Texts in Theoretical Computer Science
An EATCS Series

Daniel Kroening
Ofer Strichman *Editors*

Decision Procedures

An Algorithmic Point of View

Second Edition

Springer

Texts in Theoretical Computer Science. An EATCS Series

Series editors

More information about this series at http://www.springer.com/series/3214

Daniel Kroening · Ofer Strichman

Decision Procedures

An Algorithmic Point of View

Second Edition

Springer

Daniel Kroening
Computing Laboratory
University of Oxford
Oxford
UK

Ofer Strichman
Information Systems Engineering
The William Davidson Faculty of Industrial Engineering and Management
Technion – Israel Institute of Technology
Haifa
Israel

ISSN 1862-4499
Texts in Theoretical Computer Science. An EATCS Series
ISBN 978-3-662-50496-3 ISBN 978-3-662-50497-0 (eBook)
DOI 10.1007/978-3-662-50497-0

Library of Congress Control Number: 2016957285

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer-Verlag GmbH Germany
The registered company address is: Heidelberger Platz 3, 14197 Berlin, Germany

Foreword to the first edition

By Randal E. Bryant

Research in decision procedures started several decades ago, but both their practical importance and the underlying technology have progressed rapidly in the last five years. Back in the 1970s, there was a flurry of activity in this area, mostly centered at Stanford and the Stanford Research Institute (SRI), motivated by a desire to apply formal logic to problems in artificial intelligence and software verification. This work laid foundations that are still in use today. Activity dropped off through the 1980s and 1990s, accompanied by a general pessimism about automated formal methods. A conventional wisdom arose that computer systems, especially software, were far too complex to reason about formally.

One notable exception to this conventional wisdom was the success of applying Boolean methods to hardware verification, beginning in the early 1990s. Tools such as model checkers demonstrated that useful properties could be proven about industrial-scale hardware systems, and that bugs that had otherwise escaped extensive simulation could be detected. These approaches improved on their predecessors by employing more efficient logical reasoning methods, namely ordered binary decision diagrams and Boolean satisfiability solvers. The importance of considering algorithmic efficiency, and even low-level concerns such as cache performance, became widely recognized as having a major impact on the size of problems that could be handled.

Representing systems at a detailed Boolean level limited the applicability of early model checkers to control-intensive hardware systems. Trying to model data operations, as well as the data and control structures found in software, leads to far too many states, when every bit of a state is viewed as a separate Boolean signal.

One way to raise the level of abstraction for verifying a system is to view data in more abstract terms. Rather than viewing a computer word as a collection of 32 Boolean values, it can be represented as an integer. Rather than viewing a floating-point multiplier as a complex collection of Boolean functions, many verification tasks can simply view it as an "uninterpreted function" computing some repeatable function over its inputs. From this approach came a renewed interest in decision procedures, automating the process of reasoning about different mathematical forms. Some of this work revived methods dating back many years, but alternative approaches also arose that made use of Boolean methods, exploiting the greatly improved performance of Boolean satisfiability (SAT) solvers. Most recently, decision procedures have become quite sophisticated, using the general framework of search-based SAT solvers, integrated with methods for handling the individual mathematical theories.

With the combination of algorithmic improvements and the improved performance of computer systems, modern decision procedures can readily handle problems that far exceed the capacity of their forebearers from the 1970s. This progress has made it possible to apply formal reasoning to both hardware and software in ways that disprove the earlier conventional wisdom. In addition, the many forms of malicious attacks on computer systems have created a program execution environment where seemingly minor bugs can yield serious vulnerabilities, and this has greatly increased the motivation to apply formal methods to software analysis.

Until now, learning the state of the art in decision procedures required assimilating a vast amount of literature, spread across journals and conferences in a variety of different disciplines and over multiple decades. Ideas are scattered throughout these publications, but with no standard terminology or notation. In addition some approaches have been shown to be unsound, and many have proven ineffective. I am therefore pleased that Daniel Kroening and Ofer Strichman have compiled the vast amount of information on decision procedures into a single volume. Enough progress has been made in the field that the results will be of interest to those wishing to apply decision procedures. At the same time, this is a fast-moving and active research community, making this work essential reading for the many researchers in the field.

Foreword to the second edition

By Leonardo de Moura[1], Microsoft Research

Decision procedures are already making a profound impact on a number of application areas, and had become so efficient in practice in the last 15 years (mostly since the introduction of a new generation of SAT solvers, and in particular the introduction of Chaff in 2001) that numerous practical problems that were beyond our reach beforehand are now routinely solved in seconds. Yet, they draw on a combination of some of the most fundamental areas in computer science as well as discoveries from the past century of symbolic logic. They combine the problem of Boolean satisfiability with domains such as those studied in convex optimization and term-manipulating symbolic systems. They involve the decision problem, completeness and incompleteness of logical theories, and finally complexity theory.

It is an understatement to say that we use decision procedures in Microsoft on a daily basis. Applications include security testing, static code analysis, constraint solving and software verification, to cite a few. With more than five hundred machine years, security testing at Microsoft is the largest computational usage ever for decision procedures. It has been instrumental in uncovering hundreds of subtle security critical bugs that traditional testing methods have been unable to find.

This book is both an introduction to this fascinating topic and a reference for advanced developers. It dedicates a chapter to many of the useful theories (and their combination), and describes some of their applications in software engineering, including techniques that we use for static code analysis at Microsoft. The new information in this second edition is important for both the researcher and the practitioner, since it includes general quantification (an algorithm such as E-matching), updates on efficient SAT solving and related problems (such as incremental solving), effectively propositional reasoning (EPR), and other topics of great value.

[1] The main developer of Z3, an award-winning SMT solver

Preface

A *decision procedure* is an algorithm that, given a decision problem, terminates with a correct yes/no answer. In this book, we focus on decision procedures for decidable first-order theories that are useful in the context of automated software and hardware verification, theorem proving, compiler optimization, and, since we are covering propositional logic, any problem that is in the complexity class NP and is not polynomial. The range of modeling languages that we cover in this book—propositional logic, linear arithmetic, bitvectors, quantified formulas etc.—and the modeling examples that we include for each of those, will assist the reader to translate their particular problem and solve it with one of the publically available tools. The common term for describing this field is *Satisfiability Modulo Theories*, or SMT for short, and software that solves SMT formulas is called an *SMT solver*.

Since coping with the above-mentioned tasks on an industrial scale depends critically on effective decision procedures, SMT is a vibrant and prospering research subject for many researchers around the world, both in academia and in industry. Intel, AMD, ARM and IBM are some of the companies that routinely apply decision procedures in circuit verification with ever-growing capacity requirements. Microsoft is developing an SMT solver and applies it routinely in over a dozen code analysis tools. Every user of Microsoft Windows and Microsoft Office therefore indirectly enjoys the benefits of this technology owing to the increased reliability and resilience to hacker attacks of these software packages. There are hundreds of smaller, less famous companies that use SMT solvers for various software engineering tasks, and for solving various planning and optimization problems.

There are now numerous universities that teach courses dedicated to decision procedures; occasionally, the topic is also addressed in courses on algorithms or on logic for computer science. The primary goal of this book is to serve as a textbook for an advanced undergraduate- or graduate-level computer science course. It does not assume specific prior knowledge beyond what is expected from a third-year undergraduate computer science student. The

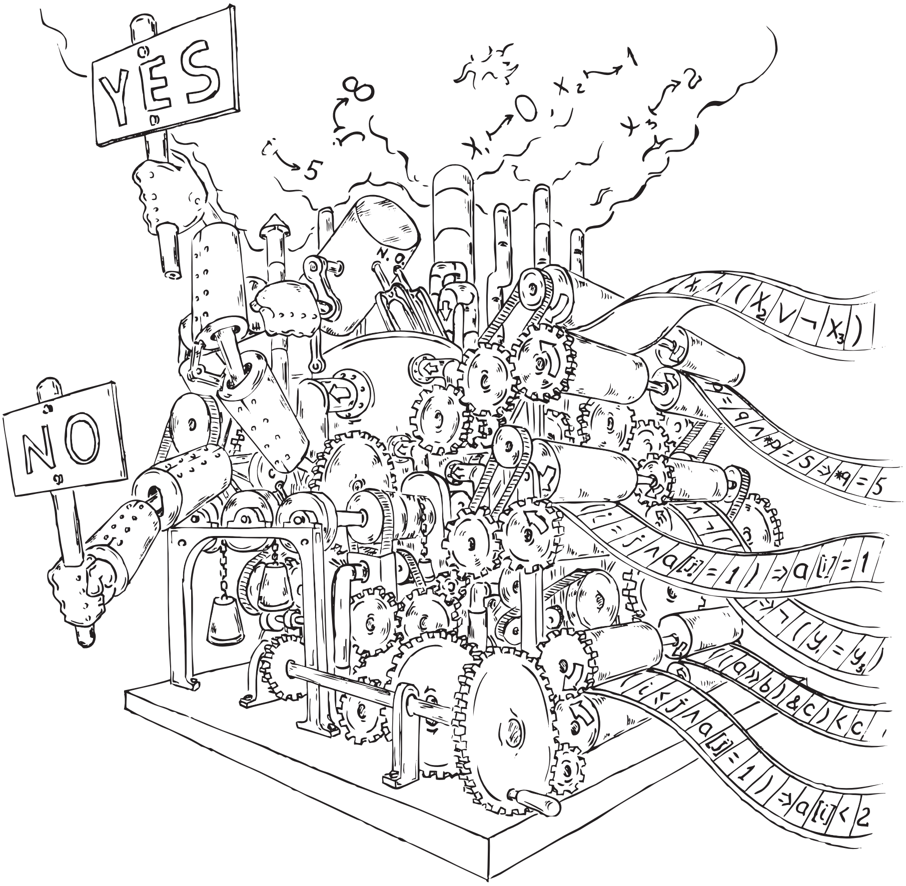

YES
NO
→5
→8
$x_1 \to 0$
$x_2 \to 1$
$x_3 \to 2$
$x_1 \wedge (x_2 \vee \neg x_3)$
$\wedge *a = 5 \to *a = 5$
$i = j \wedge a[j] = 1) \to a[i] = 1$
$\neg((y_1 = y_3)$
$((a \gg b) \& c) < c$
$(i < j \wedge a[j] = 1) \to a[i] < 2$

Fig. 1. Decision procedures can be rather complex ... those that we consider in this book take formulas of different theories as input, possibly mix them (using the Nelson–Oppen procedure—see Chap. 10), decide their satisfiability ("YES" or "NO"), and, if yes, provide a satisfying assignment

book may also help graduate students entering the field, who can save the effort to gather information from what seems to be an endless list of articles.

The decision procedures that we describe in this book draw from diverse fields such as graph theory, logic, operations research, and artificial intelligence. These procedures have to be highly efficient, since the problems they solve are inherently hard. They never seem to be efficient enough, however: what we want to be able to prove is always harder than what we *can* prove. Their asymptotic complexity and their performance in practice must always be pushed further. These characteristics are what makes this topic so compelling for research and teaching.

Which Theories? Which Algorithms?

A first-order theory can be considered "interesting", at least from a practical perspective, if it fulfills at least these two conditions:

1. The theory is expressive enough to model a real decision problem. Moreover, it is more expressive or more natural for the purpose of expressing some models in comparison with theories that are easier to decide.
2. The theory is either decidable or semidecidable, and more efficiently solvable than theories that are more expressive, at least in practice if not in theory.[2]

All the theories described in this book fulfill these two conditions. Furthermore, they are all used in practice. We illustrate applications of each theory with examples representative of real problems, whether they may be verification of C programs, verification of hardware circuits, or optimizing compilers. Background in any of these problem domains is not assumed, however.

Other than in one chapter, all the theories considered are quantifier-free. The problem of deciding them is NP-complete. In this respect, they can all be seen as alternative modeling languages that can be solved with a variety of decision procedures. They differ from each other mainly in how naturally they can be used for modeling various decision problems. For example, consider the theory of equality, which we describe in Chap. 4: this theory can express any Boolean combination of Boolean variables and expressions of the form $x_1 = x_2$, where x_1 and x_2 are variables ranging over, for example, the natural numbers. The problem of satisfying an expression in this theory can be reduced to a satisfiability problem of a propositional logic formula (and vice versa). Hence, there is no difference between propositional logic and the theory of equality in terms of their ability to model decision problems. However, many problems are more naturally modeled with the equality operator and non-Boolean variables.

For each theory that is discussed, there are many alternative decision procedures in the literature. Effort was made to select those procedures that are known to be relatively efficient in practice, and at the same time are based on what we believe to be an interesting idea. In this respect, we cannot claim to have escaped the natural bias that one has towards one's own line of research.

Every year, new decision procedures and tools are being published, and it is impossible to write a book that reports on this moving target of "the most efficient" decision procedures (the worst-case complexity of most of the competing procedures is the same). Moreover, many of them have never been thoroughly compared with one another. We refer readers who are interested in the latest developments in this field to the SMT-LIB web page, as well as to the results of the annual tool competition SMT-COMP (see Appendix A). The SMT-COMP competitions are probably the best way to stay up to date as to the relative efficiency of the various procedures and the tools that implement

[2] Terms such as *expressive* and *decidable* have precise meanings, and we will define them in the first chapter.

them. One should not forget, however, that it takes much more than a good algorithm to be efficient in practice.

The Structure and Nature of This Book

The first chapter is dedicated to basic concepts that should be familiar to third- or fourth-year computer science students, such as formal proofs, the satisfiability problem, soundness and completeness, and the trade-off between expressiveness and decidability. It also includes the theoretical basis for the rest of the book. From Sect. 1.5 onwards, the chapter is dedicated to more advanced issues that are necessary as a general introduction to the book, and are therefore recommended even for advanced readers. Chapters 2 and 3 describe how propositional formulas are checked for satisfiability, and then how this capability can be extended to more sophisticated theories. These chapters are necessary for understanding the rest of the book. Chapters 4–11 are mostly self-contained, and generally do not rely on references to material other than that in the first three chapters. The last chapter describes the application of these methods for verifying the correctness of software, and for solving various problems in computational biology.

The mathematical symbols and notations are mostly local to each chapter. Each time a new symbol is introduced, it appears in a rounded box in the margin of the page for easy reference. All chapters conclude with problems, bibliographic notes, and a glossary of symbols.

Teaching with This Book

We are aware of 38 courses worldwide that list the first edition of this book as the textbook of the course, in addition to our own courses in the Technion (Haifa, Israel) and Oxford University (UK). Our own courses are combined undergraduate and graduate courses. The slides that were used in these courses, as well as links to other resources and ideas for projects, appear on the book's web page (`www.decision-procedures.org`). Source code of a C++ library for rapid development of decision procedures can also be downloaded from this page. This library provides the necessary infrastructure for programming many of the algorithms described in this book, as explained in Appendix B. Implementing one of these algorithms was a requirement in the course, and it proved successful. It even led several students to their thesis topic.

Notes for the Second Edition

The sales of the first edition of this book crossed, apparently, the threshold above which the publisher asks the authors to write a second one... Writing this edition was a necessity for more fundamental reasons, however: at the time the first edition was written (2004–2008) the field now called SMT was in its infancy, without the standard terminology and canonic algorithms that it has now. What constituted the majority of Chap. 11 in the first edition (propositional encodings and the DPLL(T) framework) became so dominant in the years that have passed that we expanded it and brought it forward to Chap. 3. In turn, most of the so-called *eager-encoding* algorithms have been moved to Chap. 11. In addition, we updated Chap. 2 with further modern SAT heuristics, added a section about incremental satisfiability, and added a section on the related constraint satisfaction problem (CSP). To the quantifiers chapter (Chap. 9) we added a section about general quantification using *E-matching* and a section about the Bernays–Schönfinkel–Ramsey fragment of first-order logic (also called EPR). Finally, we added a new chapter (Chap. 12) on the application of SMT for software engineering in industry, partially based on writings of Nikolaj Bjørner and Leonardo de Moura from Microsoft Research, and for solving problems in computational biology based on writings of Hillel Kugler, also from Microsoft Research.

Acknowledgments

Many people read drafts of this manuscript and gave us useful advice. We would like to thank, in alphabetical order, those who helped in the first edition: Domagoj Babic, Josh Berdine, Hana Chockler, Leonardo de Moura, Benny Godlin, Alberto Griggio, Alan Hu, Wolfgang Kunz, Shuvendu Lahiri, Albert Oliveras Llunell, Joel Ouaknine, Hendrik Post, Sharon Shoham, Aaron Stump, Cesare Tinelli, Ashish Tiwari, Rachel Tzoref, Helmut Veith, Georg Weissenbacher, and Calogero Zarba, and those who helped with the second edition: Francesco Alberti, Alberto Griggio, Marijn Heule, Zurab Khasidashvili, Daniel Le Berre, Silvio Ranise, Philipp Ruemmer, Natarajan Shankar, and Cesare Tinelli. We thank Ilya Yodovsky Jr. for the drawing in Fig. 1.

Sep. 2016

Daniel Kroening
University of Oxford, United Kingdom

Ofer Strichman
Technion, Haifa, Israel

Contents

1

Introduction and Basic Concepts

While the focus of this book is on algorithms rather than mathematical logic, the two points of view are inevitably mixed: one cannot truly understand why a given algorithm is correct without understanding the logic behind it. This does not mean, however, that logic is a prerequisite, or that without understanding the fundamentals of logic, it is hard to learn and use these algorithms. It is similar, perhaps, to a motorcyclist who has the choice of whether to learn how his or her bike works.

He or she can ride a long way without such knowledge, but at certain points, when things go wrong or if the bike has to be tuned for a particular ride, understanding how and why things work comes in handy. And then again, suppose our motorcyclist does decide to learn mechanics: where should he or she stop? Is the physics of combustion engines important? Is the "why" important at all, or just the "how"? Or an even more fundamental question: should one first learn how to ride a motorcycle and then refer to the basics when necessary, or learn things "bottom-up", from principles to mechanics—from science to engineering—and then to the rules of driving?

The reality is that different people have different needs, tendencies, and backgrounds, and there is no right way to write a motorcyclist's manual that fits all. And things can get messier when one is trying to write a book about decision procedures which is targeted, on the one hand, at practitioners—programmers who need to know about algorithms that solve their particular problems—and, on the other hand, at students and researchers who need to see how these algorithms can be defined in the theoretical framework that they are accustomed to, namely logic.

This first chapter has been written with both types of reader in mind. It is a combination of a reference for later chapters and a general introduction. Section 1.1 describes the two most common approaches to formal reasoning, namely deduction and enumeration, and demonstrates them with propositional logic. Section 1.2 serves as a reference for basic terminology such as *validity*, *satisfiability*, *soundness*, and *completeness*. More basic terminology is given in Sect. 1.3, which is dedicated to *normal forms* and some of their

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_1

properties. Up to that point in the chapter, there is no new material. As of Sect. 1.4, the chapter is dedicated to more advanced issues that are necessary as a general introduction to the book. Section 1.4 positions the subject which this book is dedicated to in the theoretical framework in which it is typically discussed in the literature. This is important mainly for the second type of reader: those who are interested in entering this field as researchers, and, more generally, those who are trained to some extent in mathematical logic. This section also includes a description of the types of problems that we are concerned with in this book, and the standard form in which they are presented in the following chapters. Section 1.5 describes the trade-off between expressiveness and decidability. In Sect. 1.6, we conclude the chapter by discussing the need for reasoning about formulas with a Boolean structure.

What about the rest of the book? Each chapter is dedicated to a different first-order **theory**. We have not yet explained what a theory is, and specifically what a **first-order theory** is—that is the role of Sect. 1.4—but some examples are still in order, as some intuition as to what theories are is required before we reach that section in order to understand the direction in which we are proceeding.

Informally, one may think of a theory as a finite or an infinite set of formulas, which are characterized by common grammatical rules, the functions and predicates that are allowed, and a domain of values. The fact that they are called "first-order" means only that there is a restriction on the quantifiers (only variables, rather than sets of variables, can be quantified), but this is mostly irrelevant to us, because, in all chapters but one, we restrict the discussion to quantifier-free formulas. The table below lists some of the first-order theories that are covered in this book. The list includes the theories of bit-vectors, arrays, and pointer logic, which are necessary components in the field of software **formal verification**. This is a subfield of software engineering dedicated to proving the correctness of computer programs with respect to a given formal specification. Indeed, in Chap. 12, we describe several software verification techniques that are based on the ability to solve such formulas.

Theory name	**Example formula**	**Chapter**
Propositional logic	$x_1 \wedge (x_2 \vee \neg x_3)$	2
Equality	$y_1 = y_2 \wedge \neg(y_1 = y_3) \implies \neg(y_2 = y_3)$	4,11
Difference logic	$(z_1 - z_2 < 5) \vee (z_2 - z_3 \leq 6)$	5
Linear arithmetic	$(2z_1 + 3z_2 \leq 5) \vee (z_2 + 5z_2 - 10z_3 \geq 6)$	5
Bit vectors	$((a >> b)\ \&\ c) < c$	6
Arrays	$(i = j \wedge a[j] = 1) \implies a[i] = 1$	7
Pointer logic	$p = q \wedge *p = 5 \implies *q = 5$	8
Quantified Boolean formulas	$\forall x_1.\ \exists x_2.\ \forall x_3.\ x_1 \implies (x_2 \vee x_3)$	9
Combined theories	$(i \leq j \wedge a[j] = 1) \implies a[i] < 2$	10

In the next few sections, we use propositional logic, which we assume the reader is familiar with, in order to demonstrate various concepts that apply equally to other first-order theories.

1.1 Two Approaches to Formal Reasoning

The primary problem that we are concerned with is that of the validity (or satisfiability) of a given formula. Two fundamental strategies for solving this problem are the following:

- The **proof-theoretic** approach is to use a deductive mechanism of reasoning, based on **axioms** and **inference rules**, which together are called an **inference system**.
- The **model-theoretic** approach is to enumerate possible solutions from a finite number of candidates.

These two directions—deduction and enumeration—are apparent as early as the first lessons on propositional logic. We dedicate this section to demonstrating them.

Consider the following three claims that together are inconsistent:

1. If x is a prime number greater than 2, then x is odd.
2. It is not the case that x is not a prime number greater than 2.
3. x is not odd.

Denote the statement "*x is a prime number greater than 2*" by A and the statement "*x is odd*" by B. These claims translate into the following propositional formulas:

$$\begin{aligned} & A \implies B \\ & \neg\neg A \\ & \neg B \,. \end{aligned} \tag{1.1}$$

We would now like to prove that this set of formulas is indeed inconsistent.

1.1.1 Proof by Deduction

The first approach is to derive conclusions by using an inference system. Inference rules relate **antecedents** to their **consequents**. For example, the following are two inference rules, called modus ponens (M.P.) and Contradiction:

$$\frac{\varphi_1 \implies \varphi_2 \qquad \varphi_1}{\varphi_2}\ (\text{M.P.})\,, \tag{1.2}$$

$$\frac{\varphi \qquad \neg\varphi}{\text{FALSE}}\ (\textsc{Contradiction})\,. \tag{1.3}$$

The rule M.P. can be read as follows: from $\varphi_1 \implies \varphi_2$ and φ_1 being TRUE, deduce that φ_2 is TRUE. The formula φ_2 is the consequent of the rule M.P. Axioms are inference rules without antecedents:

$$\frac{}{\neg\neg\varphi \iff \varphi} \; (\textsc{Double-negation-AX}) \; . \tag{1.4}$$

(Axioms are typically written without the separating line above them.) We can also write a similar inference rule:

$$\frac{\neg\neg\varphi}{\varphi} \; (\textsc{Double-negation}) \; . \tag{1.5}$$

(Double-negation-AX and Double-negation are not the same, because the latter is not symmetric.) Many times, however, axioms and inference rules are interchangeable, so there is not always a sharp distinction between them.

The inference rules and axioms above are expressed with the help of arbitrary formula symbols (such as φ_1 and φ_2 in (1.2)). In order to use them for proving a particular theorem, they need to be *instantiated*, which means that these arbitrary symbols are replaced with specific variables and formulas that are relevant to the theorem that we wish to prove. For example, the inference rules (1.2), (1.3), and (1.5) can be instantiated such that false, i.e., a contradiction, can be derived from the set of formulas in (1.1):

$$\begin{array}{lll}
(1) & A \implies B & \text{(premise)} \\
(2) & \neg\neg A & \text{(premise)} \\
(3) & A & (2;\ \textsc{Double-negation}) \\
(4) & \neg B & \text{(premise)} \\
(5) & B & (1,3;\ \text{M.P.}) \\
(6) & \textsc{false} & (4,5;\ \textsc{Contradiction}) \; .
\end{array} \tag{1.6}$$

Here, in step (3), φ in the rule Double-negation is instantiated with A. The antecedent φ_1 in the rule M.P. is instantiated with A, and φ_2 is instantiated with B.

More complicated theorems may require more complicated inference systems. This raises the question of whether everything that can be proven with a given inference system is indeed valid (in this case the system is called **sound**), and whether there exists a proof of validity using the inference system for every valid formula (in this case it is called **complete**). These questions are fundamental for every deduction system; we delay further discussion of this subject and a more precise definition of these terms to Sect. 1.2.

While deductive methods are very general, they are not always the most convenient or the most efficient way to know whether a given formula is valid.

1.1.2 Proof by Enumeration

The second approach is relevant if the problem of checking whether a formula is satisfiable can be reduced to a problem of *searching* for a satisfying assignment within a finite set of options. This is the case, for example, if the variables range over a finite domain,[1] such as in propositional logic. In the case of propositional logic, enumerating solutions can be done using **truth tables**, as demonstrated by the following example:

A	B	$(A \implies B)$	$\neg\neg A$	$\neg B$	$(A \implies B) \wedge \neg\neg A \wedge \neg B$
1	1	1	1	0	0
1	0	0	1	1	0
0	1	1	0	0	0
0	0	1	0	1	0

The third, fourth, and fifth columns list the truth value of the subformulas of (1.1), whereas the sixth column lists the truth value of their conjunction. As can be seen, this formula is not satisfied by any of the four possible assignments, and is hence unsatisfiable.

