Programming Languages (POPL), pages 105–118. ACM Press, 2011.

- [30] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Symposium on Principles of Programming Languages (POPL)*, pages 84–96. ACM Press, 1978.
- [31] Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In *Security in Pervasive Computing (SPC)*, pages 193–209. Springer, 2005.
- [32] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.
- [33] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k -limiting. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 230–241. ACM Press, 1994.
- [34] Alain Deutsch. Static verification of dynamic properties. Technical report, Polyspace Technologies, 2003.
- [35] Dino Distefano, Peter O’Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 287–302. Springer, 2006.
- [36] Nurit Dor, Michael Rodeh, and Shmuel Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 155–167. ACM Press, 2003.
- [37] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.
- [38] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 169–181. Springer, 1980.
- [39] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Formal Verification of Object-Oriented Software (FoVeOOS 2010)*, pages 10–30. Springer, 2010.
- [40] Christian Ferdinand, Reinhold Heckmann, and Reinhard Wilhelm. Analyzing the worst-case execution time by abstract interpretation of executable code. In *Automotive Software Workshop (ASWSD)*, pages 1–14. Springer, 2004.
- [41] Jérôme Feret. Abstract interpretation of mobile systems. *Journal of Logic and Algebraic Programming (JLAP)*, 63(1):59–130, 2005.
- [42] Jeanne Ferrante and Karl J. Ottenstein. A program form based on data dependency in predicate regions. In *Symposium on Principles of Programming Languages (POPL)*, pages 217–236. ACM Press, 1983.
- [43] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *International Conference on Computer Aided Verification (CAV)*, pages 173–177. Springer, 2007.

- [44] Robert Floyd. Assigning meaning to programs. *Proceedings of the Symposium in Applied Mathematics*, 19:19–32, 1967.
- [45] Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Symposium on Principles of Programming Languages (POPL)*, pages 186–197. ACM Press, 2004.
- [46] Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. Complete abstract interpretations made constructive. In *International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 366–377. Springer, 1998.
- [47] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM Press, 2005.
- [48] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In *Symposium on Principles of Programming Languages (POPL)*, pages 338–350. ACM Press, 2005.
- [49] Michael J. Gordon and Tom F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
- [50] Michael J. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. A metalanguage for interactive proof in LCF. In *Symposium on Principles of Programming Languages (POPL)*, pages 119–130. ACM Press, 1978.
- [51] Philippe Granger. Improving the results of static analyses of programs by local decreasing iterations. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 68–79. Springer, 1992.
- [52] Philippe Granger. Static analyses of congruence properties on rational numbers. In *Static Analysis Symposium (SAS)*, pages 278–292. Springer, 1997.
- [53] Arie Gurfinkel and Sagar Chaki. Combining predicate and numeric abstraction for software model checking. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–9. IEEE, 2008.
- [54] Arie Gurfinkel and Sagar Chaki. Boxes: A symbolic abstract domain of boxes. In *Static Analysis Symposium (SAS)*, pages 287–303. Springer, 2010.
- [55] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 339–348. ACM Press, 2008.
- [56] Maria Handjieva and Stanislav Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Static Analysis Symposium (SAS)*, pages 200–214. Springer, 1998.
- [57] Williams Ludwell Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*, 2(3-4):179–396, 1989.
- [58] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 24–34. ACM Press, 2001.
- [59] Manuel V. Hermenegildo, Germán Puebla, Kim Marriott, and Peter J. Stuckey. Incremental analysis of logic programs. In *International Conference on Logic Programming (ICLP)*,

- pages 797–811. MIT Press, 1995.
- [60] Charles Hymans. Checking safety properties of behavioral VHDL descriptions by abstract interpretation. In *Static Analysis Symposium (SAS)*, pages 444–460. Springer, 2002.
 - [61] Bertrand Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in System Design (FMSD)*, 23(1):5–37, 2003.
 - [62] Bertrand Jeannet, Alexey Loginov, Thomas W. Reps, and Shmuel Sagiv. A relational approach to interprocedural shape analysis. In *Static Analysis Symposium (SAS)*, pages 246–264. Springer, 2004.
 - [63] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *International Conference on Computer Aided Verification (CAV)*, pages 661–667. Springer, 2009.
 - [64] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *Transactions on Programming Languages and System*, 5(4):596–619, 1983.
 - [65] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *Symposium on Principles of Programming Languages (POPL)*, pages 247–259. ACM Press, 2015.
 - [66] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware C analyzer by a Bayesian statistical post analysis. In *Static Analysis Symposium (SAS)*, pages 203–217. Springer, 2005.
 - [67] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.
 - [68] Daniel Kästner and Christian Ferdinand. Proving the absence of stack overflows. In *International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, pages 202–213, 2014.
 - [69] Gerwin Klein, Philip Derrin, and Kevin Elphinstone. Experience report: seL4: Formally verifying a high-performance microkernel. In *International Conference on Functional Programming (ICFP)*, pages 91–96. ACM Press, 2009.
 - [70] Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Design Automation Conference (DAC)*, pages 368–371. ACM Press, 2003.
 - [71] David J. Kuck, Robert H. Kuhn, David A. Padua, Bruce Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *Symposium on Principles of Programming Languages (POPL)*, pages 207–218. ACM Press, 1981.
 - [72] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.
 - [73] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 235–248. ACM Press, 1992.
 - [74] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):707–723,

1998.

