Introduction to Static Analysis
An Abstract Interpretation Perspective

Xavier Rival and Kwangkeun Yi

The MIT Press
Cambridge, Massachusetts
London, England

This book was set in Syntax and Times Roman by the authors.

Printed and bound in the United States of America.

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 9780262043410 (hardcover : alk. paper)
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.

10 9 8 7 6 5 4 3 2 1

To our parents

Contents

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

MITPress Times.cls LaTeX Times A Priori Book Style Typeset with PDFLaTeX Size: 7x9 August 26, 2019 2:20am

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

	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

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.

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 **`setjump`/`longjump`**), 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 exe-*

cution 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 $\mathbb{L}$. Second, given a program $\mathtt{p}$ in $\mathbb{L}$, we say that an execution e terminates if it reaches the end of $\mathtt{p}$ 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,*

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

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 $\mathbb{L}$ be a Turing-complete language, and let $\mathcal{P}$ be a nontrivial semantic property of programs of $\mathbb{L}$. There exists no algorithm such that,*

for every program $\mathtt{p} \in \mathbb{L}$*, it returns* **true** *if and only if* `p` *satisfies the semantic property* $\mathcal{P}$.

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 $\mathscr{P}$ and an analysis tool `analysis`, to determine whether this property holds.

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

$$\text{for every program } \mathtt{p} \in \mathbb{L},\ \mathtt{analysis(p)} = \mathbf{true} \iff \mathtt{p} \text{ satisfies } \mathscr{P}.$$

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

$$\begin{cases} \text{For every program } \mathtt{p} \in \mathbb{L}, & \mathtt{analysis(p)} = \mathbf{true} \implies \mathtt{p} \text{ satisfies } \mathscr{P}. \\ \text{For every program } \mathtt{p} \in \mathbb{L}, & \mathtt{analysis(p)} = \mathbf{true} \impliedby \mathtt{p} \text{ satisfies } \mathscr{P}. \end{cases}$$

Therefore, we can weaken the equivalence by simply dropping either of these two implications. In both cases, we get a partially accurate too 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* $\mathscr{P}$ *whenever, for any program* $\mathtt{p} \in \mathbb{L}$*,* $\mathtt{analysis}(\mathtt{p}) = \mathbf{true}$ *implies that* `p` *satisfies property* $\mathscr{P}$*.*

When a sound analysis (or analyzer) claims that the program has property $\mathscr{P}$, 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 $\mathscr{P}$ unless this property can be guaranteed. In other words, a sound analysis will reject all programs that do not satisfy $\mathscr{P}$.

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* $\mathscr{P}$ *whenever, for every program* $\mathtt{p} \in \mathbb{L}$*, such that* `p` *satisfies* $\mathscr{P}$*,* $\mathtt{analysis}(\mathtt{p}) = \mathbf{true}$.

A complete program analysis will accept every program that satisfies property $\mathscr{P}$. 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 $\mathscr{P}$.

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.*

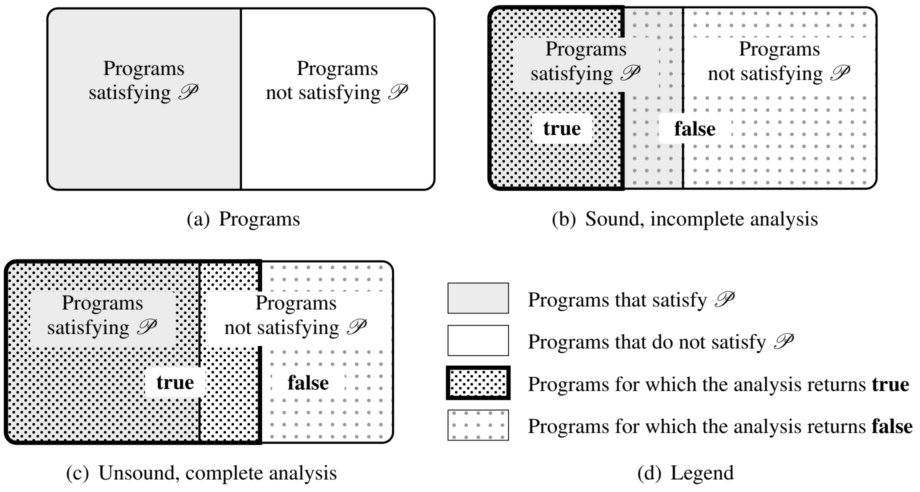
Programs satisfying $\mathscr{P}$
Programs not satisfying $\mathscr{P}$
(a) Programs
Programs satisfying $\mathscr{P}$
Programs not satisfying $\mathscr{P}$
true
false
(b) Sound, incomplete analysis
Programs satisfying $\mathscr{P}$
Programs not satisfying $\mathscr{P}$
true
false
(c) Unsound, complete analysis
Programs that satisfy $\mathscr{P}$
Programs that do not satisfy $\mathscr{P}$
Programs for which the analysis returns **true**
Programs for which the analysis returns **false**
(d) Legend

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 $\mathscr{P}$ and the programs that do not satisfy $\mathscr{P}$. 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 $\mathscr{P}$.
- 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 $\mathscr{P}$.

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.

	Automatic	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.