1.1.3 Deduction and Enumeration

The two basic approaches demonstrated above, deduction and enumeration, go a long way, and in fact are major subjects in the study of logic. In practice, many decision procedures are not based on explicit use of either enumeration or deduction. Yet, typically their actions can be understood as performing one or the other (or both) implicitly, which is particularly helpful when arguing for their correctness.

1.2 Basic Definitions

We begin with several basic definitions that are used throughout the book. Some of the definitions that follow do not fully coincide with those that are common in the study of mathematical logic. The reason for these gaps is that we focus on quantifier-free formulas, which enables us to simplify various definitions. We discuss these issues further in Sect. 1.4.

Definition 1.1 (assignment). *Given a formula φ, an* assignment *of φ from a domain D is a function mapping φ's variables to elements of D. An assignment to φ is* full *if all of φ's variables are assigned, and* partial *otherwise.*

Definition 1.2 (satisfiability, validity, and contradiction). *A formula is* satisfiable *if there exists an assignment of its variables under which the formula evaluates to* TRUE*. A formula is a* contradiction *if it is not satisfiable. A formula is* valid *(also called a* tautology*) if it evaluates to* TRUE *under all assignments.*

[1] A finite domain is a sufficient but not a necessary condition. In many cases, even if the domain is infinite, it is possible to find a bound such that, if there exists a satisfying assignment, then there exists one within this bound. Theories that have this property are said to have the **small-model property**.

What does it mean that a formula "evaluates to TRUE" under an assignment? To evaluate a formula, one needs a definition of the **semantics** of the various functions and predicates in the formula. In propositional logic, for example, the semantics of the propositional connectives is given by truth tables. Indeed, given an assignment of all variables in a propositional formula, a truth table can be used for checking whether it satisfies a given formula, or, in other words, whether the given formula evaluates to TRUE under this assignment. The term **Satisfiability Modulo Theories** (SMT) is used to describe the satisfiability problem for an arbitrary theory or combinations thereof. Software that solves this problem is called an **SMT solver**. This book is mostly about algorithms that are used in such solvers.

It is not hard to see that a formula φ is valid if and only if $\neg\varphi$ is a contradiction. Although somewhat trivial, this is a very useful observation, because it means that we can check whether a formula is valid by checking instead whether its negation is a contradiction, i.e., not satisfiable.

Example 1.3. The propositional formula

$$A \wedge B \tag{1.7}$$

is satisfiable because there exists an assignment, namely $\{A \mapsto \text{TRUE},\ B \mapsto \text{TRUE}\}$, which makes the formula evaluate to TRUE. The formula

$$(A \implies B) \wedge A \wedge \neg B \tag{1.8}$$

is a contradiction, as we saw earlier: no assignment satisfies it. On the other hand, the negation of this formula, i.e.,

$$\neg((A \implies B) \wedge A \wedge \neg B)\ , \tag{1.9}$$

is valid: every assignment satisfies it. ▟

$\boxed{\alpha \models \varphi}$

$\boxed{\models \varphi}$

Given a formula φ and an assignment α of its variables, we write $\alpha \models \varphi$ to denote that α satisfies φ. If a formula φ is valid (and hence, all assignments satisfy it), we write $\models \varphi$.[2]

Definition 1.4 (the decision problem for formulas). *The* decision problem *for a given formula φ is to determine whether φ is valid.*

$\boxed{T}$

Given a theory T, we are interested in a procedure[3] that terminates with

[2] Recall that the discussion here refers to propositional logic. In the more general case, we are not talking about assignments, rather about structures that may or may not satisfy a formula. In that case, the notation $\models \varphi$ means that all structures satisfy φ. These terms are explained later in Sect. 1.4.

a correct answer to the decision problem, for every formula of the theory T.[4]

This can be formalized with a generalization of the notions of "soundness" and "completeness" that we saw earlier in the context of inference systems. These terms can be defined for the more general case of procedures as follows:

Definition 1.5 (soundness of a procedure). *A procedure for a decision problem is* sound *if, when it returns "Valid", the input formula is valid.*

Definition 1.6 (completeness of a procedure). *A procedure for a decision problem is* complete *if*

- *it always terminates, and*
- *it returns "Valid" when the input formula is valid.*

Definition 1.7 (decision procedure). *A procedure is called a* decision procedure *for T if it is sound and complete with respect to every formula of T.*

Definition 1.8 (decidability of a theory). *A theory is* decidable *if and only if there is a decision procedure for it.*

Given these definitions, we are able to classify procedures according to whether they are sound and complete or only sound. It is rarely the case that unsound procedures are of interest. Ideally, we would always like to have a decision procedure, as defined above. However, sometimes either this is not possible (if the problem is undecidable) or the problem is easier to solve with an incomplete procedure. Some incomplete procedures are categorized as such because they do not *always* terminate (or they terminate with a "don't know" answer). However, in many practical cases, they do terminate. Thus, completeness can also be thought of as a quantitative property rather than a binary one.

All the theories that we consider in this book, with a short exception in Sect. 9.5, are decidable. Once a theory is decidable, the next question is how difficult it is to decide it. Most of the decision problems that we consider in this book are NP-complete. The worst-case complexity of the various algorithms that we present for solving them is frequently the same, but this is not the only important measure. It is rare that one procedure dominates another. The common practice is to consider a decision procedure relevant if it is able to perform faster than others on some significant subset of public benchmarks,

[3] We follow the convention by which a **procedure** does not necessarily terminate, whereas an **algorithm** terminates. This may cause confusion, because a "decision procedure" is by definition terminating, and thus should actually be called a "decision algorithm". This confusion is rooted in the literature, and we follow it here.

[4] Every theory is defined over a set of symbols (e.g., linear arithmetic is defined over symbols such as "+" and "$\geq$"). By saying "every formula of the theory" we mean every formula that is restricted to the symbols of the theory. This will be explained in more detail in Sect. 1.4.

or on some well-defined subclass of problems. When there is no way to predict the relative performance of procedures without actually running them, they can be run in parallel, with a “first-to-end kills all others” policy. This is a common practice in industry.

1.3 Normal Forms and Some of Their Properties

The term **normal form**, in the context of formulas, is commonly used to indicate that a formula has certain syntactic properties. In this chapter, we introduce normal forms that refer to the Boolean structure of the formula. It is common to begin the process of deciding whether a given formula is satisfiable by transforming it to some normal form that the decision procedure is designed to work with. In order to argue that the overall procedure is correct, we need to show that the transformation preserves satisfiability. The relevant term for describing this relation is the following:

Definition 1.9 (equisatisfiability). *Two formulas are* equisatisfiable *if they are both satisfiable or they are both unsatisfiable.*

The basic blocks of a first-order formula are its predicates, also called the **atoms** of the formula. For example, Boolean variables are the atoms of propositional logic, whereas equalities of the form $x_i = x_j$ are the atoms of the theory of equality that is studied in Chap. 4.

Definition 1.10 (negation normal form (NNF)). *A formula is in* negation normal form *(NNF) if negation is allowed only over atoms, and* $\wedge, \vee, \neg$ *are the only allowed Boolean connectives.*

For example, $\neg(x_1 \vee x_2)$ is *not* an NNF formula, because the negation is applied to a subformula which is not an atom.

Every quantifier-free formula with a Boolean structure can be transformed in linear time to NNF, by rewriting $\implies$,

$$(a \implies b) \equiv (\neg a \vee b)\ , \tag{1.10}$$

and applying repeatedly what are known as **De Morgan’s rules**,

$$\begin{aligned} \neg(a \vee b) &\equiv (\neg a \wedge \neg b)\ , \\ \neg(a \wedge b) &\equiv (\neg a \vee \neg b)\ . \end{aligned} \tag{1.11}$$

In the case of the formula above, this results in $\neg x_1 \wedge \neg x_2$.

Definition 1.11 (literal). *A* literal *is either an atom or its negation. We say that a literal is* negative *if it is a negated atom, and* positive *otherwise.*

For example, in the propositional logic formula

$$(a \vee \neg b) \wedge \neg c \; , \tag{1.12}$$

the set of literals is $\{a, \neg b, \neg c\}$, where the last two are negative. In the theory of equality, where the atoms are equality predicates, a set of literals can be $\{x_1 = x_2, \neg(x_1 = x_3), \neg(x_2 = x_1)\}$.

Literals are syntactic objects. The set of literals of a given formula changes if we transform it by applying De Morgan's rules. Formula (1.12), for example, can be written as $\neg(\neg a \wedge b) \wedge \neg c$, which changes its set of literals.

Definition 1.12 (state of a literal under an assignment). *A positive literal is* satisfied *if its atom is assigned* TRUE. *Similarly, a negative literal is* satisfied *if its atom is assigned* FALSE.

Definition 1.13 (pure literal). *A literal is called* pure *in a formula φ, if all occurrences of its variable have the same sign.*

In many cases, it is necessary to refer to the set of a formula's literals as if this formula were in NNF. In such cases, either it is assumed that the input formula is in NNF (or transformed to NNF as a first step), or the set of literals in this form is computed indirectly. This can be done by simply counting the number of negations that nest each atom instance: it is negative if and only if this number is odd.

For example, $\neg x_1$ is a literal in the NNF of

$$\varphi := \;\neg(\neg x_1 \implies x_2) \; , \tag{1.13}$$

because there is an occurrence of x_1 in φ that is nested in three negations (the fact that x_1 is on the left-hand side of an implication is counted as a negation). It is common in this case to say that the **polarity** (also called the **phase**) of this occurrence is negative.

Theorem 1.14 (monotonicity of NNF). *Let φ be a formula in NNF and let α be an assignment of its variables. Let the* positive set *of α with respect to φ, denoted pos(α, φ), be the literals that are satisfied by α. For every assignment α' to φ's variables such that pos$(\alpha, \varphi) \subseteq$ pos(α', φ), $\alpha \models \varphi \implies \alpha' \models \varphi$.* *pos*

Figure 1.1 illustrates this theorem: increasing the set of literals satisfied by an assignment maintains satisfiability. It does *not* maintain unsatisfiability, however: it can turn an unsatisfying assignment into a satisfying one.

The proof of this theorem is left as an exercise (Problem 1.3).

Example 1.15. Let

$$\varphi := \;(\neg x \wedge y) \vee z \tag{1.14}$$

be an NNF formula. Consider the following assignments and their corresponding positive sets with respect to φ:

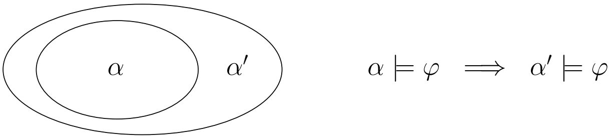


Fig. 1.1. Illustration of Theorem 1.14. The ellipses correspond to the sets of literals satisfied by α and α', respectively

$$\begin{array}{rclrcl} \alpha & := & \{x \mapsto 0,\ y \mapsto 1,\ z \mapsto 0\} & pos(\alpha, \varphi) & := & \{\neg x, y\}\ , \\ \alpha' & := & \{x \mapsto 0,\ y \mapsto 1,\ z \mapsto 1\} & pos(\alpha', \varphi) & := & \{\neg x, y, z\}\ . \end{array} \tag{1.15}$$

By Theorem 1.14, since $\alpha \models \varphi$ and $pos(\alpha, \varphi) \subseteq pos(\alpha', \varphi)$, then $\alpha' \models \varphi$. Indeed, $\alpha' \models \varphi$. ▟

We now describe two very useful restrictions of NNF: disjunctive normal form (DNF) and conjunctive normal form (CNF).

Definition 1.16 (disjunctive normal form (DNF)). *A formula is in* disjunctive normal form *if it is a disjunction of conjunctions of literals, i.e., a formula of the form*

$$\bigvee_i \big(\bigwedge_j l_{ij} \big)\ , \tag{1.16}$$

where l_{ij} is the j-th literal in the i-th **term** *(a term is a conjunction of literals).*

Example 1.17. In propositional logic, l is a Boolean literal, i.e., a Boolean variable or its negation. Thus the following formula over Boolean variables a, b, c, and d is in DNF:

$$\begin{array}{ll} (a \land c \land \neg b) & \lor \\ (\neg a \land d) & \lor \\ (b \land \neg c \land \neg d) & \lor \\ \vdots & \end{array} \tag{1.17}$$

In the theory of equality, the atoms are equality predicates. Thus, the following formula is in DNF:

$$\begin{array}{ll} ((x_1 = x_2) \land \neg(x_2 = x_3) \land \neg(x_3 = x_1)) & \lor \\ (\neg(x_1 = x_4) \land (x_4 = x_2)) & \lor \\ ((x_2 = x_3) \land \neg(x_3 = x_4) \land \neg(x_4 = x_1)) & \lor \\ \vdots & \end{array} \tag{1.18}$$

▟

Every formula with a Boolean structure can be transformed into DNF, while potentially increasing the size of the formula exponentially. The following example demonstrates this exponential ratio:

Example 1.18. The following formula is of length linear in n:

$$(x_1 \vee x_2) \wedge \cdots \wedge (x_{2n-1} \vee x_{2n}) \ . \tag{1.19}$$

The length of the equivalent DNF, however, is exponential in n, since every new binary clause (a disjunction of two literals) doubles the number of terms in the equivalent DNF, resulting, overall, in 2^n terms:

$$\begin{array}{c} (x_1 \wedge x_3 \wedge \cdots \wedge x_{2n-3} \wedge x_{2n-1}) \vee \\ (x_1 \wedge x_3 \wedge \cdots \wedge x_{2n-3} \wedge x_{2n}) \ \ \vee \\ (x_1 \wedge x_3 \wedge \cdots \wedge x_{2n-2} \wedge x_{2n}) \ \ \vee \\ \vdots \end{array} \tag{1.20}$$

▟

Although transforming a formula to DNF can be too costly in terms of computation time, it is a very natural way to decide formulas with an arbitrary Boolean structure.

Suppose we are given a disjunctive linear arithmetic formula, that is, a Boolean structure in which the atoms are linear inequalities over the reals. We know how to decide whether a *conjunction* of such literals is satisfiable: there is a known method called Simplex that can give us this answer. In order to use the Simplex method to solve the more general case in which there are also disjunctions in the formula, we can perform **syntactic case-splitting**. This means that the formula is transformed into DNF, and then each term is solved separately. Each such term contains a conjunction of literals, a form which we know how to solve. The overall formula is satisfiable, of course, if any one of the terms is satisfiable. **Semantic case-splitting**, on the other hand, refers to techniques that split the search space, in the case where the variables are finite ("first the case in which $x = 0$, then the case in which $x = 1$ ...").

The term **case-splitting** (without being prefixed with "syntactic") usually refers in the literature to either syntactic case-splitting or a "smart" implementation thereof. Indeed, many of the cases that are generated in syntactic case-splitting are redundant, i.e., they share a common subset of conjuncts that contradict each other. Efficient decision procedures should somehow avoid replicating the process of deducing this inconsistency, or, in other words, they should be able to **learn**, as demonstrated in the following example:

Example 1.19. Consider the following formula:

$$\varphi := \ (a = 1 \vee a = 2) \wedge a \geq 3 \wedge (b \geq 4 \vee b \leq 0) \ . \tag{1.21}$$

The DNF of φ consists of four terms:

$$\begin{array}{l} (a = 1 \wedge a \geq 3 \wedge b \geq 4) \ \ \vee \\ (a = 2 \wedge a \geq 3 \wedge b \geq 4) \ \ \vee \\ (a = 1 \wedge a \geq 3 \wedge b \leq 0) \ \ \vee \\ (a = 2 \wedge a \geq 3 \wedge b \leq 0) \ . \end{array} \tag{1.22}$$

These four cases can each be discharged separately, by using a decision procedure for linear arithmetic (Chap. 5). However, observe that the first and the third case share the two conjuncts $a = 1$ and $a \geq 3$, which already makes the case unsatisfiable. Similarly, the second and the fourth case share the conjuncts $a = 2$ and $a \geq 3$. Thus, with the right learning mechanism, two of the four calls to the decision procedure can be avoided. This is still case-splitting, but more efficient than a plain transformation to DNF. ▟

The problem of reasoning about formulas with a general Boolean structure is a common thread throughout this book.

Definition 1.20 (conjunctive normal form (CNF)). *A formula is in* conjunctive normal form *if it is a conjunction of disjunctions of literals, i.e., it has the form*

$$\bigwedge_i \Big(\bigvee_j l_{ij} \Big) \, , \tag{1.23}$$

where l_{ij} is the j-th literal in the i-th **clause** *(a clause is a disjunction of literals).*

Every formula with a Boolean structure can be transformed into an equivalent CNF formula, while potentially increasing the size of the formula exponentially. Yet, any propositional formula can also be transformed into an equisatisfiable CNF formula with only a *linear* increase in the size of the formula. The price to be paid is n new Boolean variables, where n is the number of **logical gates** in the formula. This transformation is done via **Tseitin's encoding** [277].

Tseitin suggested that one new variable should be added for every *logical gate* in the original formula, and several clauses to constrain the value of this variable to be equal to the gate it represents, in terms of the inputs to this gate. The original formula is satisfiable if and only if the conjunction of these clauses together with the new variable associated with the topmost operator is satisfiable. This is best illustrated with an example.

Example 1.21. Given a propositional formula

$$x_1 \implies (x_2 \wedge x_3) \, , \tag{1.24}$$

with Tseitin's encoding we assign a new variable to each subexpression, or, in other words, to each logical gate, for example, AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). For this example, let us assign the variable a_2 to the AND gate (corresponding to the subexpression $x_2 \wedge x_3$) and a_1 to the IMPLICATION gate (corresponding to $x_1 \implies a_2$), which is also the topmost operator of this formula. Figure 1.2 illustrates the **derivation tree** of our formula, together with these auxiliary variables in square brackets.

We need to satisfy a_1, together with two equivalences,

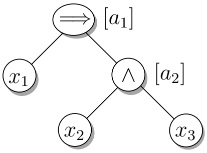


Fig. 1.2. Tseitin's encoding. Assigning an auxiliary variable to each logical gate (given here in square brackets) enables us to translate each propositional formula to CNF, while increasing the size of the formula only linearly

$$\begin{array}{lll} a_1 & \Longleftrightarrow & (x_1 \implies a_2) \ , \\ a_2 & \Longleftrightarrow & (x_2 \land x_3) \ . \end{array} \tag{1.25}$$

The first equivalence can be rewritten in CNF as

$$\begin{array}{ll} (a_1 \lor x_1) & \land \\ (a_1 \lor \neg a_2) & \land \\ (\neg a_1 \lor \neg x_1 \lor a_2) \ , & \end{array} \tag{1.26}$$

and the second equivalence can be rewritten in CNF as

$$\begin{array}{ll} (\neg a_2 \lor x_2) & \land \\ (\neg a_2 \lor x_3) & \land \\ (a_2 \lor \neg x_2 \lor \neg x_3) \ . & \end{array} \tag{1.27}$$

Thus, the overall CNF formula is the conjunction of (1.26), (1.27), and the unit clause

$$(a_1) \ , \tag{1.28}$$

which represents the topmost operator. ▟

There are various optimizations that can be performed in order to reduce the size of the resulting formula and the number of additional variables. For example, consider the following formula:

$$x_1 \lor (x_2 \land x_3 \land x_4 \land x_5) \ . \tag{1.29}$$

With Tseitin's encoding, we need to introduce four auxiliary variables. The encoding of the clause on the right-hand side, however, can be optimized to use just a single variable, say a_2:

$$a_2 \iff (x_2 \land x_3 \land x_4 \land x_5) \ . \tag{1.30}$$

In CNF,

$$\begin{array}{ll} (\neg a_2 \lor x_2) & \land \\ (\neg a_2 \lor x_3) & \land \\ (\neg a_2 \lor x_4) & \land \\ (\neg a_2 \lor x_5) & \land \\ (a_2 \lor \neg x_2 \lor \neg x_3 \lor \neg x_4 \lor \neg x_5) \ . & \end{array} \tag{1.31}$$

In general, we can encode a conjunction of n literals with a single variable and $n+1$ clauses, which is an improvement over the original encoding, requiring $n-1$ auxiliary variables and $3(n-1)$ clauses.

Such savings are also possible for a series of disjunctions (see Problem 1.1). Another popular optimization is that of **subsumption**: given two clauses such that the set of literals in one of the clauses subsumes the set of literals in the other clause, the longer clause can be discarded without affecting the satisfiability of the formula.

Finally, if the original formula is in NNF, the number of clauses can be reduced substantially, as was shown by Plaisted and Greenbaum in [225]. Tseitin's encoding is based on constraints of the form

$$\text{auxiliary variable} \iff \text{formula}\ , \tag{1.32}$$

but only the left-to-right implication is necessary. The proof that this improvement is correct is left as an exercise (Problem 1.4). In practice, experiments show that, owing to the requirement to transform the formula to NNF first, this reduction has a relatively small (positive) effect on the run time of modern SAT solvers compared with Tseitin's encoding.

Example 1.22. Consider a gate $x_1 \wedge x_2$, which we encode with a new auxiliary variable a. Three clauses are necessary to encode the constraint $a \iff (x_1 \wedge x_2)$, as was demonstrated in (1.27). The constraint $a \Longleftarrow (x_1 \wedge x_2)$ (equivalently, $(a \vee \neg x_1 \vee \neg x_2)$) is redundant, however, which means that only two out of the three constraints are necessary. ▟

A conversion algorithm with similar results to [225], in which the elimination of the negations is built in (rather than the formula being converted to NNF a priori), has been given by Wilson [285].

1.4 The Theoretical Point of View

While we take the algorithmic point of view in this book, it is important to understand also the theoretical context, especially for readers who are also interested in following the literature in this field or are more used to the terminology of formal logic. It is also necessary for understanding Chaps. 3 and 10. We must assume in this subsection that the reader is familiar to some extent with first-order logic—a reasonable exposition of this subject is beyond the scope of this book. See [43, 138] for a more organized study of these matters. Let us recall some of the terms that are directly relevant to our topic.

First-order logic (also called **predicate logic**) is based on the following elements:

1. *Variables*: a set of *variables.*

2. *Logical symbols*: the standard Boolean connectives (e.g., "$\wedge$", "$\neg$", and "$\vee$"), quantifiers ("$\exists$" and "$\forall$"), and parentheses.
3. *Nonlogical symbols*: function, predicate, and constant symbols.
4. *Syntax*: rules for constructing formulas. Formulas adhering to these rules are said to be **well formed**.

Essentially, first-order logic extends propositional logic with quantifiers and the nonlogical symbols. The syntax of first-order logic extends the syntax of propositional logic naturally. Two examples of such formulas are

- $\exists y \in \mathbb{Z}.\ \forall x \in \mathbb{Z}.\ x > y$,
- $\forall n \in \mathbb{N}.\ \exists p \in \mathbb{N}.\ n > 1 \implies (isprime(p) \wedge n < p < 2n)$,

where "$>$", "$<$", and "*isprime*" are nonlogical binary predicate symbols.

The elements listed above only refer to symbols and syntax—they still do not tell us how to evaluate whether a given formula is true or false. This separation between symbols and their interpretation—between syntax and semantics—is an important principle in the study of logic. We shall explain this separation with an example. Let Σ denote the set of symbols $\{0, 1, +, =\}$, where "0" and "1" are constant symbols, "+" is a binary function symbol, and "=" is a binary predicate symbol. Consider the following formula over Σ:

$$\varphi := \ \exists x.\ x + 0 = 1\ . \tag{1.33}$$

Now, is φ true in $\mathbb{N}_0$? ($\mathbb{N}_0$ denotes the naturals, including 0.)

What seems like a trivial question is not so simple in the world of formal logic. A logician would say that the answer depends, among other things, on the **interpretation** of the symbols in Σ. What does the "+" symbol mean? Which elements in the domain do "0" and "1" refer to? From a formal perspective, whether φ is true can only be answered with respect to a given interpretation, which has the form of a **structure**:

- A domain
- An interpretation of the nonlogical symbols, in the form of a mapping from each function and predicate symbol to a function and a predicate, respectively, and an assignment of a domain element to each of the constant symbols
- An assignment of a domain element to each of the free (unquantified) variables

For example, if we choose to interpret the "+" symbol as the *multiplication* function, the answer is that φ in (1.33) is false.

The formula φ is **satisfiable** if and only if there *exists* a structure under which the formula is true. Indeed, in this case there exists such a domain and interpretation—namely, $\mathbb{N}_0$ and the common interpretation of "+", "=", "0", and "1"—and, hence, the formula is satisfiable.

This example is confusing because it demonstrated that we have to specify an interpretation for the symbol "+", which already has a common interpretation. It is of course inadvisable to do such a thing when defining a theory:

one should not use symbols preloaded with meaning, and then require that the meaning will be formally specified. We will therefore assume that the interpretation of such symbols is *fixed* and matches their common interpretation. Interpretation will only be necessary for “anonymous” symbols.

First-order logic can be thought of as a framework giving a generic syntax and the building blocks for defining specific restrictions thereof, called **theories**. The restrictions defined by a theory are on the nonlogical symbols that can be used and the interpretation that we can give them. Indeed, in a practical setting, we would not want to consider an arbitrary interpretation of the symbols as above (where “+” is multiplication); rather we consider only specific ones.

A set of nonlogical symbols is called a **signature**. Given a signature Σ, a **Σ-formula** is a formula that uses only nonlogical symbols from Σ (possibly in addition to logical symbols). A variable is **free** if it is not bound by a quantifier. A **sentence** is a formula without free variables. A first-order **Σ-theory** T consists of a set of Σ-sentences. For a given Σ-theory T, a Σ-formula φ is **T-satisfiable** if there exists a structure that satisfies both the formula and the sentences of T. Similarly, a Σ-formula φ is **T-valid** if all structures that satisfy the sentences of T also satisfy φ.

The set of sentences that are required is sometimes large or even infinite. It is therefore common to define theories via a set of axioms, which implicitly represent all the sentences that can be inferred from them, using some sound and complete inference system for the logical symbols.