- [75] Woosuk Lee, Wonchan Lee, Dongok Kang, Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Sound non-statistical clustering of static analysis alarms. *Transactions on Programming Languages and Systems (TOPLAS)*, 39(4):16:1–16:35, 2017.
- [76] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 348–370. Springer, 2010.
- [77] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Symposium on Principles of Programming Languages (POPL)*, pages 42–54. ACM Press, 2006.
- [78] Huisong Li, Francois Berenger, Bor-Yuh Evan Chang, and Xavier Rival. Semantic-directed clumping of disjunctive abstract states. In *Symposium on Principles of Programming Languages (POPL)*, pages 32–45. ACM Press, 2017.
- [79] Junghee Lim and Thomas W. Reps. A system for generating static analyzers for machine instructions. In *International Conference on Compiler Construction (CC)*, pages 36–52. Springer, 2008.
- [80] Jan Midtgaard, Flemming Nielson, and Hanne Riis Nielson. A parametric abstract domain for lattice-valued regular expressions. In *Static Analysis Symposium (SAS)*, pages 338–360. Springer, 2016.
- [81] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.
- [82] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation (HOSC)*, 19(1):31–100, 2006.
- [83] Antoine Miné. Static analysis of run-time errors in embedded critical parallel C programs. In *European Symposium on Programming (ESOP)*, pages 398–418. Springer, 2011.
- [84] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.
- [85] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for C-like languages. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 229–238. ACM Press, 2012.
- [86] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 475–484. ACM Press, 2014.
- [87] Susan S. Owicky and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.
- [88] Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *Conference on Automated Deduction (CADE)*, pages 748–752. Springer, 1992.
- [89] Rohit Parikh. On context-free languages. *Journal of the ACM*, 13(4):570–581, 1966.

- [90] François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *Symposium on Principles of Programming Languages (POPL)*, pages 276–290. ACM Press, 1999.
- [91] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [92] Corneliu Popescu and Wei-Ngan Chin. Inferring disjunctive postconditions. In *Advances in Computer Science – ASIAN Computing Science Conference*, pages 331–345. Springer, 2006.
- [93] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351. Springer, 1982.
- [94] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symposium on Principles of Programming Languages (POPL)*, pages 49–61. ACM Press, 1995.
- [95] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logics in Computer Science (LICS)*, pages 55–74. IEEE, 2002.
- [96] Xavier Rival. Understanding the origin of alarms in Astrée. In *Static Analysis Symposium (SAS)*, pages 303–319. Springer, 2005.
- [97] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):26, 2007.
- [98] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
- [99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.
- [100] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.
- [101] Olin Shivers. Control-flow analysis in scheme. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 164–174. ACM Press, 1988.
- [102] Axel Simon and Andy King. Analyzing string buffers in C. In *International Conference on Algebraic Methodology and Software Technology (AMAST)*, pages 365–379. Springer, 2002.
- [103] Fausto Spoto. The Julia static analyzer for Java. In *Static Analysis Symposium (SAS)*, pages 39–57. Springer, 2016.
- [104] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages (POPL)*, pages 32–41. ACM Press, 1996.
- [105] Sarah Thompson and Alan Mycroft. Abstract interpretation of combinational asynchronous circuits. *Science of Computer Programming (SCP)*, 64(1):166–183, 2007.
- [106] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937.
- [107] Caterina Urban and Antoine Miné. A decision tree abstract domain for proving conditional termination. In *Static Analysis Symposium (SAS)*, pages 302–318. Springer, 2014.
- [108] Arnaud Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *Static Analysis Symposium (SAS)*, pages 366–382. Springer, 1996.

- [109] Mark Weiser. Program slicing. In *International Conference on Software Engineering (ICSE)*, pages 439–449, 1981.
- [110] Yichen Xie and Alexander Aiken. Scalable error detection using Boolean satisfiability. In *Symposium on Principles of Programming Languages (POPL)*, pages 351–363. ACM Press, 2005.

Index

abstract domain, 66, 102, 163, 180, 200
abstract garbage collection, 139
abstract interpretation, 32, 35, 44
abstract interpretation, compositional style, 32, 75, 90
abstract interpretation, transitional style, 44
abstract interpreter, 90
abstract interpreter, compositional style, 187
abstract interpreter, transitional style, 191
abstract iteration, 39, 41, 53, 55, 83, 109, 131, 164
abstract post-condition, 33, 78, 107
abstract semantic function, 105
abstract transition, 201
abstraction, 24, 30, 66, 67
abstraction function, 24, 28, 68
activation record, 232
address (memory location), 220
alarm, 91, 109, 166
alias, 198
aliasing abstraction, 222
allocation site abstraction, 226
analysis coverage, 164
analysis parameters, 163
analysis setup, 159
analyzer implementation bug, 157
array, 212
array abstraction, 215
array error, 213, 214
array errors, 6
array partitioning, 216