Example 1.23. Consider a simple signature Σ consisting only of the predicate symbol “=”.[5] Let T be a Σ-theory. An example of a well-formed Σ-formula is

$$\forall x.\ \forall y.\ \forall z.\ (((x = y) \land \neg(y = z)) \implies \neg(x = z))\ . \tag{1.34}$$

If we wish T to restrict the interpretation of “=” to the equality predicate, the following three axioms are sufficient:

$$\begin{array}{ll} \forall x.\ x = x & (\textsc{reflexivity})\ , \\ \forall x.\ \forall y.\ x = y \implies y = x & (\textsc{symmetry})\ , \\ \forall x.\ \forall y.\ \forall z.\ x = y \land y = z \implies x = z & (\textsc{transitivity})\ . \end{array} \tag{1.35}$$

Since every domain and interpretation that satisfy these axioms also satisfy (1.34), then (1.34) is T-valid. ▟

As said above, a theory restricts only the nonlogical symbols. If we want to restrict the set of logical symbols or the grammar, we need to speak about **fragments** of the logic. For example, we can speak about the **quantifier-free fragment** of T. This fragment, called **equality logic**, happens to be

[5] It is frequently the case in the literature that the equality sign is considered as a logical symbol, and then the theory defined here has an empty signature. We do not follow this convention here, however.

the subject of Chap. 4. Most of the chapters, in fact, are concerned with quantifier-free fragments of theories. Another useful fragment is called the **conjunctive fragment**, which means that the only Boolean connective that is allowed is conjunction. What about restricting the interpretation of the logical symbols? The axioms that restrict the interpretation of the logical symbols, called the **logical axioms**, are assumed to be "built in", i.e., they are common to all first-order theories.

Numerous theories have been considered over the years, corresponding to various problems of interest. Many of them lead to decidability and, frequently, to efficient decision procedures. The theory of **Presburger arithmetic**, for example, is defined with a signature $\Sigma = \{0, 1, +, =\}$ and is still decidable. By contrast, the theory of **Peano arithmetic**, which is defined over a signature $\Sigma = \{0, 1, +, \cdot, =\}$, is undecidable. Thus, the addition of the multiplication symbol and the corresponding axioms that define it makes the decision problem undecidable. Other famous theories include the theory of equality, the theory of real arithmetic, the theory of integer arithmetic, the theory of arrays, the theory of recursive data structures, and the theory of sets. Many of the decidable ones that are in practical use are covered in this book.

1.4.1 The Problem We Solve

Unless otherwise stated, we are concerned with

> *the satisfiability problem of the quantifier-free fragment of various first-order theories.*

Formulas in such fragments are called **ground formulas**, as they only contain free (unquantified, also called ground) variables and constants. Exceptions are Chap. 9, which is concerned with quantified formulas, and a small part of Chap. 7, which is concerned with quantified array logic.

There is a subtle difference between the satisfiability problem of ground formulas and the satisfiability problem of existentially quantified formulas. It is, of course, trivial that a ground formula φ over variables $x_1, \ldots, x_n$ is satisfiable if and only if

$$\exists x_1, \ldots, x_n .\ \varphi \tag{1.36}$$

is satisfiable. Thus, the decision procedures for both problems can be similar. The reason we use the former definition is that this entails, from a formal perspective, that the satisfying structure includes an assignment of the variables, because they are all free. In many practical applications, such an assignment is necessary. In fact, the former problem can be seen as an instance of the **constraint satisfaction problem** (CSP), which is all about finding an assignment that satisfies a set of unquantified constraints.[6]

[6] The emphasis and terminology are somewhat different. Most of the research in the CSP community is concerned with finite, discrete domains, in contrast to the problems considered in this book.

We assume that the input formulas are given in negation normal form, or that they are implicitly transformed to this form as a first step of any of the algorithms described later. As explained after Definition 1.10, every formula can be transformed to this form in linear time. The reason that this assumption is important is that it simplifies the algorithms and the arguments for their correctness.

1.4.2 Our Presentation of Theories

Our presentation of theories in the chapters to come is not as defined above. In an attempt to make the presentation more accessible and the chapters more self-contained, we make the following changes:

1. Rather than specifying theories through their set of symbols and sentences, we give the domain explicitly, and fix the interpretations of symbols in accordance with their common use. Hence, "+" is always the addition function, "0" is the 0 element in the given domain, and so forth.
2. Rather than specifying the theory fragment we are concerned with by referring to the generic grammar of first-order logic as a starting point, we give an explicit, self-contained definition of the grammar.

From a formal-logic point of view, fixing the interpretation means only that we have the sentences implicitly; the satisfiability problem remains the same. From the *algorithmic* point of view, however, the satisfiability problem now amounts to searching for a satisfying assignment of variables from the predefined domain. Whether a given assignment satisfies the formula can be determined according to the commonly used meanings of the various symbols.

This form of presentation is in line with our focus on the algorithmic point of view: when designing a decision procedure for a theory, the interpretation of the symbols has to be predefined. In other words, changing the domain or interpretation of symbols changes the algorithm.

1.5 Expressiveness vs. Decidability

There is an important trade-off between what a theory can express and how hard it is to decide, that is, how hard it is to determine whether a given formula allowed by the theory is valid or not. This is the reason for defining many different theories: otherwise, we would define and use only a single theory sufficiently expressive for all perceivable decision problems.

A theory can be seen as a tool for defining **languages**. Every formula in the theory defines a language, which is the set of "words" (the assignments, in the case of quantifier-free formulas) that satisfy it. We now define what it means that one theory is more expressive than another.

Definition 1.24 (expressiveness). *Theory A is* more expressive *than theory B if every language that can be defined by a B-formula can also be defined by an A-formula, and there exists at least one language definable by an A-formula that cannot be defined by a B-formula. We denote the fact that theory B is less expressive than theory A by* $B \prec A$*.* $\boxed{B \prec A}$

For example, propositional logic is more expressive than what is known as "**2-CNF**", i.e., CNF in which each clause has at most two literals. In propositional logic, we can define the formula

$$x_1 \vee x_2 \vee x_3 \ , \tag{1.37}$$

which defines a language that we cannot define with 2-CNF: it accepts all truth assignments to x_1, x_2, x_3 except $\{x_1 \mapsto \text{FALSE},\ x_2 \mapsto \text{FALSE},\ x_3 \mapsto \text{FALSE}\}$. How can we prove this?

Well, assume that there exists a 2-CNF representation of this formula using the same set of variables, and consider one of its binary clauses. Such a clause contradicts two of the eight possible assignments. For example, a clause $(x_1 \vee x_2)$ contradicts $\{x_1 \mapsto \text{FALSE},\ x_2 \mapsto \text{FALSE},\ x_3 \mapsto \text{FALSE}\}$ and $\{x_1 \mapsto \text{FALSE},\ x_2 \mapsto \text{FALSE},\ x_3 \mapsto \text{TRUE}\}$. Any additional clause can only contradict more assignments. Hence, we can never create a 2-CNF formula such that exactly one of the eight assignments does not satisfy it.

On the other hand, 2-CNF is a restriction of propositional logic; Hence, obviously, any 2-CNF formula can be expressed in propositional logic. Thus, we have

$$\text{2-CNF} \prec \text{propositional logic} \ . \tag{1.38}$$

This example also demonstrates the influence of expressiveness on computational hardness: while propositional logic is NP-complete, 2-CNF can be solved in polynomial time.

In order to illustrate the trade-off between how expressive a theory is and how hard it is to decide formulas in that theory, consider a theory T defined by some syntax. Let $T_1, \ldots, T_n$ denote a list of fragments of T, defined by various restrictions on the syntax of T (similarly to the way we restricted propositional logic to 2-CNF above), for which $T_1 \prec T_2 \prec \ldots \prec T_n \prec T$. Technically, this means that we have imposed a **total order** on these fragments in terms of their expressive power. Under such assumptions, Fig. 1.3 illustrates the trade-off between expressiveness and computational hardness: the less expressive the theory is (the more restrictions we put on it), the easier it is to decide it. Assume our imaginary theory T is undecidable. After some threshold is crossed (from right to left in the figure), the theory fragments can become decidable. After enough restrictions have been added, the theory becomes solvable in polynomial time. The decidable but nonpolynomially decidable fragments pose a *computational challenge*. This is one of the challenges we focus on in this book.

This view is simplistic, however, because there is no total order on the expressive power of theories, only a partial order. This means that there can be two theories, A and B, neither of which is more expressive than the other, yet their expressive power is different. In other words, there are languages that can be defined by A and not by B, and there are languages that can be defined by B and not by A.

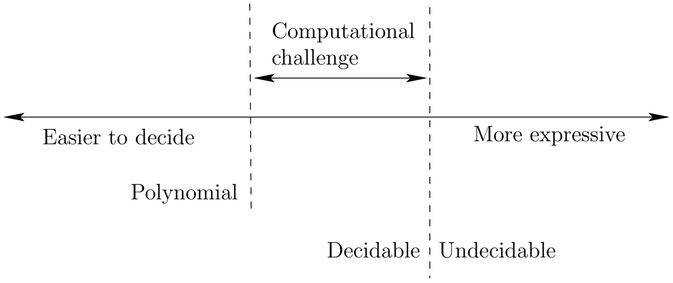


Fig. 1.3. The trade-off between expressiveness of theories and the hardness of deciding them, illustrated for an imaginary series of theories $T_1, \ldots, T_n$ for which each T_i, $i \in \{1, \ldots, n\}$ is less expressive than its successor

1.6 Boolean Structure in Decision Problems

Many decision procedures assume that the decision problem is given by a conjunction of constraints. The Simplex algorithm and the Omega test, both of which are described in Chap. 5, are examples of such procedures.

Many applications, however, require a more complex Boolean structure. In program analysis and verification, for example, disjunctions may appear in the program to be verified, either explicitly (e.g., `x = y || z`) or implicitly through constructs such as `if` and `switch` statements. Any reasoning system about such programs, therefore, must be able to deal with disjunctions. For example, **verification conditions** that arise in program verification (e.g., using **Hoare logic**) often have the form of an implication.

The following example focuses on a technique for reasoning about programs that demonstrates how program structure, including `if` statements, is evident in the underlying verification conditions that need to be checked:

Example 1.25. Bounded model checking (BMC) of programs is a technique for verifying that a given property (typically given as an assertion by the user) holds for a program in which the number of loop iterations and recursive calls is bounded by a given number k. The states that the program can reach within this bound are represented symbolically by a formula, together with the negation of the property. If the combined formula is satisfiable, then there exists a path in the program that violates the property.

Consider the program in the left part of Fig. 1.4. The number of paths through this program is exponential in N, as each of the $a[i]$ elements can be either zero or nonzero. Despite the exponential number of paths through the program, its states can be encoded with a formula of size linear in N, as demonstrated in the right part of the figure.

```
int a[N]; unsigned c;
  ...

  c = 0;
  for(i = 0; i < N; i++)
    if(a[i] == 0)
      c++;
```

$$
\begin{aligned}
c_1 &= 0 \;\wedge \\
c_2 &= (a[0] = 0)\,?\,c_1 + 1 \,:\, c_1 \;\wedge \\
c_3 &= (a[1] = 0)\,?\,c_2 + 1 \,:\, c_2 \;\wedge \\
&\ldots \\
c_{N+1} &= (a[N-1] = 0)\,?\,c_N + 1 : c_N
\end{aligned}
$$

Fig. 1.4. A simple program with an exponential number of paths (*left*), and a static single assignment (SSA) form of this program after unwinding its `for` loop (*right*)

The formula on the right of Fig. 1.4 encodes the states of the program on its left, using the **static single assignment** (SSA) form. Briefly, this means that, in each assignment of the form `x = exp;`, the left-hand side variable `x` is replaced with a new variable, say x_1, and any reference to `x` after this line and before `x` is assigned again is replaced with x_1. Such a replacement is possible because there are no loops (recall that this is done in the context of BMC). After this transformation, the statements are conjoined. The resulting equation represents the states of the original program.

The ternary operator $c\,?\,x : y$ in the equation on the right of Fig. 1.4 can be rewritten using a disjunction, as illustrated in (1.39). These disjunctions lead to an exponential number of clauses once the formula is converted to DNF.

$$
\begin{aligned}
& c_1 = 0 \wedge \\
& ((a[0] = 0 \wedge c_2 = c_1 + 1) \vee (a[0] \neq 0 \wedge c_2 = c_1)) \wedge \\
& ((a[1] = 0 \wedge c_3 = c_2 + 1) \vee (a[1] \neq 0 \wedge c_3 = c_2)) \wedge \\
& \ldots \\
& ((a[N-1] = 0 \wedge c_{N+1} = c_N + 1) \vee (a[N-1] \neq 0 \wedge c_{N+1} = c_N))\ .
\end{aligned}
\tag{1.39}
$$

In order to verify that some assertion holds at a specific location in the program, it is sufficient to add a constraint corresponding to the negation of this assertion, and check whether the resulting formula is satisfiable. For example, to prove that at the end of the program $c \leq N$, we need to conjoin (1.39) with $(c_{N+1} > N)$. ▟

To summarize this section, there is a need to reason about formulas with disjunctions, as illustrated in the example above. The simple solution of going through DNF does not scale, and better solutions are needed. Solutions that perform better in practice (the worst case remains exponential, of course) indeed exist, and are covered extensively in this book.

1.7 Logic as a Modeling Language

Before we continue with algorithms, let us briefly discuss the benefits of modeling with logic to a layperson. Indeed, whereas (mathematical) logic was originally developed, over a century ago, as a means to formalize math, today most people know it as a *modeling language* that may help them solve various problems in engineering, logistics, combinatorial optimization, etc. The simplistic example in (1.1) uses logic to model a sequence of mathematical claims, which can then be solved by off-the-shelf software, but one can also use it to model problems that arise in industry, like discovering bugs in software or checking whether a robot can complete a specific task within a given number of steps. The availability of powerful solvers for such formulas—solvers that are based on algorithms that are presented in this book—is what enables one to forget that logic reasoning is what eventually solves their problems.

What is the alternative to reducing the problem to a logical formula? The answer is: working even harder. One can always try to find an algorithm that solves the problem directly. Perhaps that solution would even have better worst-case complexity, based on some properties unique to the problem at hand (such cases typically imply that the wrong logic fragment was used to model the problem in the first place). But it is more likely that ready-made engines, which reflect research over decades and immense effort by hundreds of people, are going to solve it faster.

In that sense the boundaries between the fields of logic, **constraint solving**, and more generally **mathematical programming** is vague. The latter is a field that was developed mostly in parallel to computational logic in the **operations research** community. It also offers a modeling language and corresponding engines to solve models written in this language, but there are still some differences in emphasis:

1. *The constraints language*: Mathematical programming is traditionally focused on solving linear and nonlinear constraints over infinite domains. Constraint solving is mostly focused on solving formulas in which the variables are restricted to a finite domain. SMT, on the other hand, offers a much more expressive modeling language that covers both[7] (consider the examples given in the beginning of this chapter: arrays, pointers, bitvectors, etc.). SMT offers a very general algorithmic framework, which separates the Boolean structure from the theory, and offers a generic way to

[7] In Sect. 2.2.8 we describe constraints solving. It is also an extensible modeling language, like SMT, and as such the SMT language cannot "cover" it. But constraints solving problems can be reduced to propositional logic, which is covered by SMT.

combine theories. Correspondingly the **SMT-LIB** standard, which covers the modeling language, is occasionally extended with new theories. A brief description of the current standard is given in Appendix A. In the same appendix we describe which of the theories in the standard are covered in this book (see diagram on page 310).

2. *The technology*: The underlying technology of SMT solvers is different, even when it comes to solving linear programming problems. SMT solvers are based on SAT solvers (Chap. 2), and as such they are highly effective for solving large formulas with a rich Boolean structure.
3. *The goal*: The focus of mathematical programming is not on satisfiability, rather on the slightly harder problem of *optimality*: given a model of the constraints, it attempts to find a solution that brings to minimum the value of a given function, called the *objective function*. When it comes to problems in which the domains are finite, however, the difference between optimization and satisfiability is shallow: given a satisfiability engine one can find the optimal solution, if a solution exists at all, with a sequence of satisfiability checks. Each check involves adding a new constraint, which requires the value of the objective to be better than the previous time. Indeed, some use SMT solvers as optimization engines.

The communities behind these fields—all of which include both academics and companies—adopt each other's algorithms and concepts, to the point that the distinction between them is mostly because of historical reasons.

1.8 Problems

Problem 1.1 (improving Tseitin's encoding).

(a) Using Tseitin's encoding, transform the following formula φ to CNF. How many clauses are needed?

$$\varphi := \neg(x_1 \wedge (x_2 \vee \ldots \vee x_n)) \,. \tag{1.40}$$

(b) Consider a clause $(x_1 \vee \ldots \vee x_n)$, $n > 2$, in a non-CNF formula. How many auxiliary variables are necessary for encoding it with Tseitin's encoding? Suggest an alternative way to encode it, using a single auxiliary variable. How many clauses are needed?

Problem 1.2 (expressiveness and complexity).

(a) Let T_1 and T_2 be two theories whose satisfiability problem is decidable and in the same complexity class. Is the satisfiability problem of a T_1-formula reducible to a satisfiability problem of a T_2-formula?

(b) Let T_1 and T_2 be two theories whose satisfiability problems are reducible to one another. Are T_1 and T_2 in the same complexity class?

Problem 1.3 (monotonicity of NNF with respect to satisfiability). Prove Theorem 1.14.

Problem 1.4 (one-sided Tseitin encoding). Let φ be an NNF formula (see Definition 1.10). Let $\overrightarrow{\varphi}$ be a formula derived from φ as in Tseitin's encoding (see Sect. 1.3), but where the CNF constraints are derived from implications from left to right rather than equivalences. For example, given a formula

$$a_1 \wedge (a_2 \vee \neg a_3)\ ,$$

the new encoding is the CNF equivalent of the following formula:

$$\begin{array}{lll} x_0 & & \wedge \\ (x_0 & \Longrightarrow & a_1 \wedge x_1) \wedge \\ (x_1 & \Longrightarrow & a_2 \vee x_2) \wedge \\ (x_2 & \Longrightarrow & \neg a_3)\ , \end{array}$$

where x_0, x_1, x_2 are new auxiliary variables. Note that Tseitin's encoding to CNF starts with the same formula, except that the "$\Longrightarrow$" symbol is replaced with "$\Longleftrightarrow$".

(a) Prove that $\overrightarrow{\varphi}$ is satisfiable if and only if φ is.
(b) Let l, m, n be the number of AND, OR, and NOT gates, respectively, in φ. Derive a formula parameterized by l, m, and n that expresses the ratio of the number of CNF clauses in Tseitin's encoding to that in the one-sided encoding suggested here.

1.9 Glossary

The following symbols were used in this chapter:

Symbol	Refers to ...	First used on page ...
$\alpha \models \varphi$	An assignment α satisfies a formula φ	6
$\models \varphi$	A formula φ is valid (in the case of quantifier-free formulas, this means that it is satisfied by all assignments from the domain)	6

continued on next page

continued from previous page		
Symbol	**Refers to ...**	**First used on page ...**
T	A theory	6
$pos(\alpha, \varphi)$	Set of literals of φ satisfied by an assignment α	9
$B \prec A$	Theory B is less expressive than theory A	19

2

Decision Procedures for Propositional Logic

2.1 Propositional Logic

We assume that the reader is familiar with propositional logic, and with the complexity classes NP and NP-complete.

The syntax of formulas in propositional logic is defined by the following grammar:

$$\begin{aligned} \mathit{formula} &: \mathit{formula} \wedge \mathit{formula} \mid \neg \mathit{formula} \mid (\mathit{formula}) \mid \mathit{atom} \\ \mathit{atom} &: \mathit{Boolean\text{-}identifier} \mid \text{TRUE} \mid \text{FALSE} \end{aligned}$$

Other Boolean operators such as OR ($\vee$) and XOR ($\oplus$) can be constructed using AND ($\wedge$) and NOT ($\neg$).

2.1.1 Motivation

Since SAT, the problem of deciding the satisfiability of propositional formulas, is NP-complete, it can be used for solving any NP problem. Any other NP-complete problem (e.g., k-coloring of a graph) can be used just as well, but none of them has a natural input language such as propositional logic to model the original problem. Indeed, propositional logic is widely used in diverse areas such as database queries, planning problems in artificial intelligence, automated reasoning, and circuit design. Let us consider two examples: a layout problem and a program verification problem.

Example 2.1. Let $S = \{s_1, \ldots, s_n\}$ be a set of radio stations, each of which has to be allocated one of k transmission frequencies, for some $k < n$. Two stations that are too close to each other cannot have the same frequency. The set of pairs having this constraint is denoted by E. To model this problem, define a set of propositional variables $\{x_{ij} \mid i \in \{1, \ldots, n\}, j \in \{1, \ldots, k\}\}$. Intuitively, variable x_{ij} is set to TRUE if and only if station i is assigned the frequency j. The constraints are:

© Springer-Verlag Berlin Heidelberg 2016
D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_2

- Every station is assigned at least one frequency:

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{k} x_{ij} \; . \tag{2.1}$$

- Every station is assigned not more than one frequency:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k-1} (x_{ij} \implies \bigwedge_{j<t\leq k} \neg x_{it}) \; . \tag{2.2}$$

- Close stations are not assigned the same frequency. For each $(i, j) \in E$,

$$\bigwedge_{t=1}^{k} (x_{it} \implies \neg x_{jt}) \; . \tag{2.3}$$

Note that the input of this problem can be represented by a graph, where the stations are the graph's nodes and E corresponds to the graph's edges. Checking whether the allocation problem is solvable corresponds to solving what is known in graph theory as the *k-colorability* problem: can all nodes be assigned one of k colors such that two adjacent nodes are assigned different colors? Indeed, one way to solve k-colorability is by reducing it to propositional logic.

Example 2.2. Consider the two code fragments in Fig. 2.1. The fragment on the right-hand side might have been generated from the fragment on the left-hand side by an optimizing compiler.

```
if(!a && !b) h();
else
    if(!a) g();
    else f();
```

```
if(a) f();
else
    if(b) g();
    else h();
```

Fig. 2.1. Two code fragments—are they equivalent?

We would like to check if the two programs are equivalent. The first step in building the **verification condition** is to model the variables a and b and the procedures that are called using the Boolean variables a, b, f, g, and h, as can be seen in Fig. 2.2.

The `if-then-else` construct can be replaced by an equivalent propositional logic expression as follows:

$$(\texttt{if } x \texttt{ then } y \texttt{ else } z) \quad \equiv \quad (x \wedge y) \vee (\neg x \wedge z) \; . \tag{2.4}$$

Consequently, the problem of checking the equivalence of the two code fragments is reduced to checking the validity of the following propositional formula:

```
if $\neg a \wedge \neg b$ then $h$
else
   if $\neg a$ then $g$
   else $f$
```

```
if $a$ then $f$
else
   if $b$ then $g$
   else $h$
```

Fig. 2.2. In the process of building a formula—the verification condition—we replace the program variables and the function symbols with new Boolean variables

$$\begin{array}{rl} & (\neg a \wedge \neg b) \wedge h \ \vee \ \neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g \ \vee \ a \wedge f) \\ \Longleftrightarrow & a \wedge f \ \vee \ \neg a \wedge (b \wedge g \ \vee \ \neg b \wedge h)\ . \end{array} \tag{2.5}$$

2.2 SAT Solvers

2.2.1 The Progress of SAT Solving

Given a propositional formula $\mathcal{B}$, a SAT solver decides whether $\mathcal{B}$ is satisfiable; if it is, it also reports a satisfying assignment. In this chapter, we consider only the problem of solving formulas in conjunctive normal form (CNF) (see Definition 1.20). Since every formula can be converted to this form in linear time (as explained right after Definition 1.20), this does not impose a real restriction.[1] Solving general propositional formulas can be somewhat more efficient in some problem domains, but most of the solvers and most of the research are still focused on CNF formulas.

The practical and theoretical importance of the satisfiability problem has led to a vast amount of research in this area, which has resulted in exceptionally powerful SAT solvers. Modern SAT solvers can solve many real-life CNF formulas with hundreds of thousands or even millions of variables in a reasonable amount of time. Figures 2.3 and 2.4 illustrate the progress of these tools through the years (see captions). Of course, there are also instances of problems two orders of magnitude smaller that these tools still cannot solve. In general, it is very hard to predict which instance is going to be hard to solve, without actually attempting to solve it. Some tools, however, called **SAT portfolio** solvers, use machine-learning techniques to extract features of CNF formulas in order to select the most suitable SAT solver for the job. More details on this approach are given in Sect. 2.4.

The success of SAT solvers can be largely attributed to their ability to learn from wrong assignments, to prune large search spaces quickly, and to focus first on the "important" variables, those variables that, once given the

[1] Appendix B provides a library for performing this conversion and generating CNF in the DIMACS format, which is used by virtually all publicly available SAT solvers.

right value, simplify the problem immensely.[2] All of these factors contribute to the fast solving of both satisfiable and unsatisfiable instances. There is empirical evidence in [213] that shows that solving satisfiable instances fast requires a different set of heuristics than those that are necessary for solving unsatisfiable instances.

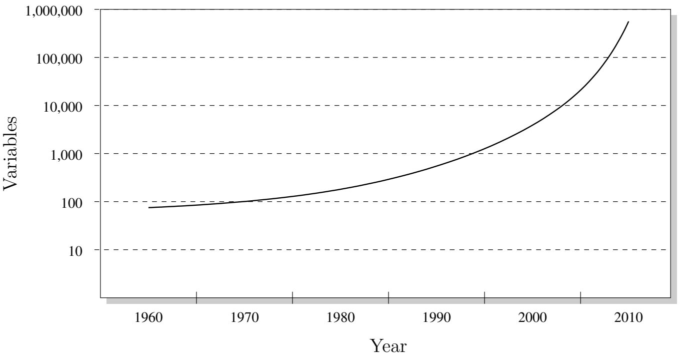


Fig. 2.3. The size of industrial CNF formulas (instances generated for solving various realistic problems such as verification of circuits and planning problems) that are regularly solved by SAT solvers in a few hours, according to year. Most of the progress in efficiency has been made in the last decade

The majority of modern SAT solvers can be classified into two main categories. The first category is based on the Conflict-Driven Clause Learning (**CDCL**) framework: in this framework the tool can be thought of as traversing and backtracking on a binary tree, in which internal nodes represent partial assignments, and the leaves represent full assignments. Building a simple CDCL solver is surprisingly easy: one can do so with fewer than 500 lines of C++ and STL.

The second category is based on a **stochastic search**: the solver guesses a full assignment, and then, if the formula is evaluated to FALSE under this assignment, starts to flip values of variables according to some (greedy) heuristic. Typically it counts the number of unsatisfied clauses and chooses the flip that minimizes this number. There are various strategies that help such solvers avoid local minima and avoid repeating previous bad moves. CDCL solvers, however, are considered better in most cases according to annual competitions that measure their performance with numerous CNF instances. CDCL solvers also have the advantage that, unlike most stochastic search methods, they are complete (see Definition 1.6). Stochastic methods seem to have an average

[2] Specifically, every formula has what is known as **backdoor variables** [284], which are variables that, once given the right value, simplify the formula to the point that it is polynomial to solve.

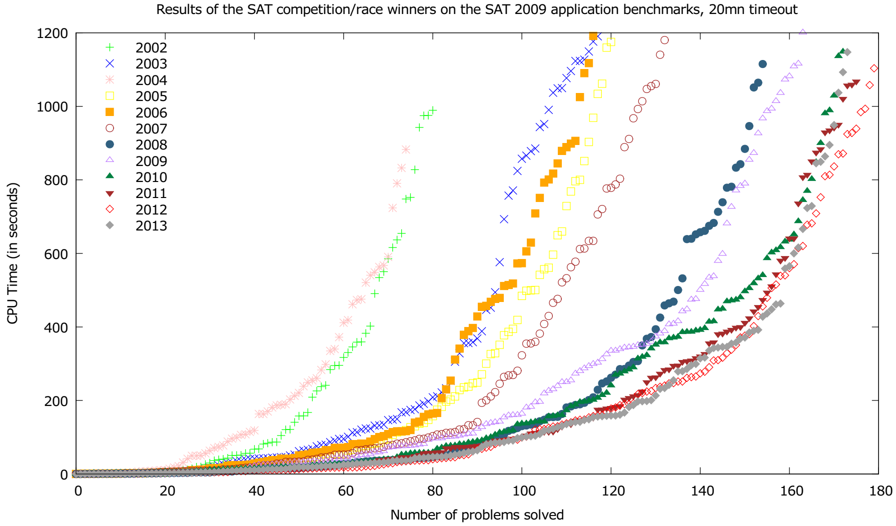


Fig. 2.4. Annual competitions measure the success of SAT solvers when applied to randomly selected benchmarks arriving from industry. The graph shows a comparison between the winners of these competitions as of 2002, when applied to a common benchmark set and using the same single-core hardware. Such graphs are nicknamed "cactus plots". A point (x, y) means that x benchmarks are solved within y amount of time each. Hence, the more the graph is to the right, the better it is. One may observe that the number of solved instances within 20 minutes has more than doubled within a decade, thanks to better algorithms. The instances in this set are large, and solvers created before 2002 run out of memory when trying to solve them. (Courtesy of Daniel Le-Berre)

advantage in solving randomly generated (satisfiable) CNF instances, which is not surprising: in these instances there is no structure to exploit and learn from, and no obvious choices of variables and values, which makes the heuristics adopted by CDCL solvers ineffective. We shall focus on CDCL solvers only.

A historical note: CDCL was developed over time as a series of improvements to the *Davis–Putnam–Loveland–Logemann* (**DPLL**) framework. See the bibliographic notes at the end of this chapter for further discussion.

2.2.2 The CDCL Framework

In its simplest form, a CDCL solver progresses by making a decision about a variable and its value, propagating implications of this decision that are easy to detect, and backtracking in the case of a conflict. Viewing the process as a search on a binary tree, each decision is associated with a **decision level**, which is the depth in the binary decision tree at which it is made, starting from 1. The assignments implied by a decision are associated with its decision level. Assignments implied regardless of the current assignments (owing to **unary clauses**, which are clauses with a single literal) are associated with decision level 0, also called the **ground level**.

Definition 2.3 (state of a clause under an assignment). *A clause is* **satisfied** *if one or more of its literals are satisfied (see Definition 1.12),* **conflicting** *if all of its literals are assigned but not satisfied,* **unit** *if it is not satisfied and all but one of its literals are assigned, and* **unresolved** *otherwise.*

Note that the definitions of a unit clause and an unresolved clause are only relevant for partial assignments (see Definition 1.1).

Example 2.4. Given the partial assignment

$$\{x_1 \mapsto 1,\ x_2 \mapsto 0,\ x_4 \mapsto 1\}\ , \tag{2.6}$$

$(x_1 \vee x_3 \vee \neg x_4)$	is satisfied,
$(\neg x_1 \vee x_2)$	is conflicting,
$(\neg x_1 \vee \neg x_4 \vee x_3)$	is unit,
$(\neg x_1 \vee x_3 \vee x_5)$	is unresolved.

■

Given a partial assignment under which a clause becomes unit, it must be extended so that it satisfies the unassigned literal of this clause. This observation is known as the **unit clause rule**. Following this requirement is necessary but obviously not sufficient for satisfying the formula.

For a given unit clause C with an unassigned literal l, we say that l is implied by C and that C is the **antecedent clause** of l, denoted by $Antecedent(l)$. If more than one unit clause implies l, the clause that the SAT solver actually used in order to imply l is the one we refer to as l's antecedent.

Example 2.5. The clause $C := (\neg x_1 \vee \neg x_4 \vee x_3)$ and the partial assignment $\{x_1 \mapsto 1,\ x_4 \mapsto 1\}$ imply the assignment x_3 and $Antecedent(x_3) = C$. ■

A framework followed by most modern CDCL solvers has been presented by, for example, Zhang and Malik [299], and is shown in Algorithm 2.2.1. The table in Fig. 2.6 includes a description of the main components used in this algorithm, and Fig. 2.5 depicts the interaction between them. A description of the ANALYZE-CONFLICT function is delayed to Sect. 2.2.6.

2.2.3 BCP and the Implication Graph

We now demonstrate Boolean constraint propagation (BCP), reaching a conflict, and backtracking. Each assignment is associated with the decision level

Algorithm 2.2.1: CDCL-SAT

Input: A propositional CNF formula $\mathcal{B}$
Output: "Satisfiable" if the formula is satisfiable and "Unsatisfiable" otherwise

```
function CDCL
    while (TRUE) do
        while (BCP() = "conflict") do
            backtrack-level := ANALYZE-CONFLICT();
            if backtrack-level < 0 then return "Unsatisfiable";
            BackTrack(backtrack-level);
        if ¬DECIDE() then return "Satisfiable";
```

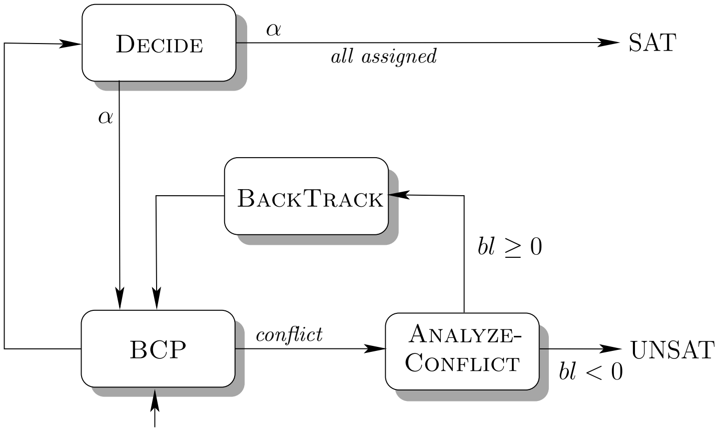


Fig. 2.5. CDCL-SAT: high-level overview of the Conflict-Driven Clause-Learning algorithm. The variable *bl* is the backtracking level, i.e., the decision level to which the procedure backtracks. α is an assignment (either partial or full)

at which it occurred. If a variable x_i is assigned 1 (TRUE) (owing to either a decision or an implication) at decision level dl, we write $x_i@dl$. Similarly, $\neg x_i@dl$ reflects an assignment of 0 (FALSE) to this variable at decision level dl. Where appropriate, we refer only to the truth assignment, omitting the decision level, in order to make the notation simpler.

$\boxed{x_i@dl}$

The process of BCP is best illustrated with an **implication graph**. An implication graph represents the current partial assignment and the reason for each of the implications.

Definition 2.6 (implication graph). *An* implication graph *is a labeled directed acyclic graph $G(V, E)$, where:*

Name	Decide()
Output	false if and only if there are no more variables to assign.
Description	Chooses an unassigned variable and a truth value for it.
Comments	There are numerous heuristics for making these decisions, some of which are described later in Sect. 2.2.5. Each such decision is associated with a decision level, which can be thought of as the depth in the search tree.
Name	BCP()
Output	“conflict” if and only if a conflict is encountered.
Description	Repeated application of the unit clause rule until either a conflict is encountered or there are no more implications.
Comments	This repeated process is called Boolean Constraint Propagation (BCP). BCP is applied even before the first decision because of the possible existence of unary clauses.
Name	Analyze-Conflict()
Output	Minus 1 if a conflict at decision level 0 is detected (which implies that the formula is unsatisfiable). Otherwise, a decision level which the solver should backtrack to.
Description	A detailed description of this function is delayed to Sect. 2.2.4. Briefly, it is responsible for computing the backtracking level, detecting global unsatisfiability, and adding new constraints on the search in the form of new clauses.
Name	BackTrack(*dl*)
Description	Sets the current decision level to *dl* and erases assignments at decision levels larger than *dl*.

Fig. 2.6. A description of the main components of Algorithm 2.2.1

- *V represents the literals of the current partial assignment (we refer to a node and the literal that it represents interchangeably). Each node is labeled with the literal that it represents and the decision level at which it entered the partial assignment.*
- *E with $E = \{(v_i, v_j) \mid v_i, v_j \in V, \neg v_i \in Antecedent(v_j)\}$ denotes the set of directed edges where each edge (v_i, v_j) is labeled with $Antecedent(v_j)$.*
- *G can also contain a single* **conflict node** *labeled with κ and incoming edges $\{(v, \kappa) \mid \neg v \in c\}$ labeled with c for some conflicting clause c.*

The root nodes of an implication graph correspond to decisions, and the internal nodes to implications through BCP. A conflict node with incoming edges labeled with c represents the fact that the BCP process has reached a conflict, by assigning 0 to all the literals in the clause c (i.e., c is conflicting).

In such a case, we say that the graph is a **conflict graph**. The implication graph corresponds to all the decision levels lower than or equal to the current one, and is dynamic: backtracking removes nodes and their incoming edges, whereas new decisions, implications, and conflict clauses increase the size of the graph.

The implication graph is sensitive to the order in which the implications are propagated in BCP, which means that the graph is not unique for a given partial assignment. In most SAT solvers, this order is rather arbitrary (in particular, BCP progresses along a list of clauses that contain a given literal, and the order of clauses in this list can be sensitive to the order of clauses in the input CNF formula). In some other SAT solvers—see for example [223]—this order is not arbitrary; rather, it is biased towards reaching a conflict faster.

A **partial implication graph** is a subgraph of an implication graph, which illustrates the BCP at a specific decision level. Partial implication graphs are sufficient for describing Analyze-Conflict. The roots in such a partial graph represent assignments (not necessarily decisions) at decision levels lower than *dl*, in addition to the decision at level *dl*, and internal nodes correspond to implications at level *dl*. The description that follows uses mainly this restricted version of the graph.

Consider, for example, a formula that contains the following set of clauses, among others:

$$\begin{aligned} c_1 &= (\neg x_1 \vee x_2)\ , \\ c_2 &= (\neg x_1 \vee x_3 \vee x_5)\ , \\ c_3 &= (\neg x_2 \vee x_4)\ , \\ c_4 &= (\neg x_3 \vee \neg x_4)\ , \\ c_5 &= (x_1 \vee x_5 \vee \neg x_2)\ , \\ c_6 &= (x_2 \vee x_3)\ , \\ c_7 &= (x_2 \vee \neg x_3)\ , \\ c_8 &= (x_6 \vee \neg x_5)\ . \end{aligned} \tag{2.7}$$

Assume that at decision level 3 the decision was $\neg x_6@3$, which implied $\neg x_5@3$ owing to clause c_8 (hence, $Antecedent(\neg x_5) = c_8$). Assume further that the solver is now at decision level 6 and assigns $x_1 \mapsto 1$. At decision levels 4 and 5, variables other than $x_1, \ldots, x_6$ were assigned, and are not listed here as they are not relevant to these clauses.

The implication graph on the left of Fig. 2.7 demonstrates the BCP process at the current decision level 6 until, in this case, a conflict is detected. The roots of this graph, namely $\neg x_5@3$ and $x_1@6$, constitute a sufficient condition for creating this conflict. Therefore, we can safely add to our formula the **conflict clause**

$$c_9 = (x_5 \vee \neg x_1)\ . \tag{2.8}$$

While c_9 is logically implied by the original formula and therefore does not change the result, it prunes the search space. The process of adding conflict clauses is generally referred to as **learning**, reflecting the fact that this is the

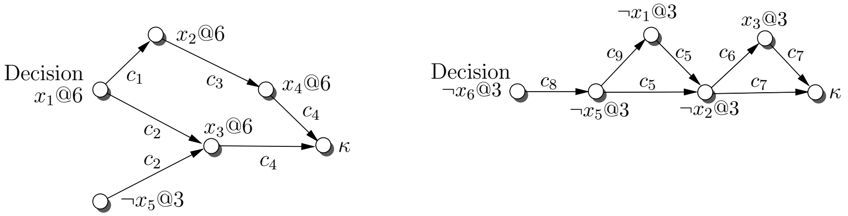


Fig. 2.7. A partial implication graph for decision level 6, corresponding to the clauses in (2.7), after a decision $x_1 \mapsto 1$ (*left*) and a similar graph after learning the conflict clause $c_9 = (x_5 \vee \neg x_1)$ and backtracking to decision level 3 (*right*)

solver's way to learn from its past mistakes. As we progress in this chapter, it will become clear that conflict clauses not only prune the search space, but also have an impact on the decision heuristic, the backtracking level, and the set of variables implied by each decision.

Analyze-Conflict is the function responsible for deriving new conflict clauses and computing the backtracking level. It traverses the implication graph backwards, starting from the conflict node κ, and generates a conflict clause through a series of steps that we describe later in Sect. 2.2.4. For now, assume that c_9 is indeed the clause generated.

After detecting the conflict and adding c_9, the solver determines which decision level to backtrack to according to the **conflict-driven backtracking** strategy. According to this strategy, the backtracking level is set to the *second most recent decision level in the conflict clause*, while erasing all decisions and implications made *after* that level. There are two special cases: when learning a unary clause, the solver backtracks to the ground level; when the conflict is *at* the ground level, the backtracking level is set to -1 and the solver exits and declares the formula to be unsatisfiable.

In the case of c_9, the solver backtracks to decision level 3 (the decision level of x_5), and erases all assignments from decision level 4 onwards, including the assignments to x_1, x_2, x_3, and x_4.

The newly added conflict clause c_9 becomes a unit clause since $x_5 = 0$, and therefore the assignment $\neg x_1@3$ is implied. This new implication restarts the BCP process at level 3. Clause c_9 is a special kind of conflict clause, called an **asserting clause**: it forces an immediate implication after backtracking. Analyze-Conflict can be designed to generate asserting clauses only, as indeed most competitive solvers do.

After asserting $x_1 = 0$ the solver again reaches a conflict, as can be seen in the right drawing in Fig. 2.7. This time the conflict clause (x_2) is added, and the solver backtracks to decision level 0 and continues from there. Why (x_2)? The strategy of Analyze-Conflict in generating these clauses is explained later in Sect. 2.2.4, but observe for the moment how indeed $\neg x_2$ leads to a conflict through clauses c_6 and c_7, as can also be inferred from Fig. 2.7 (right).

Aside: Multiple Conflict Clauses

More than one conflict clause can be derived from a conflict graph. In the present example, the assignment $\{x_2 \mapsto 1,\ x_3 \mapsto 1\}$ is also a sufficient condition for the conflict, and hence $(\neg x_2 \vee \neg x_3)$ is also a conflict clause. A generalization of this observation requires the following definition.

Definition 2.7 (separating cut). *A* separating cut *in a conflict graph is a minimal set of edges whose removal breaks all paths from the root nodes to the conflict node.*

This definition is applicable to a full implication graph (see Definition 2.6), as well as to a partial graph focused on the decision level of the conflict. The cut bipartitions the nodes into the *reason* side (the side that includes all the roots) and the *conflict* side. The set of nodes on the reason side that have at least one edge to a node on the conflict side constitute a sufficient condition for the conflict, and hence their negation is a legitimate conflict clause. Different SAT solvers have different strategies for choosing the conflict clauses that they add: some add as many as possible (corresponding to many different cuts), while others try to find the most effective ones. Some, including most of the modern SAT solvers, add a single clause, which is an asserting clause (see below), for each conflict. Modern solvers also have a strategy for *erasing* conflict clauses: without this feature the memory is quickly filled with millions of clauses. A typical strategy is to measure the *activity* of each clause, and periodically erase clauses with a low activity score. The activity score of a clause is increased when it participates in inferring new clauses.

Conflict-driven backtracking raises several issues:

- *It seems to waste work*, because the partial assignments up to decision level 5 can still be part of a satisfying assignment. However, empirical evidence shows that conflict-driven backtracking, coupled with a conflict-driven decision heuristic such as VSIDS (discussed later in Sect. 2.2.5), performs very well. A possible explanation for the success of this heuristic is that the conflict encountered can influence the decision heuristic to decide values or variables different from those at deeper decision levels (levels 4 and 5 in this case). Thus, keeping the decisions and implications made before the new information (i.e., the new conflict clause) had arrived may skew the search to areas not considered best anymore by the heuristic. Some of the wasted work can be saved, however, by simply keeping the last assignment, and reusing it whenever the variable's value has to be decided again [261]. An extensive analysis of this technique can be found in [222].
- *Is this process guaranteed to terminate?* In other words, how do we know that a partial assignment cannot be repeated forever? The learned conflict clauses cannot be the reason, because in fact most SAT solvers erase many

of them after a while to prevent the formula from growing too much. The reason is the following:

Theorem 2.8. *It is never the case that the solver enters decision level dl again with the same partial assignment.*

Proof. Consider a partial assignment up to decision level $dl - 1$ that does not end with a conflict, and assume falsely that this state is repeated later, after the solver backtracks to some lower decision level dl^- ($0 \leq dl^- < dl$). Any backtracking from a decision level dl^+ ($dl^+ \geq dl$) to decision level dl^- adds an implication at level dl^- of a variable that was assigned at decision level dl^+. Since this variable has not so far been part of the partial assignment up to decision level dl, once the solver reaches dl again, it is with a different partial assignment, which contradicts our assumption. ▟

The (hypothetical) progress of a SAT solver based on this strategy is illustrated in Fig. 2.8. More details of this graph are explained in the caption.

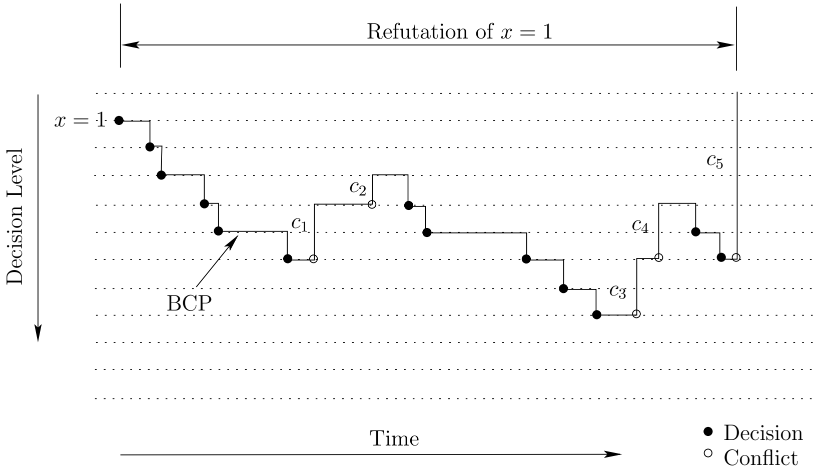


Fig. 2.8. Illustration of the progress of a SAT solver based on conflict-driven backtracking. Every conflict results in a conflict clause (denoted by $c_1, \ldots, c_5$ in the drawing). If the top left decision is $x = 1$, then this drawing illustrates the work done by the SAT solver to refute this wrong decision. Only some of the work during this time was necessary for creating c_5, refuting this decision, and computing the backtracking level. The "wasted work" (which might, after all, become useful later on) is due to the imperfection of the decision heuristic

2.2.4 Conflict Clauses and Resolution

Now consider Analyze-Conflict (Algorithm 2.2.2). The description of the algorithm so far has relied on the fact that the conflict clause generated is an asserting clause, and we therefore continue with this assumption when considering the termination criterion for line 3. The following definitions are necessary for describing this criterion:

```
Algorithm 2.2.2: ANALYZE-CONFLICT

Input:
Output: Backtracking decision level + a new conflict clause

1. if current-decision-level = 0 then return -1;
2. cl := current-conflicting-clause;
3. while (¬STOP-CRITERION-MET(cl)) do
4.     lit := LAST-ASSIGNED-LITERAL(cl);
5.     var := VARIABLE-OF-LITERAL(lit);
6.     ante := ANTECEDENT(lit);
7.     cl := RESOLVE(cl, ante, var);
8. add-clause-to-database(cl);
9. return clause-asserting-level(cl);        ▷ 2nd highest decision level in cl
```

Definition 2.9 (unique implication point (UIP)). *Given a partial conflict graph corresponding to the decision level of the conflict, a* unique implication point (UIP) *is any node other than the conflict node that is on all paths from the decision node to the conflict node.*

The decision node itself is a UIP by definition, while other UIPs, if they exist, are internal nodes corresponding to implications at the decision level of the conflict. In graph-theoretical terms, UIPs *dominate* the conflict node.

Definition 2.10 (first UIP). *A* first UIP *is a UIP that is closest to the conflict node.*

We leave the proof that the notion of a first UIP in a conflict graph is well defined as an exercise (see Problem 2.14). Figure 2.9 demonstrates UIPs in a conflict graph (see also the caption).

Empirical studies show that a good strategy for STOP-CRITERION-MET(*cl*) (line 3) is to return TRUE if and only if *cl* contains the negation of the first UIP as its single literal at the current decision level. This negated literal becomes asserted immediately after backtracking. There are several advantages to this strategy, which may explain the results of the empirical studies:

1. *The strategy has a low computational cost, compared with strategies that choose UIPs further away from the conflict.*
2. *It backtracks to the lowest decision level.*

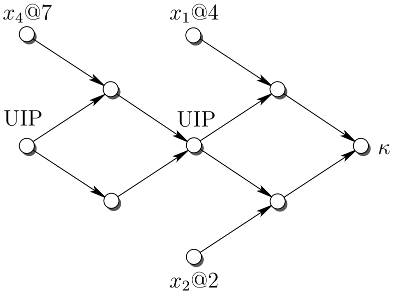


Fig. 2.9. An implication graph (stripped of most of its labels) with two UIPs. The left UIP is the decision node, and the right one is the first UIP, as it is the one closest to the conflict node

The second fact can be demonstrated with the help of Fig. 2.9. Let l_1 and l_2 denote the literals at the first and the second UIP, respectively. The asserting clauses generated with the first-UIP and second-UIP strategies are, respectively, $(\neg l_1 \vee \neg x_1 \vee \neg x_2)$ and $(\neg l_2 \vee \neg x_1 \vee \neg x_2 \vee \neg x_4)$. It is not a coincidence that the second clause subsumes the first, other than the asserting literals $\neg l_1$ and $\neg l_2$: it is always like this, by construction. Now recall how the backtracking level is determined: it is equal to the decision level corresponding to the second highest in the asserting clause. Clearly, this implies that the backtracking level computed with regard to the first clause is lower than that computed with regard to the second clause. In our example, these are decision levels 4 and 7, respectively.

In order to explain lines 4–7 of Analyze-Conflict, we need the following definition:

Definition 2.11 (binary resolution and related terms). *Consider the following inference rule:*

$$\frac{(a_1 \vee \ldots \vee a_n \vee \beta) \qquad (b_1 \vee \ldots \vee b_m \vee \neg\beta)}{(a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)} \quad (\textsc{Binary Resolution}) \ , \quad (2.9)$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are literals and β is a variable. The variable β is called the **resolution variable***. The clauses $(a_1 \vee \ldots \vee a_n \vee \beta)$ and $(b_1 \vee \ldots \vee b_m \vee (\neg\beta))$ are the* **resolving clauses***, and $(a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)$ is the* **resolvent clause***.*

A well-known result obtained by Robinson [243] shows that a deductive system based on the binary-resolution rule as its single inference rule is sound and complete. In other words, a CNF formula is unsatisfiable if and only if there exists a finite series of binary-resolution steps ending with the empty clause.

The function Resolve(c_1, c_2, v) used in line 7 of Analyze-Conflict returns the resolvent of the clauses c_1, c_2, where the resolution variable is v. The

Aside: Hard Problems for Resolution-Based Procedures
Some propositional formulas can be decided with no less than an exponential number of resolution steps in the size of the input. Haken [137] proved in 1985 that the **pigeonhole problem** is one such problem: given $n > 1$ pigeons and $n - 1$ pigeonholes, can each of the pigeons be assigned a pigeonhole without sharing? While a formulation of this problem in propositional logic is rather trivial with $n \cdot (n-1)$ variables, currently no SAT solver (which, recall, implicitly perform resolution) can solve this problem in a reasonable amount of time for n larger than several tens, although the size of the CNF itself is relatively small. As an experiment, we tried to solve this problem for $n = 20$ with four leading SAT solvers: Siege4 [248], zChaff-04 [202], HaifaSat [124], and Glucose-2014 [8]. On a Pentium 4 with 1 GB of main memory, none of them could solve this problem within three hours. Compare this result with the fact that, bounded by the same timeout, these tools routinely solve problems arising in industry with hundreds of thousands and even millions of variables.