assertion, 6, 10
assisted proving, 13, 17
automation, 9
average execution time, 256

backward analysis, 147
backward semantics, 147
best abstraction, 28, 68, 74
beta reduction, 265
bottom, 103
bottom-up analysis, 161
buffer, 217
buffer overrun, 6
bug finding, 16, 17

cache hierarchy, 237
call stack, 232
call string, 233
cardinal power abstract domain, 127
case splits, 126
chain, 103
coalescent product abstract domain, 124
collecting semantics, 20
complete, 10, 11
complete partial order, 103
completeness, 10, 158
compositional semantics, 60, 176
computability, 7
concrete domain, 66, 101
concrete semantic domain, 101
concrete semantic function, 101
concretization function, 25, 67
concurrency, 237
conditional statements, 81
congruence abstraction, 71
conjunctive properties, 122

context sensitivity, 233
context-insensitive abstraction, 234
context-insensitive analysis, 210
context-sensitive abstraction, 233
context-sensitive analysis, 210
continuation, 204
continuous function, 100
control flow, 46, 97, 113
control state, 63
convergence, 86
convex polyhedra abstraction, 29, 73
CPO, 103
critical software, 4

data-flow analysis, 258
deadlock, 6, 237, 239
def-use chain, 140
denotational semantics, 62
dependence property, 251
disjunctive completion, 127
disjunctive properties, 126
dispatch model, 170
domain-specific analysis, 5
dynamic memory allocation, 196, 223

environment, 204
execution, 20
execution order, 156

fairness, 239, 247
fixpoint, 65
floating-point arithmetics, 155
flow sensitivity, 129
flow-sensitive, 102
frame rule, 139
function, 231

Galois connection, 68

halting problem, 7, 8

heterogeneous memory state, 120

higher-order control flow analysis, 266

hyperproperties, 254

incremental analysis, 143

information flow, 6, 251

information flow property, 6

interleaved execution, 237

interpreter, 62, 174

interprocedural analysis, 231

interval abstraction, 27, 70

invariant, 9, 83, 246

lattice Hasse diagram, 70

least fixpoint, 100

linear equalities abstraction, 73

live-lock, 6

liveness, 6

liveness property, 6, 247

local iterations, 152

loop unrolling, 42, 131

loops, 38, 82, 164

memory safety, 221

memory state, 63

model checking, 14, 17

modular analysis, 143

modularity, 236

monotonic closure, 262

non-domain-specific analysis, 5

non-relational abstraction, 29, 71, 121

octagon abstraction, 74
over-approximation, 15

parallelism, 237
parameter instances, 204
parameterization, 144
partial completeness, 158
partial program analysis, 159
pointer, 196, 219
pointer abstraction, 222
pointer analysis, 263
pointer error, 221
polymorphic type system, 273
principal types, 273
procedure, 231
procedure call, 231
procedure summary, 144
procedure summary abstraction, 236
product abstract domain, 122
program analysis, 2
program counter, 46
program execution, 20
program point, 46
program semantics, 20
proof system, 268

race, 237, 239
ranking function, 250
reachability property, 21, 61
reachable states, 98
recursive calls, 203
reduced cardinal power abstract domain, 128
reduced product abstract domain, 123
reduction, 123, 128
relational abstraction, 72, 121
relational property, 233

reliability, 4
Rice theorem, 8
run-time error, 6, 213, 214, 218

safety property, 6, 244
scalability, 9, 145
security, 4
security property, 251
semantic property, 2, 154
semantics, 2, 20, 60, 96, 154
separation logic, 228
shape abstraction, 226, 227
shape analysis, 227
sign abstraction, 25, 69
simple types, 269
slicing, 251
software errors, 4
sound, 10, 11
sound abstract transition, 107, 117
sound static analysis, 108, 109
soundness, 10, 90, 156
soundness under assumption, 156
sparse analysis, 137
spatial sparsity, 137, 139
state, 19
state partitioning, 128
state property, 244
statement label, 46
static analysis, 15, 17, 56
static single assignment, 265
static typing, 268
string buffer, 217
strong update, 201, 216
summarization, 215

taint analysis, 254

temporal sparsity, 137, 140
termination, 6, 247
testing, 12, 17
thread modularity, 241
threads, 237
three-valued logic, 228
time-out, 164
top-down analysis, 161
total correctness, 244
trace abstraction, 129, 247
trace partitioning, 129
trace property, 243
trace semantics, 61
transfer function, 34
transition relation, 97
transition sequence, 48, 98
transitional semantics, 45, 96, 179
triage of alarms, 91, 109, 167
type inference, 268
types, 6, 10

undecidability, 7
under-approximation, 16

value abstraction, 69
variant, 248
verification, 154

weak update, 201, 216, 223
whole-program analysis, 159
widening, 41, 55, 87, 133
widening operator, 109
worklist, 111
worst-case execution time, 256