The good news is that some SAT solvers now support a preprocessing step with which **cardinality constraints** (constraints of the form $\Sigma_i x_i \leq k$) are identified in the CNF, and solved by a separate technique. The pigeonhole problem implicitly uses a cardinality constraint of the form $\Sigma_i x_i \leq 1$ for each pigeonhole, to encode the fact that it can hold at most one pigeon, and indeed a SAT solver such as SAT4J, which supports this technique, can solve this problem even with $n = 200$ [32].

Antecedent function used in line 6 of this function returns *Antecedent*(*lit*). The other functions and variables are self-explanatory.

Analyze-Conflict progresses from right to left on the conflict graph, starting from the conflicting clause, while constructing the new conflict clause through a series of resolution steps. It begins with the conflicting clause *cl*, in which all literals are set to 0. The literal *lit* is the literal in *cl* assigned last, and *var* denotes its associated variable. The antecedent clause of *var*, denoted by *ante*, contains $\neg lit$ as the only satisfied literal, and other literals, all of which are currently unsatisfied. The clauses *cl* and *ante* thus contain *lit* and $\neg lit$, respectively, and can therefore be resolved with the resolution variable *var*. The resolvent clause is again a conflicting clause, which is the basis for the next resolution step.

Example 2.12. Consider the partial implication graph and set of clauses in Fig. 2.10, and assume that the implication order in the BCP was x_4, x_5, x_6, x_7.

The conflict clause $c_5 := (x_{10} \vee x_2 \vee \neg x_4)$ is computed through a series of binary resolutions. Analyze-Conflict traverses backwards through the implication graph starting from the conflicting clause c_4, while following the order of the implications in reverse, as can be seen in the table below. The

$$\begin{aligned} c_1 &= (\neg x_4 \lor x_2 \lor x_5) \\ c_2 &= (\neg x_4 \lor x_{10} \lor x_6) \\ c_3 &= (\neg x_5 \lor \neg x_6 \lor \neg x_7) \\ c_4 &= (\neg x_6 \lor x_7) \\ &\vdots \end{aligned}$$

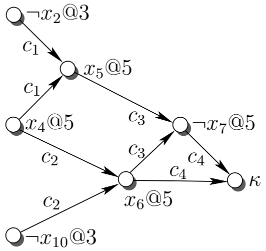


Fig. 2.10. A partial implication graph and a set of clauses that demonstrate Algorithm 2.2.2. The nodes are depicted so that their horizontal position is consistent with the order in which they were created. Algorithm 2.2.2 traverses the nodes in reverse order, from right to left. The first UIP it finds is x_4, and, correspondingly, the asserted literal is $\neg x_4$

intermediate clauses, in this case the second and third clauses in the resolution sequence, are typically discarded.

Name	*cl*	*lit*	*var*	*ante*
c_4	$(\neg x_6 \lor x_7)$	x_7	x_7	c_3
	$(\neg x_5 \lor \neg x_6)$	$\neg x_6$	x_6	c_2
	$(\neg x_4 \lor x_{10} \lor \neg x_5)$	$\neg x_5$	x_5	c_1
c_5	$(\neg x_4 \lor x_2 \lor x_{10})$			

The clause c_5 is an asserting clause in which the negation of the first UIP (x_4) is the only literal from the current decision level. ⌟

2.2.5 Decision Heuristics

Probably the most important element in SAT solving is the strategy by which the variables and the value given to them are chosen. This strategy is called the **decision heuristic** of the SAT solver. Let us survey some of the best-known decision heuristics, in the order in which they were suggested, which is also the order of their average efficiency as measured by numerous experiments. New strategies are published every year.

Jeroslow–Wang

Given a CNF formula $\mathcal{B}$, compute for each literal l

$$J(l) = \Sigma_{\omega \in \mathcal{B}, l \in \omega} 2^{-|\omega|} , \tag{2.10}$$

where ω represents a clause and $|\omega|$ its length. Choose the literal l for which $J(l)$ is maximal, and for which neither l or $\neg l$ is asserted.

This strategy gives higher priority to literals that appear frequently in short clauses. It can be implemented statically (one computation at the beginning of the run) or dynamically, where in each decision only unsatisfied clauses are considered in the computation. The dynamic approach produces better decisions, but also imposes large overhead at each decision point.

Dynamic Largest Individual Sum (DLIS)

At each decision level, choose the unassigned literal that satisfies the largest number of currently unsatisfied clauses.

The common way to implement such a heuristic is to keep a pointer from each literal to a list of clauses in which it appears. At each decision level, the solver counts the number of clauses that include this literal and are not yet satisfied, and assigns this number to the literal. Subsequently, the literal with the largest count is chosen. DLIS imposes a large overhead, since the complexity of making a decision is proportional to the number of clauses. Another variation of this strategy, suggested by Copty et al. [79], is to count the number of satisfied clauses resulting from each possible decision *and its implications through BCP*. This variation indeed makes better decisions, but also imposes more overhead.

Variable State Independent Decaying Sum (VSIDS) and Variants

This is a strategy that was introduced in the SAT solver Chaff [202], which is reminiscent of DLIS but far more efficient. First, when counting the number of clauses in which every literal appears, disregard the question of whether that clause is already satisfied or not. This means that the estimation of the quality of every decision is compromised, but the complexity of making a decision is better: it takes a constant time to make a decision assuming we keep the literals in a list sorted by their score. Second, periodically divide all scores by 2.

The idea is to make the decision heuristic **conflict-driven**, which means that it tries to solve recently discovered conflicts first. For this purpose, it needs to give higher scores to variables that are involved in recent conflicts. Recall that every conflict results in a conflict clause. A new conflict clause, like any other clause, adds 1 to the score of each literal that appears in it. The greater the amount of time that has passed since this clause was added, the more often the score of these literals is divided by 2. Thus, variables in new conflict clauses become more influential. The SAT solver Chaff, which introduced VSIDS, allows one to tune this strategy by controlling the frequency with which the scores are divided and the constant by which they are divided. It turns out that different families of CNF formulas are best solved with different parameters.

There are other conflict-driven heuristics. Consider, for example, the strategy adopted by the award-winning SAT solver MiniSAT. MiniSAT maintains an activity score for each variable (in the form of a floating-point number with double precision), which measures the involvement of each variable in inferring new clauses. If a clause c is inferred from clauses $c_1, \ldots, c_n$, then each instance of a variable v in $c_1, \ldots, c_n$ entails an increase in the score of v by some constant *inc*. *inc* is initially set to 1, and then multiplied by 1.05 after each conflict, thus giving higher score to variables that participate in recent conflicts. To prevent overflow, if the activity score of some variable is higher than 10^{100}, then all variable scores as well as *inc* are multiplied by 10^{-100}. The variable that has the highest score is selected. The value chosen for this variable is either FALSE, random, or, when relevant, the previous value that this variable was assigned. The fact that in such a successful solver as MiniSAT there is no attempt to guess the right value of a variable indicates that what matters is the locality of the search coupled with learning, rather than a correct guess of the branch. This is not surprising: most branches, even in satisfiable formulas, do not lead to a satisfying assignment.

Clause-Based Heuristics

In this family of heuristics, literals in recent conflict clauses are given absolute priority. This effect is achieved by traversing backwards the list of learned clauses each time a decision has to be made. We begin by describing in detail a heuristic called Berkmin.

Maintain a score per variable, similar to the score VSIDS maintains for each literal (i.e., increase the counter of a variable if one of its literals appears in a clause, and periodically divide the counters by a constant). Maintain a similar score for each literal, but do not divide it periodically. Push conflict clauses into a stack. When a decision has to be made, search for the topmost clause on this stack that is unresolved. From this clause, choose the unassigned variable with the highest variable score. Determine the value of this variable by choosing the literal corresponding to this variable with the highest literal score. If the stack is empty, the same strategy is applied, except that the variable is chosen from the set of all unassigned variables rather than from a single clause.

This heuristic was first implemented in a SAT solver called BERKMIN. The idea is to give variables that appear in recent conflicts absolute priority, which seems empirically to be more effective. It also concentrates only on unresolved conflicts, in contrast to VSIDS.

A different clause-based strategy is called Clause-Move-To-Front (CMTF). It is similar to Berkmin, with the difference that, at the time of learning a new clause, k clauses (k being a constant that can be tuned) that participated in resolving the new clause are pushed to the end of the list, just before the new clause. The justification for this strategy is that it keeps the search more focused. Suppose, for example, that a clause c is resolved from c_1, c_2, and c_3. We can write this as $c_1 \wedge c_2 \wedge c_3 \implies c$, which makes it clear that satisfying c is easier than satisfying $c_1 \wedge c_2 \wedge c_3$. Hence the current partial assignment contradicted $c_1 \wedge c_2 \wedge c_3$, the solver backtracked, and now tries to satisfy an easier formula, namely c, before returning to those three clauses. CMTF is implemented in a SAT solver called HaifaSat [124], and a variation of it called Clause-Based Heuristic (CBH) is implemented in Eureka [99].

2.2.6 The Resolution Graph and the Unsatisfiable Core

Since each conflict clause is derived from a set of other clauses, we can keep track of this process with a **resolution graph**.

Definition 2.13 (binary resolution graph). *A* binary resolution graph *is a directed acyclic graph where each node is labeled with a clause, each root corresponds to an original clause, and each nonroot node has exactly two incoming edges and corresponds to a clause derived by binary resolution from its parents in the graph.*

Typically, SAT solvers do not retain all the intermediate clauses that are created during the resolution process of the conflict clause. They store enough clauses, however, for building a graph that describes the relation between the conflict clauses.

Definition 2.14 (hyper-resolution graph). *A* hyper-resolution graph *is a directed acyclic graph where each node is labeled with a clause, each root corresponds to an original clause, and each nonroot node has two or more incoming edges and corresponds to a clause derived by binary resolution from its parents in the graph, possibly through other clauses that are not represented in the graph.*

Example 2.15. Consider once again the implication graph in Fig. 2.10. The clauses $c_1, \ldots, c_4$ participate in the resolution of c_5. The corresponding resolution graph appears in Fig. 2.11. ▟

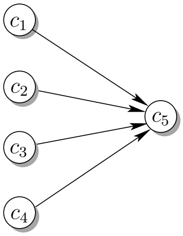


Fig. 2.11. A hyper-resolution graph corresponding to the implication graph in Fig. 2.10

In the case of an unsatisfiable formula, the resolution graph has a sink node (i.e., a node with incoming edges only), which corresponds to an empty clause.[3]

The resolution graph can be used for various purposes, some of which we mention here. The most common use of this graph is for deriving an *unsatisfiable core* of unsatisfiable formulas.

Definition 2.16 (unsatisfiable core). *An* unsatisfiable core *of a CNF unsatisfiable formula is any unsatisfiable subset of the original set of clauses.*

Unsatisfiable cores which are relatively small subsets of the original set of clauses are useful in various contexts, because they help us to focus on a *cause* of unsatisfiability (there can be multiple unsatisfiable cores not contained in each other, and not even intersecting each other). We leave it to the reader in Problem 2.17 to find an algorithm that computes a core given a resolution graph.

Another common use of a resolution graph is for certifying a SAT solver's conclusion that a formula is unsatisfiable. Unlike the case of satisfiable instances, for which the satisfying assignment is an easy-to-check piece of evidence, checking an unsatisfiability result is harder. Using the resolution graph, however, an independent checker can replay the resolution steps starting from the original clauses until it derives the empty clause. This verification requires time that is linear in the size of the resolution proof.

2.2.7 Incremental Satisfiability

In numerous industrial applications the SAT solver is a component in a bigger system that sends it satisfiability queries. For example, a program that plans a path for a robot may use a SAT solver to find out if there exists a path within k steps from the current state. If the answer is negative, it increases k and tries again. The important point here is that the sequence of formulas that the SAT solver is asked to solve is not arbitrary: these formulas have a lot in common. Can we use this fact to make the SAT solver run faster? We should somehow reuse information that was gathered in previous instances to expedite the solution of the current one. To make things simpler, consider two CNF formulas, C_1 and C_2, which are solved consecutively, and assume that C_2 is known at the time of solving C_1. Here are two kinds of information that can be reused when solving C_2:

- *Reuse clauses.* We should answer the following question: if c is a conflict clause learned while solving C_1, under what conditions are C_2 and $C_2 \wedge c$ equisatisfiable? It is easier to answer this question if we view C_1 and C_2

[3] In practice, SAT solvers terminate before they actually derive the empty clause, as can be seen in Algorithms 2.2.1 and 2.2.2, but it is possible to continue developing the resolution graph after the run is over and derive a full resolution proof ending with the empty clause.

as sets of clauses. Let C denote the clauses in the intersection $C_1 \cap C_2$. Any clause learnt solely from C clauses can be reused when solving C_2. In practice, as in the path planning problem mentioned above, consecutive formulas in the sequence are very similar, and hence C_1 and C_2 share the vast majority of their clauses, which means that most of what was learnt can be reused. We leave it as an exercise (Problem 2.16) to design an algorithm that discovers the clauses that can be reused.

- *Reuse heuristic parameters.* Various weights are updated during the solving process, and used to heuristically guide the search, e.g., variable score is used in decision making (Sect. 2.2.5), weights expressing the activity of clauses in deriving new clauses are used for determining which learned clauses should be maintained and which should be deleted, etc. If C_1 and C_2 are sufficiently similar, starting to solve C_2 with the weights at the end of the solving process of C_1 can expedite the solving of C_2.

To understand how modern SAT solvers support incremental solving, one should first understand a mechanism called **assumptions**, which was introduced with the SAT solver MiniSAT [110]. Assumptions are literals that are known to hold when solving C_1, but may be removed or negated when solving C_2. The list of assumption literals is passed to the solver as a parameter. The solver treats assumptions as special literals that dictate the initial set of decisions. If the solver backtracks beyond the decision level of the last assumption, it declares the formula to be unsatisfiable, since there is no solution without changing the assumptions. For example, suppose $a_1, \ldots, a_n$ are the assumption literals. Then the solver begins by making the decisions $a_1 = \text{TRUE}, \ldots, a_n = \text{TRUE}$, while applying BCP as usual. If at any point the solver backtracks to level n or less, it declares the formula to be unsatisfiable.

The key point here is that all clauses that are learnt are *independent of the assumptions* and can therefore be reused when these assumptions no longer hold. This is the nature of learning: it learns clauses that are independent of specific decisions, and assumptions are just decisions. Hence, we can start solving C_2 while maintaining all the clauses that were learnt during the solving process of C_1. Note that in this way we reuse both types of information mentioned above, and save the time of reparsing the formula.

We now describe how assumptions are used for solving the general incremental SAT problem, which requires both addition and deletion of clauses between instances. As for adding clauses, the solver receives the set of clauses that should be added ($C_2 \setminus C_1$ in our case) as part of its interface. Removing clauses is done by adding a new assumption literal (corresponding to a *new* variable) to every clause $c \in (C_1 \setminus C_2)$, negated. For example, if $c = (x_1 \vee x_2)$, then it is replaced with $c' = (\neg a \vee x_1 \vee x_2)$, where a is a new variable. Note that under the assumption $a = \text{TRUE}$, $c = c'$, and hence the added assumption literal does not change the satisfiability of the formula. When solving C_2, however, we replace that assumption with the assumption $a = \text{FALSE}$, which is equivalent to erasing the clause c. Assumption literals used in this way are called **clause selectors**.

2.2.8 From SAT to the Constraint Satisfaction Problem

In parallel to the research on the SAT problem, there has been a lot of research on the Constraint Satisfaction Problem (CSP) [98], with a lot of cross-fertilization between these two fields. CSP allows arbitrary constraints over variables with finite discrete domains. For example, a CSP instance can be defined by variable domains $x_1 \in \{1 \ldots 10\}$, $x_2 \in \{2, 4, 6, \ldots, 30\}$, $x_3 \in \{-5, -4, \ldots, 5\}$ and a Boolean combination of constraints over these variables

$$\textsc{AllDifferent}(x_1, x_2, x_3) \wedge x_1 < x_3 \; . \tag{2.11}$$

The AllDifferent constraint means that its arguments must be assigned different values. Modern CSP solvers support dozens of such constraints. A propositional formula can be seen as a special case of a CSP model, which does not use constraints, other than the Boolean connectives, and the domains of the variables are limited to $\{0, 1\}$.

CSP solving is an NP problem, and hence can be reduced to SAT in polynomial time.[4] Since the domains are restricted to finite discrete domains, "flattening" them to propositional logic requires some work. For example, a variable with a domain $\{1, \ldots, n\}$ can be encoded with $\lceil \log(n) \rceil$ propositional variables. If there are "holes" in the domain, then additional constraints are needed on the values that these variables can be assigned. Similarly, the constraints should be translated to propositional formulas. For example, if x_1 is encoded with propositional variables $b_1, \ldots, b_5$ and x_2 with $c_1, \ldots c_5$, then the constraint $x_1 \neq x_2$ can be cast in propositional logic as $\vee_{i=1}^{5} (b_i \vee c_i) \wedge (\neg b_i \vee \neg c_i)$, effectively forcing at least one of the bits to be different. AllDifferent is then just a pairwise disequality of all its arguments. Additional bitwise (logarithmic) translation methods appear in Chap. 6, whereas a simpler linear-size translation is the subject of Problem 2.11. Indeed, some of the competitive CSP solvers are just *front-end* utilities that translate the CSP to SAT, either up-front, or lazily. Other solvers handle the constraints directly.

A description of how CSP solvers work is beyond the scope of this book. We will only say that they can be built with a core similar to that of SAT, including decision making, constraint propagation, and learning. Each type of constraint has to be handled separately, however. CSP solvers are typically modular, and hence adding one's favorite constraint to an existing CSP solver is not difficult. Most of the work is in defining a *propagator* for the constraint. A propagator of a constraint c is a function that, given c and the current

[4] This complexity result is based on the set of constraints that is typically supported by CSP solvers. Obviously it is possible to find constraints that will push CSP away from NP.

domains of the variables, can (a) infer reductions in domains that are implied by c, and (b) detect that c is conflicting, i.e., it cannot be satisfied in the current domains. In the example above, a propagator for the $<$ constraint should conclude from $x_1 < x_3$ that the domain of x_1 should be reduced to $\{1, \ldots, 4\}$, because higher values are not *supported* by the current domain of x_3. In other words, for an assignment such as $x_1 = 5$, no value in the current domain of x_3, namely $\{-5, \ldots, 5\}$, can satisfy $x_1 < x_3$. As another example, suppose we have the CSP

$$\begin{array}{l} x_1 \in \{0,1\}, \quad x_2 \in \{0,1\}, \quad x_3 \in \{0,1\} \\ \textsc{AllDifferent}(x_1, x_2, x_3) \,. \end{array}$$

The propagator of AllDifferent should detect that the constraint cannot be satisfied under these domains. The equivalent of a propagator in SAT is BCP—see Sect. 2.2.3. Propagators must be sound, but for some constraints it is too computationally expensive to make them complete (see Definition 1.6). In other words, it may not find all possible domain reductions, and may not always detect a conflict under a partial assignment. This does not change the fact that the overall solver *is* complete, because it *does* detect a conflict with a full assignment.

Given a constraints problem, there are two potential advantages to modeling it as a CSP, rather than in propositional logic:

- CSP as a modeling language is far more readable and succinct, and
- When using a CSP solver that is *not* based on reduction to SAT, one may benefit from the fact that some constraints, such as AllDifferent, have polynomial-time precise propagators, whereas solving the same constraint with SAT after it is reduced to propositional logic is worst-case exponential.[5]

Typically these two potential advantages do not play a major role, however, because (a) most problems that are solved in industry are generated automatically, and hence readability is immaterial, and (b) realistic constraints problems mix many types of constraints, and hence solving them remains exponential. The current view is that neither CSP nor SAT dominate the other in terms of run time, and it is more a question of careful engineering than something substantial.

2.2.9 SAT Solvers: Summary

In this section we have covered the basic elements of modern CDCL solvers, including decision heuristics, learning with conflict clauses, and conflict-driven backtracking. There are various other mechanisms for gaining efficiency that

[5] Certain constraints that have a polynomial propagator can be translated to a specific form of propositional formula that, with the right strategy of the SAT solver, can be solved in polynomial time as well—see [220].

we do not cover in this book, such as efficient implementation of BCP, detection of subsumed clauses, preprocessing and simplification of the formula, deletion of conflict clauses, and **restarts** (i.e., restarting the solver when it seems to be in a hopeless branch of the search tree). The interested reader is referred to the references given in Sect. 2.4.

Let us now reflect on the two approaches to formal reasoning that we described in Sect. 1.1—deduction and enumeration. Can we say that SAT solvers, as described in this section, follow either one of them? On the one hand, SAT solvers can be thought of as searching a binary tree with 2^n leaves, where n is the number of Boolean variables in the input formula. Every leaf is a full assignment, and, hence, traversing all leaves corresponds to enumeration. From this point of view, conflict clauses are generated in order to prune the search space. On the other hand, conflict clauses are *deduced* via the resolution rule from other clauses. If the formula is unsatisfiable, then the sequence of applications of this rule, as listed in the SAT solver's log, is a deductive proof of unsatisfiability. The search heuristic can therefore be understood as a strategy of applying an inference rule—searching for a proof. Thus, the two points of view are equally legitimate.

2.3 Problems

2.3.1 Warm-up Exercises

Problem 2.1 (propositional logic: practice). For each of the following formulas, determine if it is satisfiable, unsatisfiable or valid:

1. $(p \implies (q \implies p))$
2. $(p \land q) \land (a \implies q) \land (b \implies \neg p) \land (a \lor b)$
3. $(p \lor q) \land (\neg p \lor \neg q) \land (p \lor \neg q) \land (\neg p \lor q)$
4. $(a \implies \neg a) \implies \neg a$
5. $(((a \implies p) \land (b \implies q)) \land (\neg a \lor \neg b)) \implies \neg(p \land q)$
6. $(a \land b \land c) \land (a \oplus b \oplus c) \land (a \lor b \lor c)$
7. $(a \land b \land c \land d) \land (a \oplus b \oplus c \oplus d) \land (a \lor b \lor c \lor d)$

where $\oplus$ denotes the XOR operator.

Problem 2.2 (modeling: simple). Consider three persons A, B, and C who need to be seated in a row, but:

- A does not want to sit next to C.
- A does not want to sit in the left chair.
- B does not want to sit to the right of C.

Write a propositional formula that is satisfiable if and only if there is a seat assignment for the three persons that satisfies all constraints. Is the formula satisfiable? If so, give an assignment.

Problem 2.3 (modeling: program equivalence). Show that the two `if-then-else` expressions below are equivalent:

!(a || b) ? h : !(a == b) ? f : g   !(!a || !b) ? g : (!a && !b) ? h : f

You can assume that the variables have only one bit.

Problem 2.4 (SAT solving). Consider the following set of clauses:

$$\begin{array}{ll} (x_5 \vee \neg x_1 \vee x_3)\ , & (\neg x_1 \vee x_2)\ , \\ (\neg x_2 \vee x_4)\ , & (\neg x_3 \vee \neg x_4)\ , \\ (\neg x_5 \vee x_1)\ , & (\neg x_5 \vee \neg x_6)\ , \\ (x_6 \vee x_1)\ . & \end{array} \tag{2.12}$$

Apply the VSIDS decision heuristic and ANALYZE-CONFLICT with conflict-driven backtracking. In the case of a tie (during the application of VSIDS), make a decision that eventually leads to a conflict. Show the implication graph at each decision level.

2.3.2 Propositional Logic

Problem 2.5 (propositional logic: NAND and NOR). Prove that any propositional formula can be equivalently written with

- only a NAND gate,
- only a NOR gate.

2.3.3 Modeling

Problem 2.6 (a reduction from your favorite NP-C problem). Show a reduction to propositional logic from your favorite NP-complete problem (not including SAT itself and problems that appear below). A list of famous NP-complete problems can be found online (some examples are: *vertex-cover*, *hitting-set*, *set-cover*, *knapsack*, *feedback vertex set*, *bin-packing*...). Note that many of those are better known as optimization problems, so begin by formulating a corresponding decision problem. For example, the optimization variant of *vertex-cover* is: find the minimal number of vertices that together touch all edges; the corresponding decision problem is: given a natural number k, is it possible to touch all edges with k vertices?

Problem 2.7 (unwinding a finite automaton). A *nondeterministic finite automaton* is a 5-tuple $\langle Q, \Sigma, \delta, I, F\rangle$, where

- Q is a finite set of states,
- Σ is the alphabet (a finite set of letters),
- $\delta : Q \times \Sigma \mapsto 2^Q$ is the transition function (2^Q is the power set of Q),
- $I \subseteq Q$ is the set of initial states, and

- $F \subseteq Q$ is the set of accepting states.

The transition function determines to which states we can move given the current state and input. The automaton is said to *accept* a finite input string $s_1, \ldots, s_n$ with $s_i \in \Sigma$ if and only if there is a sequence of states $q_0, \ldots, q_n$ with $q_i \in Q$ such that

- $q_0 \in I$,
- $\forall i \in \{1, \ldots, n\}.\ q_i \in \delta(q_{i-1}, s_i)$, and
- $q_n \in F$.

For example, the automaton in Fig. 2.12 is defined by $Q = \{s_1, s_2\}$, $\Sigma = \{a, b\}$, $\delta(s_1, a) = \{s_1\}$, $\delta(s_1, b) = \{s_1, s_2\}$, $I = \{s_1\}$, $F = \{s_2\}$, and accepts strings that end with b. Given a nondeterministic finite automaton $\langle Q, \Sigma, \delta, I, F\rangle$ and a fixed input string $s_1, \ldots, s_n$, $s_i \in \Sigma$, construct a propositional formula that is satisfiable if and only if the automaton accepts the string.

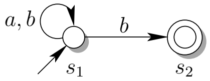


Fig. 2.12. A nondeterministic finite automaton accepting all strings ending with the letter b

Problem 2.8 (assigning teachers to subjects). A problem of covering m subjects with k teachers may be defined as follows. Let $T : \{T_1, \ldots, T_n\}$ be a set of teachers. Let $S : \{S_1, \ldots, S_m\}$ be a set of subjects. Each teacher $t \in T$ can teach some subset $S(t)$ of the subjects S (i.e., $S(t) \subseteq S$). Given a natural number $k \leq n$, is there a subset of size k of the teachers that together covers all m subjects, i.e., a subset $C \subseteq T$ such that $|C| = k$ and $(\bigcup_{t \in C} S(t)) = S$?

Problem 2.9 (Hamiltonian cycle). Show a formulation in propositional logic of the following problem: given a directed graph, does it contain a Hamiltonian cycle (a closed path that visits each node, other than the first, exactly once)?

2.3.4 Complexity

Problem 2.10 (space complexity of CDCL with learning). What is the worst-case space complexity of a CDCL SAT solver as described in Sect. 2.2, in the following cases:

(a) Without learning
(b) With learning, i.e., by recording conflict clauses
(c) With learning in which the length of the recorded conflict clauses is bounded by a natural number k

Problem 2.11 (from CSP to SAT). Suppose we are given a CSP over some unified domain $\mathcal{D} : [min..max]$ where all constraints are of the form $v_i \leq v_j, v_i - v_j \leq c$ or $v_i = v_j + c$ for some constant c. For example

$$((v_2 \leq v_3) \vee (v_4 \leq v_1)) \wedge v_2 = v_1 + 4 \wedge v_4 = v_3 + 3$$

for $v_1, v_2, v_3, v_4 \in [0..7]$ is a formula belonging to this fragment. This formula is satisfied by one of two solutions: $(v_1 \mapsto 0, v_2 \mapsto 4, v_3 \mapsto 4, v_4 \mapsto 7)$, or $(v_3 \mapsto 0, v_1 \mapsto 3, v_4 \mapsto 3, v_2 \mapsto 7)$.

Show a reduction of such formulas to propositional logic. Hint: an encoding which requires $|V| \cdot |\mathcal{D}|$ propositional variables, where $|V|$ is the number of variables and $|\mathcal{D}|$ is the size of the domain, is given by introducing a propositional variable b_{ij} for each variable $v_i \in V$ and $j \in \mathcal{D}$, which indicates that $v_i \leq j$ is true.

For the advanced reader: try to find a logarithmic encoding.

Problem 2.12 (polynomial-time (restricted) SAT). Consider the following two restrictions of CNF:

- A CNF in which there is not more than one positive literal in each clause.
- A CNF formula in which no clause has more than two literals.

1. Show a polynomial-time algorithm that solves each of the problems above.
2. Show that every CNF can be converted to another CNF which is a conjunction of the two types of formula above. In other words, in the resulting formula all the clauses are either unary, binary, or have not more than one positive literal. How many additional variables are necessary for the conversion?

2.3.5 CDCL SAT Solving

Problem 2.13 (backtracking level). We saw that SAT solvers working with conflict-driven backtracking backtrack to the second highest decision level dl in the asserting conflict clause. This wastes all of the work done from decision level $dl + 1$ to the current one, say dl' (although, as we mentioned, this has other advantages that outweigh this drawback). Suppose we try to avoid this waste by performing conflict-driven backtracking as usual, but then repeat the assignments from levels $dl + 1$ to $dl' - 1$ (i.e., override the standard decision heuristic for these decisions). Can it be guaranteed that this reassignment will progress without a conflict?

Problem 2.14 (is the first UIP well defined?). Prove that, in a conflict graph, the notion of a first UIP is well defined, i.e., there is always a single UIP closest to the conflict node. Hint: you may use the notion of *dominators* from graph theory.

2.3.6 Related Problems

Problem 2.15 (blocked clauses). Let φ be a CNF formula, let $c \in \varphi$ be a clause such that $l \in c$ where l is a literal, and let $\varphi_{\neg l} \subseteq \varphi$ be the subset of $\varphi's$ clauses that contain $\neg l$. We say that c is *blocked* by l if the resolution of c with any clause in $\varphi_{\neg l}$ using $var(l)$ as the pivot is a tautology. For example, if $c = (l \vee x \vee y)$ and $\varphi_{\neg l}$ has a single clause $c' = (\neg l \vee \neg x \vee z)$, then resolving c and c' on l results in $(x \vee \neg x \vee y \vee z)$, which is a tautology, and hence c is blocked by l. Prove that φ is equisatisfiable to $\varphi \setminus c$, i.e., blocked clauses can be removed from φ without affecting its satisfiability.

Problem 2.16 (incremental satisfiability). In Sect. 2.2.7 we saw a condition for sharing clauses between similar instances. Suggest a way to implement this check, i.e., how can a SAT solver detect those clauses that were inferred from clauses that are common to both instances? The solution cannot use the mechanism of assumptions.

Problem 2.17 (unsatisfiable cores).

(a) Suggest an algorithm that, given a resolution graph (see Definition 2.14), finds an unsatisfiable core of the original formula that is as small as possible (by this we do not mean that it has to be minimal).
(b) Given an unsatisfiable core, suggest a method that attempts to minimize it further.

Problem 2.18 (unsatisfiable cores and transition clauses). Let $\mathcal{B}$ be an unsatisfiable CNF formula, and let c be a clause of $\mathcal{B}$. If removing c from $\mathcal{B}$ makes $\mathcal{B}$ satisfiable, we say that c is a **transition clause**. Prove the following claim: all transition clauses of a formula are contained in each of its unsatisfiable cores.

2.4 Bibliographic Notes

The very existence of the 980-pages *Handbook of Satisfiability* [35] from 2009, which covers all the topics mentioned in this chapter and much more, indicates what a small fraction of SAT can be covered here. More recently Donald Knuth dedicated almost 300 pages to this topic in his book series *The Art of Computer Programming* [168]. Some highlights from the history of SAT are in order nevertheless.

The Davis–Putnam–Loveland–Logemann (DPLL) framework was a two-stage invention. In 1960, Davis and Putnam considered CNF formulas and offered a procedure to solve them based on an iterative application of three rules [88]: the pure literal rule, the unit clause rule (what we now call BCP), and what they called "the elimination rule", which is a rule for eliminating a variable by invoking resolution (e.g., to eliminate x from a given CNF, apply resolution to each pair of clauses of the form $(x \vee A) \wedge (\neg x \vee B)$, erase the resolving clauses, and maintain the resolvent). Their motivation was to optimize a previously known incomplete technique for deciding first-order formulas. Note that, at the time, "optimizing" also meant a procedure that was easier to conduct by hand. In 1962, Loveland and Logemann, two programmers hired by Davis and Putnam to implement their idea, concluded that it was more efficient to split and backtrack rather than to apply resolution, and together with Davis published what we know today as the basic DPLL framework [87]. The SAT community tends to distinguish modern solvers from those based on DPLL by referring to them as Conflict-Driven Clause Learning (CDCL) solvers, which emphasizes their learning capability combined with nonchronological backtracking, and the fact that their search is strongly tied to the learning scheme, via a heuristic such as VSIDS. The fact that modern solvers restart the search very frequently adds another major distinction from the earlier DPLL solvers. The main alternative to DPLL/CDCL are the stochastic solvers, also called **local-search** SAT solvers, which were not discussed at length in this chapter. For many years they were led by the GSAT and WalkSat solvers [254]. There are various solvers that combine local search with learning and propagation, such as UnitWalk [143].

The development of SAT solvers has always been influenced by developments in the world of Constraint Satisfaction Problem (CSP), a problem which generalizes SAT to arbitrary finite discrete domains and arbitrary constraints. The definition of CSP by Montanari [201] (and even before that by Waltz in 1975), and the development of efficient CSP solvers, led to cross-fertilization between the two fields: nonchronological backtracking, for example, was first used in CSP, and then adopted by Marques-Silva and Sakallah for their GRASP SAT solver [262], which was the fastest from 1996 to 2000. In addition, learning via conflict clauses in GRASP was inspired by CSP's *no-good recording*. Bayardo and Schrag [21] also published a method for adapting conflict-driven learning to SAT. CSP solvers have an annual competition, called the MiniZinc challenge. The winner of the 2016 competition in the free (unrestricted) search category is Michael Veksler's solver HaifaCSP [279].

The introduction of Chaff in 2001 [202] by Moskewicz, Madigan, Zhao, Zhang, and Malik marked a breakthrough in performance that led to renewed interest in the field. Chaff introduced the idea of conflict-driven nonchronological backtracking coupled with VSIDS, the first conflict-driven decision heuristic. It also included a new mechanism for performing fast BCP based on a data structure called two-watched literals, which is now standard in all competitive solvers. The description of the main SAT procedure in this chapter was inspired mainly by works related to Chaff [298, 299]. Berkmin, a SAT solver developed by Goldberg and Novikov, introduced what we have named "the Berkmin decision heuristic" [133]. The solver Siege introduced Variable-Move-To-Front (VMTF), a decision heuristic that moves a constant number of variables from the conflict clause to the top of the list, which performs very well in practice [248]. MiniSAT [110], a minimalistic open-source solver by

Niklas Eén and Niklas Sörensson, has not only won several competitions in the last decade, but also became a standard platform for SAT research. In the last few competitions there was even a special track for variants of MiniSAT. Starting from 2009, Glucose [8] seems to be one of the leading solvers. It introduced a technique for predicting the quality of a learned clause, based on the number of decision levels that it contains. When the solver erases some of the conflict clauses as most solvers do periodically, this measure improves its chances of keeping those that have a better chance of participating in further learning. The series of solvers by Armin Biere, PicoSAT [30], PrecoSAT, and Lingeling [31] have also been very dominant in the last few years and include dozens of new optimizations. We will only mention here that they are designed for very large CNF instances, and contain accordingly **inprocessing**, i.e., linear-time simplifications of the formula that are done periodically during the search. The simplifications are a selected subset of those that are typically done only as a preprocessing phase. Another very successful technique called **cube-and-conquer** [140] partitions the SAT problem into possibly millions of much easier ones. The challenge is of course to know how to perform this partitioning such that the total run time is reduced. In the first phase, it finds consistent *cubes*, which are simply partial assignments (typically giving values to not more than 10% of the variables); heuristics that are relatively computationally expensive are used in this phase, e.g., checking the impact of each possible assignment on the size of the remaining formula before making the decision. Such an expensive procedure is not competitive if applied throughout the solving process, but here it is used only for identifying relatively short cubes. In the second phase it uses a standard SAT solver to solve the formula after being simplified by the cube. This type of strategy is very suitable for parallelization, and indeed the solver Treengeling, also by Biere, is a highly efficient cube-and-conquer parallel solver. New SAT solvers are introduced every year; readers interested in the latest tools should check the results of the annual SAT competitions.

The realization that different classes of problems are best solved with different solvers led to a strategy of invoking an **algorithm portfolio**. This means that one out of n predefined solvers is chosen automatically for a given problem instance, based on a prediction of which solver is likely to perform best. First, a large “training set” is used for building **empirical hardness models** [212] based on various features of the instances in this set. Then, given a problem instance, the run time of each of the n solvers is predicted, and accordingly the solver is chosen for the task. SATzilla [289] is a successful algorithm portfolio based on these ideas that won several categories in the 2007 competition. Even without the learning phase and the automatic selection of solvers, running different solvers in parallel and reporting the result of the first one to find a solution is a very powerful technique. A parallel version of Lingeling, for example, called Plingeling [31], won the SAT competition in 2010 for parallel solvers, and was much better than any of the single-core ones. It simply runs Lingeling on several cores but each with a different random seed, and with slightly different parameters that affect its preprocessing and tie-breaking strategies. The various threads only share learned unit clauses.

The connection between the process of deriving conflict clauses and resolution was discussed in, for example, [22, 122, 180, 295, 298]. Zhang and Malik described a procedure for efficient extraction of unsatisfiable cores and unsatisfiability proofs from a SAT solver [298, 299]. There are many algorithms for minimizing such cores—see, for example, [123, 150, 184, 214]. A variant of the minimal unsatisfiable core (MUC) problem is called the **high-level minimal unsatisfiable core** (HLMUC) problem [203, 249]. The input to the problem, in addition to the CNF $\mathcal{B}$, is a set of sets of clauses from $\mathcal{B}$. Rather than minimizing the core, here the problem is to minimize the number of such sets that have a nonempty intersection with the core. This problem has various applications in formal verification as described in the above references.

Incremental satisfiability in its modern version, i.e., the problem of which conflict clauses can be reused when solving a related problem (see Problem 2.16), was introduced by Strichman in [260, 261] and independently by Whittemore, Kim, and Sakallah in [281]. Earlier versions of this problem were more restricted, for example, the work of Hooker [148] and of Kim, Whittemore, Marques-Silva, and Sakallah [164].

There is a very large body of theoretical work on SAT as well. Some examples are: in complexity, SAT was the problem that was used for establishing the NP-complete complexity class by Cook in 1971 [77]; in proof complexity, there is a large body of work on various characteristics of resolution proofs [23] and various restrictions and extensions thereof. In statistical mechanics, physicists study the nature of random formulas [67, 197, 48]: for a given number of variables n, and a given fixed clause size k, a clause is randomly generated by choosing, at uniform, from the $\binom{n}{k} \cdot 2^k$ options. A formula φ is a conjunction of $\alpha \cdot n$ random clauses. It is clear that when $\alpha \to \infty$, φ is unsatisfiable, and when $\alpha = 0$, it is satisfiable. At what value of α is the probability of φ being satisfiable 0.5? The answer is $\alpha = 4.267$. It turns out that formulas constructed with this ratio tend to be the hardest to solve empirically. Furthermore, the larger n is, the sharper the phase transition between SAT and UNSAT, asymptotically reaching a step function, i.e., all formulas with $\alpha > 4.267$ are unsatisfiable, whereas all formulas with $\alpha < 4.267$ are satisfiable. This shift from SAT to UNSAT is called a **phase transition**. There have been several articles about this topic in *Science* [166, 196], *Nature* [200], and even *The New York Times* [157].

Let us conclude these notes by mentioning that, in the first edition of this book, this chapter included about 10 pages on Ordered Binary Decision Diagrams (OBDDs) (frequently called BDDs for short). BDDs were invented by Randal Bryant [55] and were extremely influential in various fields in computer science, most notably in automated theorem proving, symbolic model checking, and other subfields of formal verification. With BDDs one can represent and manipulate propositional formulas. A key benefit of BDDs is the fact that they are *canonical* as long as the BDDs are built following the same variable order, which means that logically equivalent propositional formulas have identical BDDs. If the BDD is not empty, then the formula is trivially satisfiable, which means that once the BDD is built the satisfiability problem can be solved in constant time. The catch is that the construction itself can take exponential space and time, and indeed in practice nowadays CDCL-based SAT is generally better at solving satisfiability problems. Various SAT-based techniques such as Craig interpolants [193] and Property-Directed Reachability [42] mostly replaced BDDs in verification, although BDD-based engines are still used in commercial model checkers. BDDs also find uses in solving other problems, such as precise existential quantification of propositional formulas, *counting* the number of solutions a propositional formula has (an operation that can be done in linear time in the size of the BDD), and more. Knuth dedicated a large chapter to this topic in *The Art of Computer Programming* [167].

2.5 Glossary

The following symbols were used in this chapter:

Symbol	**Refers to ...**	**First used on page ...**
$x_i@d$	(SAT) x_i is assigned TRUE at decision level d	33

3

From Propositional to Quantifier-Free Theories

3.1 Introduction

In the previous chapter we studied CDCL-based procedures for deciding propositional formulas. Suppose, now, that instead of propositional variables we have other predicates, such as equalities and disequalities over the reals, e.g.,

$$(x_1 = x_2 \vee x_1 = x_3) \wedge (x_1 = x_2 \vee x_1 = x_4) \wedge x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4 \; . \tag{3.1}$$

Or, perhaps we would like to decide a Boolean combination of linear-arithmetic predicates:

$$((x_1 + 2x_3 < 5) \vee \neg(x_3 \leq 1) \wedge (x_1 \geq 3)) \; , \tag{3.2}$$

or a formula over arrays:

$$(i = j \wedge a[j] = 1) \wedge \neg(a[i] = 1) \; . \tag{3.3}$$

Equalities, linear predicates, arrays... is there a general framework to define them? Of course there is, and it is called first-order logic. Each of the above examples is a formula in some quantifier-free fragment of a first-order *theory*. Let us recall some basic terminology that was introduced already in Sect. 1.4. Generally such formulas can use propositional connectives and a set Σ of additional function and predicate symbols that uniquely define the theory T—indeed Σ is called the *signature* of T.[1] A decision procedure for T can decide the validity of T-formulas. Accordingly such formulas can be characterized as being T-valid, T-satisfiable (also called T-consistent), etc. We refer the reader to Sect. 1.4 for a more elaborate discussion of these matters.

[1] In this book we only consider signatures with commonly used symbols (e.g., "+", "*", "<") and assume that they are interpreted in the standard way (e.g., the "+" symbol corresponds to addition). Hence, the interpretation of the symbols in Σ is fixed.

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_3

In this chapter we study a general method—a framework, really—that generalizes CDCL to a decision procedure for decidable quantifier-free first-order theories, such as those above.[2] The method is commonly referred to as DPLL(T), emphasizing that it is parameterized by a theory T. The fact that it is called DPLL(T) and not CDCL(T) is attributed to historical reasons only: it is based on modern CDCL solvers (see Sect. 2.4 for a discussion on the differences). It is implemented in most **Satisfiability Modulo Theories** (SMT) solvers. In the case of (3.1), for example, T is simply the theory of equality (see Chap. 4). DPLL(T) is based on an interplay between a SAT solver and a decision procedure DP_T for the conjunctive fragment of T, i.e., formulas which are a *conjunction* of T-literals.

$\boxed{DP_T}$

The following example demonstrates the existence of a decision procedure DP_T for the case of a conjunction of equalities:

Example 3.1. In the case where T is the theory of equality, a simple procedure DP_T is easy to design. The T-literals are either equalities or inequality predicates over some set of variables V. Given a conjunction of T-literals φ, build an undirected graph $G(N, E_=, E_{\neq})$ where the nodes N correspond to the variables V, and there are two kinds of edges, $E_=$ and $E_{\neq}$, corresponding respectively to the equality and inequality predicates in φ. This is called an *equality graph*. It is not hard to see that the formula φ is unsatisfiable if and only if there exists an edge $(v_1, v_2) \in E_{\neq}$ such that v_2 is reachable from v_1 through a sequence of $E_=$ edges. The equality graph in Fig. 3.1, for example, corresponds to $x_1 \neq x_2 \wedge x_2 = x_3 \wedge x_1 = x_3$. This procedure can be implemented with $|E_{\neq}|$ depth-first search (DFS) calls over G, and is hence polynomial in the size of the input formula. More efficient procedures exist, and will be discussed in Chap. 4. ▟

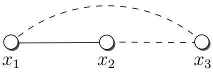


Fig. 3.1. An equality graph corresponding to $x_1 \neq x_2 \wedge x_2 = x_3 \wedge x_1 = x_3$

To reason about formulas with arbitrary propositional structure rather than just conjunctions, we can simply perform case-splitting (see Sect. 1.3), and decide each case with DP_T. If any of the cases is satisfiable, then so is the original formula. For example, there are four cases to consider in order to decide (3.1):

[2] Extending the framework for solving quantified formulas (which is not necessarily decidable) will be discussed later on in the book, namely in Sect. 9.5.

$$\begin{array}{l}(x_1 = x_2 \wedge x_1 = x_2 \wedge x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4)\ ,\\ (x_1 = x_2 \wedge x_1 = x_4 \wedge x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4)\ ,\\ (x_1 = x_3 \wedge x_1 = x_2 \wedge x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4)\ ,\\ (x_1 = x_3 \wedge x_1 = x_4 \wedge x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4)\ ,\end{array} \tag{3.4}$$

all of which are unsatisfiable. Hence, we can conclude that (3.1) is unsatisfiable.

The primary problem with this approach is the fact that the number of cases is in general exponential in the size of the original formula—think how many cases we would have if the original formula had n clauses—and indeed, when attempting to solve formulas that have a nontrivial propositional structure with this method, the number of cases is typically a major bottleneck (see Example 1.18). Furthermore, this method misses any opportunity for learning, as each case is solved independently. In the example above, the contradiction $x_1 = x_2 \wedge x_1 \neq x_2$ appears in two separate cases, but we still have to infer inconsistency for each one of them separately.

A better approach is to leverage the learning capabilities of SAT (see p. 35) and other means of efficiency, and combine it with DP_T in order to solve such formulas. The two main engines in this framework work in tight collaboration: the SAT solver chooses those literals that need to be satisfied in order to satisfy the Boolean structure of the formula, and DP_T checks whether this choice is T-satisfiable.

The advantage of this approach is that any reasoning about the propositional part of φ is performed by the propositional SAT solver; any explicit case splitting of disjunctions in φ is avoided. This has strong practical advantages, as the resulting algorithms are both very modular and very efficient.

We will now present in detail how this combination can be implemented, while continuing to use the theory of equality as an example.

3.2 An Overview of DPLL(T)

Recall that every theory T is defined with respect to a signature Σ, which is the set of allowed symbols. In the case of the theory of equality, for example, $\Sigma = \{\text{‘='}\}$. When we write Σ-literals (or, similarly, Σ-atoms and Σ-formulas), it means that the literal only uses symbols from Σ.

Let $at(\varphi)$ denote the set of Σ-atoms in a given NNF formula φ. Assuming some predefined order on the atoms, we denote the i-th distinct atom in φ by $at_i(\varphi)$.

$\boxed{at(\varphi)}$

$\boxed{at_i(\varphi)}$

Given atom a, we associate with it a unique Boolean variable $e(a)$, which we call the Boolean **encoder** of this atom. Extending this idea to formulas, given a Σ-formula t, $e(t)$ denotes the Boolean formula resulting from substituting each Σ-atom in t with its Boolean encoder.

$\boxed{e(a)}$

$\boxed{e(t)}$

For example, if $x = y$ is a Σ-atom, then $e(x = y)$, a Boolean variable, denotes its encoder. And if

$$\varphi := \; x = y \vee x = z \tag{3.5}$$

is a Σ-formula, then

$$e(\varphi) := \; e(x = y) \vee e(x = z) \; . \tag{3.6}$$

For a Σ-formula φ, the resulting Boolean formula $e(\varphi)$ is called the **propositional skeleton** of φ.

Using this notation, we can now begin to give an overview of the method studied in this chapter, while following a simple example. Some of the notation that we shall use in this example will be defined more formally later on.

As before we will use the theory of equality for the example. Let

$$\varphi := \; x = y \wedge ((y = z \wedge \neg(x = z)) \vee x = z) \; . \tag{3.7}$$

The propositional skeleton of φ is

$$e(\varphi) := \; e(x = y) \wedge ((e(y = z) \wedge \neg e(x = z)) \vee e(x = z)) \; . \tag{3.8}$$

Let $\mathcal{B}$ be a Boolean formula, initially set to $e(\varphi)$, i.e.,

$$\mathcal{B} := \; e(\varphi) \; . \tag{3.9}$$

As the next step, we pass $\mathcal{B}$ to a SAT solver. Assume that the formula is satisfiable and that the SAT solver returns the satisfying assignment

$$\alpha := \; \{e(x = y) \mapsto \text{TRUE}, \; e(y = z) \mapsto \text{TRUE}, \; e(x = z) \mapsto \text{FALSE}\} \; .$$

The decision procedure DP_T now has to decide whether the conjunction of the literals corresponding to this assignment is satisfiable. We denote this conjunction by $\hat{Th}(\alpha)$ (the "*Th*" is intended to remind the reader that the result of this function is a *Th*eory, and the "hat" that it is a conjunction of symbols). For the assignment above,

$$\hat{Th}(\alpha) := \; x = y \wedge y = z \wedge \neg(x = z) \; . \tag{3.10}$$

This formula is not satisfiable, which means that the negation of this formula is a tautology. Thus $\mathcal{B}$ is conjoined with $e(\neg\hat{Th}(\alpha))$, the Boolean encoding of this tautology:

$$e(\neg\hat{Th}(\alpha)) := \; (\neg e(x = y) \vee \neg e(y = z) \vee e(x = z)) \; . \tag{3.11}$$

This clause contradicts the current assignment, and hence *blocks* it from being repeated. Such clauses are called **blocking clauses**. In general, we denote by t the formula—also called the **lemma**—returned by DP_T. In this example, $t := \neg\hat{Th}(\alpha)$, that is, the lemma is the negation of the full assignment α and hence it is a clause, but generally t can be multiple clauses, depending on the implementation of DP_T.

After the blocking clause has been added, the SAT solver is invoked again and suggests another assignment, for example,

$$\alpha' := \ \{e(x = y) \mapsto \text{TRUE},\ e(y = z) \mapsto \text{TRUE},\ e(x = z) \mapsto \text{TRUE}\} \ .$$

The corresponding Σ-formula

$$\hat{Th}(\alpha') := \ x = y \wedge y = z \wedge x = z \tag{3.12}$$

is satisfiable, which proves that φ, the original formula, is satisfiable. Indeed, any assignment that satisfies $\hat{Th}(\alpha')$ also satisfies φ.

Figure 3.2 illustrates the information flow between the two components of the decision procedure.

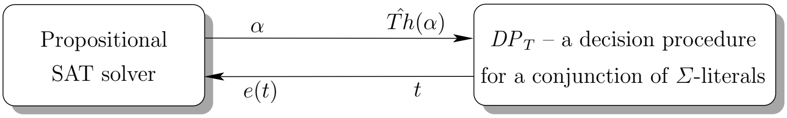

Propositional SAT solver
α
$\hat{Th}(\alpha)$
DP_T – a decision procedure for a conjunction of Σ-literals
$e(t)$
t

Fig. 3.2. The information exchanged between the SAT solver and a decision procedure DP_T for a conjunction of Σ-literals

There are many improvements to this basic procedure, some of which we shall cover later in this chapter, and some of which are left as exercises in Sect. 3.5. One such improvement, for example, is to invoke the decision procedure DP_T after some or all *partial assignments*, rather than waiting for a full assignment. A contradicting partial assignment leads to a more powerful lemma t, as it blocks all assignments that extend it. Further, when the partial assignment is not contradictory, it can be used to derive implications that are propagated back to the SAT solver. Continuing the example above, consider the partial assignment

$$\alpha := \ \{e(x = y) \mapsto \text{TRUE},\ e(y = z) \mapsto \text{TRUE}\} \ , \tag{3.13}$$

and the corresponding formula that is transferred to DP_T,

$$\hat{Th}(\alpha) := \ x = y \wedge y = z \ . \tag{3.14}$$

This leads DP_T to conclude that $x = z$ is implied, and hence accordingly to inform the SAT solver that $e(x = z) \mapsto \text{TRUE}$ is implied by the current partial assignment α. Thus, in addition to the normal Boolean constraint propagation (BCP) performed by the SAT solver, there is now also **theory propagation**. Such propagation may lead to further BCP, which means that this process may iterate several times before the next decision is made by the SAT solver.

In the next few sections, we describe variations of the process demonstrated above.

3.3 Formalization

$\boxed{\alpha}$ For a given encoding $e(\varphi)$, we denote by α an assignment, either full or partial, to the encoders in $e(\varphi)$. Then for an encoder $e(at_i)$ that is assigned a truth value by α, we define the corresponding literal, denoted $Th(at_i, \alpha)$, as follows:

$$Th(at_i, \alpha) \doteq \begin{cases} at_i & \alpha(at_i) = \text{TRUE} \\ \neg at_i & \alpha(at_i) = \text{FALSE} \ . \end{cases} \tag{3.15}$$

$\boxed{Th(\alpha)}$ Somewhat overloading the notation, we write $Th(\alpha)$ to denote the set of literals such that their encoders are assigned by α:

$$Th(\alpha) \doteq \ \{Th(at_i, \alpha) \mid e(at_i) \text{ is assigned by } \alpha\} \ . \tag{3.16}$$

$\boxed{\hat{Th}(\alpha)}$ We denote by $\hat{Th}(\alpha)$ the conjunction of the elements of the set $Th(\alpha)$.

Example 3.2. To demonstrate the use of the above notation, let

$$at_1 = (x = y), \ \ at_2 = (y = z), \ \ at_3 = (z = w) \ , \tag{3.17}$$

and let α be a partial assignment such that

$$\alpha := \ \{e(at_1) \mapsto \text{FALSE}, \ e(at_2) \mapsto \text{TRUE}\} \ . \tag{3.18}$$

Then

$$Th(at_1, \alpha) := \ \neg(x = y), \ Th(at_2, \alpha) := \ (y = z) \ , \tag{3.19}$$

and

$$Th(\alpha) := \ \{\neg(x = y), (y = z)\} \ . \tag{3.20}$$

Conjoining these terms gives us

$$\hat{Th}(\alpha) := \ \neg(x = y) \wedge (y = z) \ . \tag{3.21}$$

▟

Recall that DP_T is a decision procedure for a conjunction of T-literals, where T is a theory defined over the symbols in Σ. Let DEDUCTION be a procedure based on DP_T, which receives a conjunction of T-literals as input, decides whether it is satisfiable, and, if the answer is negative, returns constraints over these literals, as explained below. On the basis of such a procedure, we now examine variations of the method that is demonstrated in the introduction to this chapter.

$\boxed{\mathcal{B}}$ Algorithm 3.3.1 (LAZY-BASIC) decides whether a given T-formula φ is satisfiable. It does so by iteratively solving a propositional formula $\mathcal{B}$, starting from $\mathcal{B} = e(\varphi)$, and gradually strengthening it with encodings of constraints that are computed by DEDUCTION.

In each iteration, SAT-SOLVER returns a tuple ⟨assignment, result⟩ in line 4. If $\mathcal{B}$ is unsatisfiable, then so is φ: LAZY-BASIC returns “Unsatisfiable” in line 5. Otherwise, in line 7 DEDUCTION checks whether $\hat{Th}(\alpha)$ is satisfiable. It returns a tuple of the form $\langle$constraint, result$\rangle$ where the constraint is a clause over Σ-literals, and the result is one of {"Satisfiable", "Unsatisfiable"}. If it is "Satisfiable", LAZY-BASIC returns "Satisfiable" in line 8 (recall that α is a full assignment). Otherwise, the formula t returned by DEDUCTION (typically one or more clauses) corresponds to a lemma about φ. In line 9, LAZY-BASIC continues by conjoining the $e(t)$ with $\mathcal{B}$ and reiterating. $\boxed{t}$

Algorithm 3.3.1: LAZY-BASIC

Input: A formula φ
Output: "Satisfiable" if φ is satisfiable, and "Unsatisfiable" otherwise

```
1. function LAZY-BASIC($\varphi$)
2.     $\mathcal{B} := e(\varphi)$;
3.     while (TRUE) do
4.         $\langle\alpha, res\rangle :=$ SAT-SOLVER($\mathcal{B}$);
5.         if $res =$"Unsatisfiable" then return "Unsatisfiable";
6.         else
7.             $\langle t, res\rangle :=$ DEDUCTION($\hat{Th}(\alpha)$);
8.             if $res =$"Satisfiable" then return "Satisfiable";
9.             $\mathcal{B} := \mathcal{B} \wedge e(t)$;
```

Consider the following three requirements on the formula t that is returned by DEDUCTION:

1. The formula t is T-valid, i.e., t is a tautology in T. For example, if T is the theory of equality, then $x = y \wedge y = z \implies x = z$ is T-valid.
2. The atoms in t are restricted to those appearing in φ.
3. The encoding of t contradicts α, i.e., $e(t)$ is a blocking clause.

The first requirement is sufficient for guaranteeing soundness. The second and third requirements are sufficient for guaranteeing termination. Specifically, the third requirement guarantees that α is not repeated.

Two of the three requirements above can be weakened, however:

- Requirement 1: t can be any formula that is implied by φ, and not just a T-valid formula. However, finding formulas that on the one hand are implied by φ and on the other hand fulfill the other two requirements may be as hard as deciding φ itself, which misses the point. In practice, the amount of effort dedicated to computing t needs to be tuned separately for each theory and decision procedure, in order to maximize the overall performance.

- Requirement 2: t may refer to atoms that do not appear in φ, as long as the number of such new atoms is finite. For example, in equality logic, we may allow t to refer to all atoms of the form $x_i = x_j$ where x_i, x_j are variables in $var(\varphi)$, even if only some of these equality predicates appear in φ.

Integration into CDCL

$\boxed{\mathcal{B}^i}$

Let $\mathcal{B}^i$ be the formula $\mathcal{B}$ in the i-th iteration of the loop in Algorithm 3.3.1. The constraint $\mathcal{B}^{i+1}$ is strictly stronger than $\mathcal{B}^i$ for all $i \geq 1$, because blocking clauses are added but not removed between iterations. It is not hard to see that this implies that any conflict clause that is learned while solving $\mathcal{B}^i$ can be reused when solving $\mathcal{B}^j$ for $i < j$. This, in fact, is a special case of **incremental satisfiability**, which is supported by most modern SAT solvers.[3] Hence, invoking an incremental SAT solver in line 4 can increase the efficiency of the algorithm.

A better option is to integrate DEDUCTION into the CDCL-SAT algorithm, as shown in Algorithm 3.3.2. This algorithm uses a procedure ADDCLAUSES, which adds new clauses to the current set of clauses at run time. We leave the question of why this is a better option than using an incremental SAT solver to the reader (see Problem 3.1). We note that α, which is referred to in line 9, is the current assignment to the propositional variables.

3.4 Theory Propagation and the DPLL(T) Framework

3.4.1 Propagating Theory Implications

Algorithm 3.3.2 can be optimized further. Consider, for example, a formula φ that contains an integer variable x_1 and, among others, the literals $x_1 \geq 10$ and $x_1 < 0$.

Assume that the DECIDE procedure assigns $e(x_1 \geq 10) \mapsto$ TRUE and $e(x_1 < 0) \mapsto$ TRUE. Inevitably, any call to DEDUCTION results in a contradiction between these two facts, independently of any other decisions that are made. However, Algorithm 3.3.2 does not call DEDUCTION until a full satisfying assignment is found. Thus, the time taken to complete the assignment is wasted. Moreover, as was mentioned in the introduction to this chapter, the refutation of this full assignment may be due to other reasons (i.e., a proof that a different subset of the assignment is contradictory), and, hence, additional assignments that include the same wrong assignment to $e(x_1 \geq 10)$ and $e(x_1 < 0)$ are not ruled out.

[3] Incremental satisfiability was described in Sect. 2.2.7. It is concerned with the more general case in which clauses can also be *removed*. The question in that case is which conflict clauses can be reused safely. See also Problem 2.16.

Algorithm 3.3.2: LAZY-CDCL

Input: A formula φ
Output: "Satisfiable" if the formula is satisfiable, and "Unsatisfiable" otherwise

```
 1. function LAZY-CDCL
 2.     ADDCLAUSES($cnf(e(\varphi))$);
 3.     while (TRUE) do
 4.         while (BCP() = "conflict") do
 5.             backtrack-level := ANALYZE-CONFLICT();
 6.             if backtrack-level < 0 then return "Unsatisfiable";
 7.             else BackTrack(backtrack-level);
 8.         if ¬DECIDE() then                              ▷ Full assignment
 9.             $\langle t, res\rangle$:=DEDUCTION($\hat{Th}(\alpha)$);       ▷ $\alpha$ is the assignment
10.             if res="Satisfiable" then return "Satisfiable";
11.             ADDCLAUSES($e(t)$);
```

Algorithm 3.3.2 can therefore be improved by running DEDUCTION even before a full assignment to the encoders is available. This early call to DEDUCTION can serve two purposes:

1. Contradictory partial assignments are ruled out early.
2. Implications of literals that are still unassigned can be communicated back to the SAT solver, as demonstrated in Sect. 3.2. Continuing our example, once $e(x_1 \geq 10)$ has been assigned TRUE, we can infer that $e(x_1 < 0)$ must be FALSE and avoid the conflict altogether.

This brings us to the next version of the algorithm, called DPLL(T), which was first introduced in an abstract form by Tinelli [274]. As in Algorithms 3.3.1 and 3.3.2, the components of the algorithm are those of CDCL and a decision procedure for a conjunctive fragment of a theory T. The name DPLL(T) (as mentioned above, it can also be called CDCL(T)) emphasizes that this is a framework that can be instantiated with different theories and corresponding decision procedures.

In the version of DPLL(T) presented in Algorithm 3.4.1 (see also Fig. 3.3), DEDUCTION is invoked in line 9 after no further implications can be made by BCP. It then finds T-implied literals and communicates them to the CDCL part of the solver in the form of a constraint t.[4] Hence, in addition to implications in the Boolean domain, there are now also implications due to the

[4] DEDUCTION also returns the result *res* (whether $\hat{Th}(\alpha)$ is satisfiable), but *res* is not used. We have kept it in the pseudocode in order for the algorithm to stay similar to the previous algorithms.

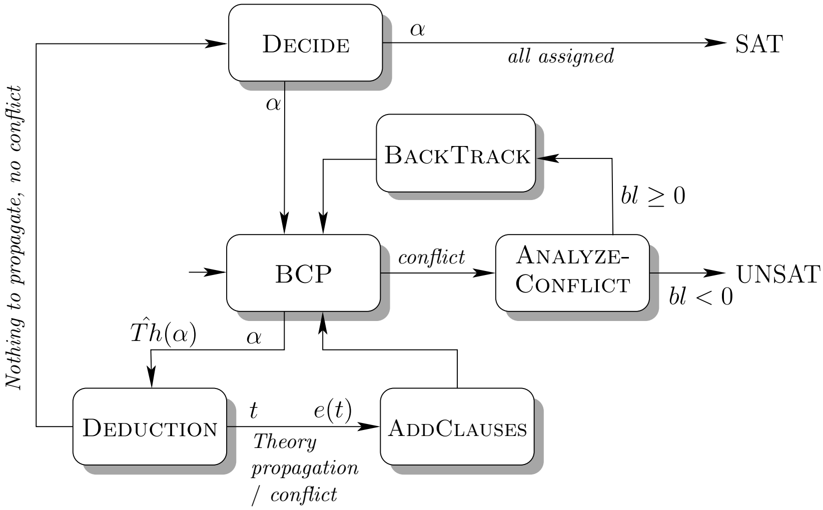


Fig. 3.3. The main components of DPLL(T). Theory propagation is implemented in Deduction

theory T. Accordingly, this technique is known by the name **theory propagation**.

What are the requirements on these new clauses? As before, they have to be implied by φ and are restricted to a finite set of atoms—typically to φ's atoms. It is desirable that, when $\hat{Th}(\alpha)$ is unsatisfiable, $e(t)$ blocks α; it is not mandatory, because whether it blocks α or not does not affect correctness—Deduction only needs to be complete when α is a full assignment. Certain SMT solvers exploit this fact to perform cheap checks on partial assignments, e.g., bound the time dedicated to them. What if $\hat{Th}(\alpha)$ is satisfiable? Then we require t to fulfill one of the following two conditions in order to guarantee termination:

1. The clause $e(t)$ is an asserting clause under α (asserting clauses are defined in Sect. 2.2.3). This implies that the addition of $e(t)$ to $\mathcal{B}$ and a call to BCP leads to an assignment to the encoder of some literal.
2. When Deduction cannot find an asserting clause t as defined above, t and $e(t)$ are equivalent to TRUE.

The second case occurs, for example, when all the Boolean variables are already assigned, and thus the formula is found to be satisfiable. In this case, the condition in line 11 is met and the procedure continues from line 13, where Decide is called again. Since all variables are already assigned, the procedure returns "Satisfiable".

Example 3.3. Consider once again the example of the two encoders $e(x_1 \geq 10)$ and $e(x_1 < 0)$. After the first of these has been set to TRUE, the procedure

DEDUCTION detects that $\neg(x_1 < 0)$ is implied, or, in other words, that

$$t := \ \neg(x_1 \geq 10) \vee \neg(x_1 < 0) \tag{3.22}$$

is T-valid. The corresponding encoded (asserting) clause

$$e(t) := \ (\neg e(x_1 \geq 10) \vee \neg e(x_1 < 0)) \tag{3.23}$$

is added to $\mathcal{B}$, which leads to an immediate implication of $\neg e(x_1 < 0)$, and possibly further implications. ▟

Algorithm 3.4.1: DPLL(T)

Input: A formula φ
Output: "Satisfiable" if the formula is satisfiable, and "Unsatisfiable" otherwise

```
function DPLL(T)
    AddClauses($cnf(e(\varphi))$);
    while (TRUE) do
        repeat
            while (BCP() = "conflict") do
                backtrack-level := Analyze-Conflict();
                if backtrack-level < 0 then return "Unsatisfiable";
                else BackTrack(backtrack-level);
            $\langle t, res\rangle$:=Deduction($\hat{Th}(\alpha)$);
            AddClauses($e(t)$);
        until $t \equiv$ TRUE;
        if $\alpha$ is a full assignment then return "Satisfiable";
        Decide();
```

3.4.2 Performance, Performance...

Recall that, when α is partial, DEDUCTION checks if there is a contradiction on the theory side, and if not, it performs theory propagation.

For performance, it is frequently better to run an approximation in this step for finding contradictions. Indeed, as long as α is partial, there is no need for a *complete* procedure for deciding satisfiability. This is not changing the completeness of the overall algorithm, since a complete check *is* performed when α is full. A good example of this is what competitive solvers do when the theory is *integer linear arithmetic* (to be covered in Sect. 5.3). Deciding the conjunctive fragment of this theory is NP-complete, and therefore they only consider the real relaxation of the problem (this means that they refer to the variables as being in $\mathbb{R}$ rather than in $\mathbb{Z}$—reals instead of integers), which can be solved in polynomial time. This means that DEDUCTION will occasionally miss a contradiction and hence not return a blocking clause.

Another performance consideration is related to theory propagation. It is important to note that theory propagation is required not for *correctness*, but only for *efficiency*. Hence, the amount of effort invested in computing new implications needs to be well tuned in order to achieve the best overall performance.

The term **exhaustive theory propagation** refers to a procedure that finds and propagates *all* literals that are implied in T by $\hat{Th}(\alpha)$. A simple, generic way (called "plunging") to perform theory propagation is the following: Given an unassigned theory atom at_i, check whether $\hat{Th}(\alpha)$ implies either at_i or $\neg at_i$. The set of unassigned atoms that are checked in this way depends on how exhaustive we want the theory propagation to be.

Example 3.4. Consider equality logic, and the notation we used in Example 3.1. A simple way to perform exhaustive theory propagation in equality logic is the following: For each unassigned atom of the form $x_i = x_j$, check if the current partial assignment forms a path in $E_=$ between x_i and x_j. If yes, then this atom is implied by the literals in the path. If the partial assignment forms a disequality path (a path in which exactly one edge is from $E_{\neq}$), the negation of this atom is implied.

This generic method is typically not the most efficient, however. In many cases a better strategy is to perform only simple, inexpensive propagations, while ignoring more expensive ones. In the case of linear arithmetic, for example, experiments have shown that exhaustive theory propagation has a negative effect on overall performance. It is more efficient in this case to search for simple-to-find implications, such as "if $x > c$ holds, where x is a variable and c a constant, then any literal of the form $x > d$ is implied if $d < c$".

3.4.3 Returning Implied Assignments Instead of Clauses

Another optimization of theory propagation is concerned with the way in which the information discovered by DEDUCTION is propagated to the Boolean part of the solver. So far, we have required that the clause t returned by DEDUCTION be T-valid. For example, if α is such that $\hat{Th}(\alpha)$ implies a literal *lit*, then

$$t := \ (lit \vee \neg \hat{Th}(\alpha))\ . \tag{3.24}$$

The encoded clause $e(t)$ is of the form

$$\Big(e(lit) \vee \bigvee_{lit' \in Th(\alpha)} \neg e(lit')\Big)\ . \tag{3.25}$$

Nieuwenhuis, Oliveras, and Tinelli concluded that this was an inefficient method, however [211]. Their experiments on various sets of benchmarks showed that on average fewer than 0.5% of these clauses were ever used again, and that they burden the process. They suggested a better alternative, in which DEDUCTION returns a list of implied assignments (containing $e(lit)$ in this case) that the SAT solver then performs.

These implied assignments have no antecedent clauses in $\mathcal{B}$, in contrast to the standard implications due to BCP. This causes a problem in ANALYZE-CONFLICT (see Algorithm 2.2.2), which relies on antecedent clauses for deriving **conflict clauses**. As a solution, when ANALYZE-CONFLICT needs an antecedent for such an implied literal, it queries the decision procedure for an **explanation**, i.e., a clause implied by φ that implies this literal given the partial assignment at the time the assignment was created.

The explanation of an assignment might be the same clause that could have been delivered in the first place, but not necessarily: for efficiency reasons, typical implementations of DEDUCTION do not retain such clauses, and hence need to generate a new explanation. As an example, to explain an implied literal $x = y$ in equality logic, one needs to search for an equality path in the equality graph between x and y, in which all the edges were present in the graph at the time that this implication was identified and propagated.

3.4.4 Generating Strong Lemmas

If $\hat{Th}(\alpha)$ is unsatisfiable, DEDUCTION returns a blocking clause t to eliminate the assignment α. The stronger t is, the greater the number of inconsistent assignments it eliminates. One way of obtaining a stronger formula is to construct a clause consisting of the negation of those literals that participate in the proof of unsatisfiability of $\hat{Th}(\alpha)$. In other words, if S is the set of literals that serve as the premises in the proof of unsatisfiability, then the blocking clause is

$$t := \big(\bigvee_{l \in S} \neg l\big)\ . \tag{3.26}$$

Computing the set S corresponds to computing an *unsatisfiable core* of the formula.[5] Given a deductive proof of unsatisfiability, a core is easy to find. For this purpose, one may represent such a proof as a directed acyclic graph, as demonstrated in Fig. 3.4 (in this case for T being equality logic and uninterpreted functions). In this graph the nodes are labeled with literals and an edge (n_1, n_2) denotes the fact that the literal labeling node n_1 was used in the inference of the literal labeling node n_2. In such a graph, there is a single sink node labeled with FALSE, and the roots are labeled with the premises (and possibly axioms) of the proof. The set of roots that can be reached by a backward traversal from the FALSE node correspond to an unsatisfiable core.

[5] *Unsatisfiable cores* are defined for the case of propositional CNF formulas in Sect. 2.2.6. The brief discussion here generalizes this earlier definition to inference rules other than BINARY RESOLUTION.

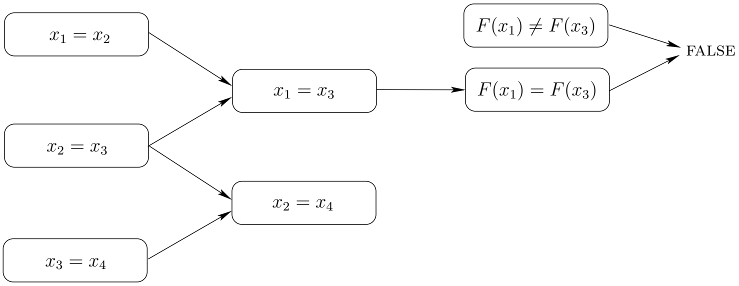


Fig. 3.4. The premises of a proof of unsatisfiability correspond to roots in the graph that can be reached by backward traversal from the FALSE node (in this case all roots other than $x_3 = x_4$). Whereas lemmas correspond to all roots, this subset of the roots can be used for generating *strong* lemmas

3.4.5 Immediate Propagation

Now consider a variation of this algorithm that calls Deduction after every new assignment to an encoding variable—which may be due to either a decision or a BCP implication—rather than letting BCP finish first. Furthermore, assume that we are implementing exhaustive theory propagation as described above. In this variant, a call to Deduction *cannot lead to a conflict*, which means that it never has to return a blocking clause. A formal proof of this observation is left as an exercise (Problem 3.5). An informal justification is that, if an assignment to a single encoder makes $\hat{Th}(\alpha)$ unsatisfiable, then the negation of that assignment would have been implied and propagated in the previous step by Deduction. For example, if an encoder $e(x = y)$ is implied and communicated to Deduction, this literal can cause a conflict only if there is a disequality path (such paths were discussed in Example 3.4) between x and y according to the previous partial assignment. This means that, in the previous step, $\neg e(x = y)$ should have been propagated to the Boolean part of the solver.

3.5 Problems

Problem 3.1 (incrementality in Lazy-CDCL). Recall that an incremental SAT solver is one that knows which conflict clauses can be reused when given a problem similar to the previous one (i.e., some clauses are added and others are erased). Is there a difference between Algorithm 3.3.2 (Lazy-CDCL) and replacing line 4 in Algorithm 3.3.1 with a call to an incremental SAT solver?

Problem 3.2 (an optimization for Algorithms 3.3.1–3.4.1?).

1. Consider the following variation of Algorithms 3.3.1–3.4.1 for an input formula φ given in NNF (negations are pushed all the way into the

atoms, e.g., $\neg(x = y)$ appears as $x \neq y$). Rather than sending $\hat{Th}(\alpha)$ to Deduction, send $\bigwedge Th_i$ for all i such that $\alpha(e_i) = \text{TRUE}$. For example, given an assignment

$$\alpha := \{e(x = y) \mapsto \text{TRUE},\ e(y = z) \mapsto \text{FALSE},\ e(x = z) \mapsto \text{TRUE}\}\ , \quad (3.27)$$

check

$$x = y \wedge x = z\ . \quad (3.28)$$

Is this variation correct? Prove that it is correct or give a counterexample.

2. Show an example in which the above variation reduces the number of iterations between Deduction and the SAT solver.

Problem 3.3 (theory propagation). Let DP_T be a decision procedure for a conjunction of Σ-literals. Suggest a procedure for performing exhaustive theory propagation with DP_T.

Problem 3.4 (pseudocode for a variant of DPLL(T)). Recall the variant of DPLL(T) suggested at the end of Sect. 3.4.5, where the partial assignment is sent to the theory solver after every assignment to an encoder, rather than only after BCP. Write pseudocode for this algorithm, and a corresponding drawing in the style of Fig. 3.3.

Problem 3.5 (exhaustive theory propagation). In Sect. 3.4.5, it was claimed that with exhaustive theory propagation, conflicts cannot occur in Deduction and that, consequently, Deduction never returns blocking clauses. Prove this claim.

3.6 Bibliographic Notes

The following are some bibliographic details about the development of the lazy encoding frameworks and DPLL(T). In 1999, Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia in [4] proposed a solver based on an interplay between a SAT solver and a theory solver, in a fashion similar to the simple lazy approach introduced at the beginning of this chapter. Their solver was tailored to a single theory called disjunctive temporal constraints, which is a restricted version of difference logic. In fact, they combined lazy with eager reasoning: they used a preprocessing step that adds a large set of constraints to the propositional skeleton (constraints of the form $(\neg e_1 \vee \neg e_2)$ if a preliminary check discovers that the theory literals corresponding to these encoders contradict each other). This saves a lot of work later for the lazy-style engine. In the same year LPSAT was introduced [286], which also includes many of the features described in this chapter, including a process of learning strong lemmas.

The basic idea of integrating DPLL with a decision procedure for some (single) theory was suggested even earlier than that; the focus of these efforts are modal and description logics [5, 129, 149, 219].

The step-change in performance of SAT solving due to the CHAFF SAT solver in 2001 [202] led several groups, a year later, to (independently) propose decision procedures that leverage this progress. All of these algorithms correspond to some variation of the lazy approach described in Sect. 3.3: CVC [19, 268] by Aaron Stump, Clark Barrett, and David Dill; ICS-SAT [113] by Jean-Christophe Filliatre, Sam Owre, Harald Ruess, and Natarajan Shankar; MATHSAT [7] by Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani; DLSAT [186] by Moez Mahfoudh, Peter Niebert, Eugene Asarin, and Oded Maler; and VERIFUN [115] by Cormac Flanagan, Rajeev Joshi, Xinming Ou, and Jim Saxe. Most of these tools were built as generic engines that can be extended with different decision procedures. Since the introduction of these tools, this approach has become mainstream, and at least twenty other solvers based on the same principles have been developed and published.

DPLL(T) was originally described in abstract terms, in the form of a calculus, by Cesare Tinelli in [274]. Theory propagation had already appeared under various names in the papers by Armando et al. [4] and Audemard et al. [7] mentioned above. Efficient theory propagation tailored to the underlying theory T (T being equalities with uninterpreted functions (EUF) in that case) appeared first in a paper by Ganzinger et al. [120]. These authors also introduced the idea of propagating theory implications by maintaining a stack of such implied assignments, coupled with the ability to explain them a posteriori, rather than sending asserting clauses to the DPLL part of the solver. The idea of minimizing the lemmas (blocking clauses) can be attributed to Leonardo de Moura and Harald Ruess [93], although, as we mentioned earlier, finding small lemmas already appeared in the description of LPSAT.

Various details of how a DPLL-based SAT solver could be transformed into a DPLL(T) solver were described for the case of EUF in [120] and for difference logic in [209]. A good description of DPLL(T), starting from an abstract DPLL procedure and ending with fine details of implementation, was given in [211]. A very comprehensive survey on lazy SMT was given by Roberto Sebastiani [253]. There has been quite a lot of research on how to design T-solvers that can give *explanations*, which, as pointed out in Sect. 3.4.5, is a necessary component for efficient implementation of this framework—see, for example, [95, 210, 270].

Let us mention some SMT solvers that are, at the time of writing this (2015), leading at various categories according to the annual competition:

1. Z3 from Microsoft Research, developed by Leonardo de Moura and Nikolaj Bjørner [92]. In addition to its superior performance in many categories, it also offers a convenient application-programming interface (API) in several languages, and infrastructure for add-ons.

2. CVC-4 [17, 15], the development of which is led by Clark Barrett and Cesare Tinelli. CVC enables each theory to produce a proof that can be checked independently with an external tool.
3. YICES-2 [90], which was originally developed by Leonardo de Moura and later by Bruno Dutertre and Dejan Jovanovic.
4. MATHSAT-5 [71], by Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani.
5. BOOLECTOR [52], by Armin Biere and Robert Brummayer, which specializes in solving bit-vector formulas.

A procedure based on Binary Decision Diagrams (BDDs) [55], where the predicates label the nodes, appeared in [132] and [199]. In the context of hardware verification there have been a number of publications on multiway decision graphs [81], a generalization of BDDs to various first-order theories.

3.7 Glossary

The following symbols were used in this chapter:

Symbol	**Refers to ...**	**First used on page ...**
DP_T	A decision procedure for a conjunction of T-atoms	60
$e(a)$	The propositional encoder of a Σ-atom a	61
$\alpha(t)$	A truth assignment (either full or partial) to the variables of a formula t	61
$at(\varphi)$	The atoms of φ	61
$at_i(\varphi)$	Assuming some predefined order on the atoms, this denotes the i-th distinct atom in φ	61
α	An assignment (either full or partial) to the atoms	64
$Th(at_i, \alpha)$	See (3.15)	64
$Th(\alpha)$	See (3.16)	64
$\hat{Th}(\alpha)$	The conjunction over the elements in $Th(\alpha)$	64
$\mathcal{B}$	A Boolean formula. In this chapter, initially set to $e(\varphi)$, and then strengthened with constraints	64

continued on next page

continued from previous page

Symbol	**Refers to ...**	**First used on page ...**
t	For a Σ-theory T, t represents a Σ-formula (typically a clause) returned by DEDUCTION	65
$\mathcal{B}^i$	The formula $\mathcal{B}$ in the i-th iteration of the loop in Algorithm 3.3.1	66

4

Equality Logic and Uninterpreted Functions

4.1 Introduction

This chapter introduces the **theory of equality**, also known by the name **equality logic**. Equality logic can be thought of as propositional logic where the atoms are equalities between variables over some infinite type or between variables and constants. As an example, the formula $(y = z \vee (\neg(x = z) \wedge x = 2))$ is a well-formed equality logic formula, where $x, y, z \in \mathbb{R}$ ($\mathbb{R}$ denotes the reals). An example of a satisfying assignment is $\{x \mapsto 2,\ y \mapsto 2,\ z \mapsto 0\}$.

Definition 4.1 (equality logic). *An equality logic formula is defined by the following grammar:*

$$\begin{aligned} \mathit{formula} &: \mathit{formula} \wedge \mathit{formula} \mid \neg\mathit{formula} \mid (\mathit{formula}) \mid \mathit{atom} \\ \mathit{atom} &: \mathit{term}\ =\ \mathit{term} \\ \mathit{term} &: \mathit{identifier} \mid \mathit{constant} \end{aligned}$$

where the identifiers are variables defined over a single infinite domain such as the `Reals` *or* `Integers`.[1] *Constants are elements from the same domain as the identifiers.*

From an algorithmic perspective, we restrict our attention to the conjunctive fragment (i.e., conjunction is the only propositional operator allowed), since the more general Boolean structure is handled in the DPLL(T) framework, as introduced in the previous chapter.

4.1.1 Complexity and Expressiveness

The satisfiability problem for equality logic, as defined in Definition 4.1, is NP-complete. We leave the proof of this claim as an exercise (Problem 4.6).

[1] The restriction to a single domain (also called a single type or a single *sort*) is not essential. It is introduced for the sake of simplicity of the presentation.

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_4

The fact that both equality logic and propositional logic are NP-complete implies that they can model the same decision problems (with not more than a polynomial difference in the number of variables). Why should we study both, then?

For two main reasons: convenience of modeling, and efficiency. It is more natural and convenient to use equality logic for modeling certain problems than to use propositional logic, and vice versa. As for efficiency, the high-level structure in the input equality logic formula can potentially be used to make the decision procedure work faster. This information may be lost if the problem is modeled directly in propositional logic.

4.1.2 Boolean Variables

Frequently, equality logic formulas are mixed with Boolean variables. Nevertheless, we shall not integrate them into the definition of the theory, in order to keep the description of the algorithms simple. Boolean variables can easily be eliminated from the input formula by replacing each such variable with an equality between two new variables. But this is not a very efficient solution. As we progress in this chapter, it will be clear that it is easy to handle Boolean variables directly, with only small modifications to the various decision procedures. The same observation applies to many of the other theories that we consider in this book.

4.1.3 Removing the Constants: a Simplification

$\boxed{\varphi^{\text{E}}}$ **Theorem 4.2.** *Given an equality logic formula φ^{E}, there is an algorithm that generates an equisatisfiable formula (see Definition 1.9) $\varphi^{\text{E}\prime}$ without constants, in polynomial time.*

Algorithm 4.1.1: Remove-Constants

Input: An equality logic formula φ^{E} with constants $c_1, \ldots, c_n$
Output: An equality logic formula $\varphi^{\text{E}\prime}$ such that $\varphi^{\text{E}\prime}$ and φ^{E} are equisatisfiable and $\varphi^{\text{E}\prime}$ has no constants

```
1. $\varphi^{\text{E}\prime} := \varphi^{\text{E}}$.
2. In $\varphi^{\text{E}\prime}$, replace each constant $c_i$, $1 \le i \le n$, with a new variable $C_{c_i}$.
3. For each pair of constants $c_i, c_j$ such that $1 \le i < j \le n$, add the
   constraint $C_{c_i} \ne C_{c_j}$ to $\varphi^{\text{E}\prime}$.
```

$\boxed{C_{c_i}}$

Algorithm 4.1.1 eliminates the constants from a given formula by replacing them with new variables. Problems 4.1 and 4.2 focus on this procedure. Unless otherwise stated, we assume from here on that the input equality formulas do not have constants.

4.2 Uninterpreted Functions

Equality logic is far more useful if combined with **uninterpreted functions**. Uninterpreted functions are used for abstracting, or generalizing, theorems. Unlike other function symbols, they should not be interpreted as part of a model of a formula. In the following formula, for example, F and G are uninterpreted, whereas the binary function symbol "+" is interpreted as the usual addition function:

$$F(x) = F(G(y)) \lor x + 1 = y \; . \tag{4.1}$$

Definition 4.3 (equality logic with uninterpreted functions (EUF)). *An equality logic formula with* uninterpreted functions *and* uninterpreted predicates[2] *is defined by the following grammar:*

$$\begin{aligned} \mathit{formula} &: \mathit{formula} \land \mathit{formula} \mid \neg \mathit{formula} \mid (\mathit{formula}) \mid \mathit{atom} \\ \mathit{atom} &: \mathit{term} \;=\; \mathit{term} \;\mid \mathit{predicate\text{-}symbol}\ (\mathit{list\ of\ terms}) \\ \mathit{term} &: \mathit{identifier} \mid \mathit{function\text{-}symbol}\ (\mathit{list\ of\ terms}) \end{aligned}$$

We generally use capital letters to denote uninterpreted functions, and use the superscript "UF" to denote EUF formulas.

Aside: The Logic Perspective

To explain the meaning of uninterpreted functions from the perspective of logic, we have to go back to the notion of a **theory**, which was explained in Sect. 1.4. Recall the set of axioms (1.35), and that in this chapter we refer to the quantifier-free fragment.

Only a single additional axiom (an axiom scheme, actually) is necessary in order to extend equality logic to EUF. For each n-ary function symbol, $n > 0$,

$$\begin{aligned} &\forall t_1, \ldots, t_n, t'_1, \ldots, t'_n. \\ &\quad \textstyle\bigwedge_i t_i = t'_i \implies F(t_1, \ldots, t_n) = F(t'_1, \ldots, t'_n) \quad (\textsc{Congruence}) \; , \end{aligned} \tag{4.2}$$

where $t_1, \ldots, t_n, t'_1, \ldots, t'_n$ are new variables. A similar axiom can be defined for uninterpreted predicates.

Thus, whereas in theories where the function symbols are interpreted there are axioms to define their semantics—what we want them to mean—in a theory over uninterpreted functions, the only restriction we have over a satisfying interpretation is that imposed by functional consistency, namely the restriction imposed by the Congruence rule.

[2] From here on, we refer only to uninterpreted functions. Uninterpreted predicates are treated in a similar way.

4.2.1 How Uninterpreted Functions Are Used

Replacing functions with uninterpreted functions in a given formula is a common technique for making it easier to reason about (e.g., to prove its validity). At the same time, this process makes the formula *weaker*, which means that it can make a valid formula invalid. This observation is summarized in the following relation, where φ^{UF} is derived from a formula φ by replacing some or all of its functions with uninterpreted functions:

$\boxed{\varphi^{\text{UF}}}$

$$\models \varphi^{\text{UF}} \implies \models \varphi\,. \tag{4.3}$$

Uninterpreted functions are widely used in calculus and other branches of mathematics, but in the context of reasoning and verification, they are mainly used for simplifying proofs. Under certain conditions, uninterpreted functions let us reason about systems while ignoring the semantics of some or all functions, assuming they are not necessary for the proof. What does it mean to ignore the semantics of a function? (A formal explanation is briefly given in the aside on p. 79.) One way to look at this question is through the axioms that the function can be defined by. Ignoring the semantics of the function means that an interpretation need not satisfy these axioms in order to satisfy the formula. The only thing it needs to satisfy is an axiom stating that the uninterpreted function, like any function, is *consistent*, i.e., given the same inputs, it returns the same outputs.[3] This is the requirement of **functional consistency** (also called **functional congruence**):

> **Functional consistency**: Instances of the same function return the same value if given equal arguments.

There are many cases in which the formula of interest is valid regardless of the interpretation of a function. In these cases, uninterpreted functions simplify the proof significantly, especially when it comes to mechanical proofs with the aid of automatic theorem provers.

Assume that we have a method for checking the validity of an EUF formula. Relying on this assumption, the basic scheme for using uninterpreted functions is the following:

1. Let φ denote a formula of interest that has interpreted functions. Assume that a validity check of φ is too hard (computationally), or even impossible.

[3] Note that the term *function* here refers to the mathematical definition. 'Functions' in programming languages such as C or JAVA are not necessarily mathematical functions, e.g., they do not necessarily terminate or return a value. Assuming they do, they are functionally consistent with respect to all the data that they read and write (including, e.g., global variables, the heap, data read from the environment). If the function operates in a multi-threaded program or it has nondeterminism, e.g., because of uninitialized local variables, then the definition of consistency changes—see a discussion in [66].

2. Assign an uninterpreted function to each interpreted function in φ. Substitute each function in φ with the uninterpreted function to which it is mapped. Denote the new formula by φ^{UF}.
3. Check the validity of φ^{UF}. If it is valid, return "φ is valid" (this is justified by (4.3)). Otherwise, return "don't know".

The transformation in step 2 comes at a price, of course, as it loses information. As mentioned earlier, it causes the procedure to be incomplete, even if the original formula belongs to a decidable logic. When there exists a decision procedure for the input formula but it is too computationally hard to solve, one can design a procedure in which uninterpreted functions are gradually substituted back to their interpreted versions. We shall discuss this option further in Sect. 4.4.

4.2.2 An Example: Proving Equivalence of Programs

As a motivating example, consider the problem of proving the equivalence of the two C functions shown in Fig. 4.1. More specifically, the goal is to prove that they return the same value for every possible input "in".

```
int power3(int in)
{
  int i, out_a;
  out_a = in;
  for (i = 0; i < 2; i++)
    out_a = out_a * in;
  return out_a;
}
```

(a)

```
int power3_new(int in)
{
  int out_b;

  out_b = (in * in) * in;

  return out_b;
}
```

(b)

Fig. 4.1. Two C functions. The proof of their equivalence is simplified by replacing the multiplications ("*") in both programs with uninterpreted functions

In general, proving the equivalence of two programs is undecidable, which means that there is no sound and complete method to prove such an equivalence. In the present case, however, equivalence can be decided.[4] A key observation about these programs is that they have only bounded loops, and therefore it is possible to compute their input/output relations. The derivation of these relations from these two programs can be done as follows:

1. Remove the variable declarations and "return" statements.

[4] The undecidability of program verification and program equivalence is caused by unbounded memory usage, which does not occur in this example.

2. Unroll the **for** loop.
3. Replace the left-hand side variable in each assignment with a new auxiliary variable.
4. Wherever a variable is read (referred to in an expression), replace it with the auxiliary variable that replaced it in the last place where it was assigned.
5. Conjoin all program statements.

These operations result in the two formulas φ_a and φ_b, which are shown in Fig. 4.2.[5]

$$\begin{array}{l|c}
\begin{array}{rcll} out0\_a &=& in0_a & \wedge \\ out1\_a &=& out0\_a * in0_a & \wedge \\ out2\_a &=& out1\_a * in0_a & \end{array} & out0\_b = (in0_b{*}in0_b){*}in0_b; \\
 & \\
(\varphi_a) & (\varphi_b)
\end{array}$$

Fig. 4.2. Two formulas corresponding to the programs (a) and (b) in Fig. 4.1. The variables are defined over finite-width integers (i.e., bit vectors)

It is left to show that these two I/O relations are actually equivalent, that is, to prove the validity of

$$in0\_a = in0\_b \wedge \varphi_a \wedge \varphi_b \implies out2\_a = out0\_b\ . \tag{4.4}$$

Uninterpreted functions can help in proving the equivalence of the programs (a) and (b), following the general scheme suggested in Sect. 4.2.1. The motivation in this case is computational: deciding formulas with multiplication over, for example, 64-bit variables is notoriously hard. Replacing the multiplication symbol with uninterpreted functions can solve the problem.

$$\begin{array}{lc}
\begin{array}{rcll} out0\_a &=& in0\_a & \wedge \\ out1\_a &=& G(out0\_a, in0\_a) & \wedge \\ out2\_a &=& G(out1\_a, in0\_a) & \end{array} & out0\_b = G(G(in0\_b, in0\_b), in0\_b) \\
 & \\
(\varphi_a^{\mathrm{UF}}) & (\varphi_b^{\mathrm{UF}})
\end{array}$$

Fig. 4.3. After replacing "$*$" with the uninterpreted function G

Figure 4.3 presents φ_a^{UF} and φ_b^{UF}, which are φ_a and φ_b after the multiplication function has been replaced with a new uninterpreted function G.

[5] A generalization of this form of translation to programs with "if" branches and other constructs is known as **static single assignment** (SSA). SSA is used in most optimizing compilers and can be applied to the verification of programs with bounded loops in popular programming languages such as C [170]. See also Example 1.25.

Similarly, if we also had addition, we could replace all of its instances with another uninterpreted function, say F. Instead of validating (4.4), we can now attempt to validate

$$\varphi_a^{\mathrm{UF}} \wedge \varphi_b^{\mathrm{UF}} \implies \mathit{out2\_a} = \mathit{out0\_b}\ . \tag{4.5}$$

Let us make the example more challenging. Consider the two programs in Fig. 4.4. Now the input "in" to both programs is a pointer to a linked list, which, we assume, is in both programs a structure of the following form:

```
struct list {
    struct list *n; // pointer to next element
    int data;
};
```

Simply enforcing the inputs to be the same, as we did in (4.4), is not sufficient and is in fact meaningless since it is not the absolute addresses that affect the outputs of the two programs, it is the data *at* these addresses that matter. Hence we need to enforce the data rooted at "in" at the time of entry to the functions, which is read by the two programs, to be the same at isomorphic locations. For example, the value of in −> n −> data is read by both programs and hence should be the same on both sides. We use uninterpreted functions to enforce this condition. In this case we need two such functions which we call list_n and list_data, corresponding to the two fields in list. See the formulation in Fig. 4.5. It gets a little more complicated when the recursive data structure also gets written to—see Problem 4.7.

```
int mul3(struct list *in)
{
 int i, out_a;
 struct list *a;
 a = in;
 out_a = in -> data;
 for (i = 0; i < 2; i++) {
   a = a -> n;
   out_a= out_a * a -> data;
 }
 return out_a;
}
```

(a)

```
int mul3_new(struct list *in)
{
  int out_b;

  out_b =
    in -> data *
    in -> n -> data *
    in -> n -> n -> data;

  return out_b;
}
```

(b)

Fig. 4.4. The difference between these programs and those in Fig. 4.1 is that here the input is a pointer to a list. Since now the input is an arbitrary address, the challenge is to enforce the inputs to be the same in the verification condition

$$\begin{array}{ll}
a0\_a = in0\_a & \wedge \\
out0\_a = list\_data(in0\_a) & \wedge \\
a1\_a = list\_n(a0\_a) & \wedge \\
out1\_a = G(out0\_a, list\_data(a1\_a)) & \wedge \\
a2\_a = list\_n(a1\_a) & \wedge \\
out2\_a = G(out1\_a, list\_data(a2\_a)) &
\end{array}$$

$$(\varphi_a^{\mathrm{UF}})$$

$$\begin{array}{l}
out0\_b = G(G(list\_data(in0\_b), \\
list\_data(list\_n(in0\_b))), \\
list\_data(list\_n(list\_n(in0\_b)))))
\end{array}$$

$$(\varphi_b^{\mathrm{UF}})$$

Fig. 4.5. After replacing "$*$" with the uninterpreted function G, and the fields n and data with the uninterpreted function *list_n* and *list_data*, respectively

It is sufficient now to prove (4.4) in order to establish the equivalence of these two programs.

As a side note, we should mention that that there are alternative methods to prove the equivalence of these two programs. In this case, *substitution* is sufficient: by simply substituting $out2\_a$ by $out1\_a * in$, $out1\_a$ by $out0\_a * in$, and $out0\_a$ by in in φ_a, we can automatically prove (4.4), as we obtain syntactically equal expressions. However, there are many cases where such substitution is not efficient, as it can increase the size of the formula exponentially. It is also possible that substitution alone may be insufficient to prove equivalence. Consider, for example, the two functions power3_con and power3_con_new:

```
int power3_con
   (int in, int con) {
  int i, out_a;
  out_a = in;
  for (i = 0; i < 2; i++)
    out_a = con?out_a * in
                   :out_a;
  return out_a;
}
```

(a)

```
int power3_con_new
   (int in, int con) {
  int out_b;


  out_b = con?(in*in)*in
                   :in;
  return out_b;
}
```

(b)

After substitution, we obtain two expressions,

$$out\_a = con?\ ((con?\ in * in : in) * in) : (con?\ in * in : in) \tag{4.6}$$

and

$$out\_b = con?\ (in * in) * in : in\ , \tag{4.7}$$

corresponding to the two functions. Not only are these two expressions not syntactically equivalent, but also the first expression grows exponentially with the number of iterations.

Other examples of the use of uninterpreted functions are presented in Sect. 4.5.

4.3 Deciding a Conjunction of Equalities and Uninterpreted Functions with Congruence Closure

We now show a method for solving a conjunction of equalities and uninterpreted functions, introduced in 1978 by Shostak [258]. As is the case for most of the theories that we consider in this book, the satisfiability problem for conjunctions of predicates can be solved in polynomial time. Recall that we are solving the satisfiability problem for formulas without constants, as those can be removed with, for example, Algorithm 4.1.1.

Starting from a conjunction φ^{UF} of equalities and disequalities over variables and uninterpreted functions, Shostak's algorithm proceeds in two stages (see Algorithm 4.3.1) and is based on computing equivalence classes. The version of the algorithm that is presented here assumes that the uninterpreted functions have a single argument. The extension to the general case is left as an exercise (Problem 4.5).

Algorithm 4.3.1: CONGRUENCE-CLOSURE

Input: A conjunction φ^{UF} of equality predicates over variables and uninterpreted functions
Output: "Satisfiable" if φ^{UF} is satisfiable, and "Unsatisfiable" otherwise

1. Build congruence-closed equivalence classes.
    (a) Initially, put two terms t_1, t_2 (either variables or uninterpreted-function instances) in their own equivalence class if $(t_1 = t_2)$ is a predicate in φ^{UF}. All other variables form singleton equivalence classes.
    (b) Given two equivalence classes with a shared term, merge them. Repeat until there are no more classes to be merged.
    (c) Compute the *congruence closure*: given two terms t_i, t_j that are in the same class and that $F(t_i)$ and $F(t_j)$ are terms in φ^{UF} for some uninterpreted function F, merge the classes of $F(t_i)$ and $F(t_j)$. Repeat until there are no more such instances.
2. If there exists a disequality $t_i \neq t_j$ in φ^{UF} such that t_i and t_j are in the same equivalence class, return "Unsatisfiable". Otherwise return "Satisfiable".

Example 4.4. Consider the conjunction

$$\varphi^{\text{UF}} := \; x_1 = x_2 \wedge x_2 = x_3 \wedge x_4 = x_5 \wedge x_5 \neq x_1 \wedge F(x_1) \neq F(x_3) \; . \qquad (4.8)$$

Initially, the equivalence classes are

$$\{x_1, x_2\}, \{x_2, x_3\}, \{x_4, x_5\}, \{F(x_1)\}, \{F(x_3)\} \; . \qquad (4.9)$$

Step 1(b) of Algorithm 4.3.1 merges the first two classes:

$$\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{F(x_1)\}, \{F(x_3)\} . \tag{4.10}$$

The next step also merges the classes containing $F(x_1)$ and $F(x_3)$, because x_1 and x_3 are in the same class:

$$\{x_1, x_2, x_3\}, \{x_4, x_5\}, \{F(x_1), F(x_3)\} . \tag{4.11}$$

In step 2, we note that $F(x_1) \neq F(x_3)$ is a predicate in φ^{UF}, but that $F(x_1)$ and $F(x_3)$ are in the same class. Hence, φ^{UF} is unsatisfiable. ▟

Variants of Algorithm 4.3.1 can be implemented efficiently with a **union–find** data structure, which results in a time complexity of $O(n \log n)$ (see, for example, [210]).

We ultimately aim at solving the general case of formulas with an arbitrary Boolean structure. In the original presentation of his method, Shostak implemented support for disjunctions by means of case-splitting, which is the bottleneck in this method. For example, given the formula

$$\varphi^{\text{UF}} := \; x_1 = x_2 \lor (x_2 = x_3 \land x_4 = x_5 \land x_5 \neq x_1 \land F(x_1) \neq F(x_3)) \;, \tag{4.12}$$

he considered separately the two cases corresponding to the left and right parts of the disjunction. This can work well as long as there are not too many cases to consider.

The general problem of how to deal with propositional structure arises with all the theories that we study in this book. There are two main approaches. As discussed in Chap. 3, a highly efficient method is to combine a DPLL-based SAT solver with an algorithm for deciding a conjunction of literals from a particular theory. The former searches for a satisfying assignment to the propositional part of the formula, and the latter is used to check whether a particular propositional assignment corresponds to a satisfying assignment to the equality predicates.

An alternative approach is based on a full reduction of the formula to propositional logic, and is the subject of Chap. 11.

4.4 Functional Consistency Is Not Enough

Functional consistency is not always sufficient for proving correct statements. This is not surprising, as we clearly lose information by replacing concrete, interpreted functions with uninterpreted functions. Consider, for example, the *plus* ("+") function. Now suppose that we are given a formula containing the two function instances $x_1 + y_1$ and $x_2 + y_2$, and, owing to other parts of the formula, it holds that $x_1 = y_2$ and $y_1 = x_2$. Further, suppose that we replace "+" with a binary uninterpreted function F. Since we only compare arguments pairwise in the order in which they appear, the proof cannot rely on the fact that these two function instances are evaluated to give the same result. In other words, functional consistency alone does not capture the commutativity of the "+" function, which may be necessary for the proof. This demonstrates the fact that by using uninterpreted functions we lose completeness (see Definition 1.6).

One may add constraints that capture more information about the original function—commutativity, in the case of the example above. For the above example, we may add

$$(x_1 = y_2 \wedge x_2 = y_1) \implies F(x_1, x_2) = F(y_1, y_2) \; . \tag{4.13}$$

Such constraints can be tailored as needed, to reflect properties of the uninterpreted functions. In other words, by adding these constraints we make them **partially interpreted functions**, as we model some of their properties. For the multiplication function, for example, we can add a constraint that, if one of the arguments is equal to 0, then so is the result. Generally, the more abstract the formula is, the easier it is, computationally, to solve it. On the other hand, the more abstract the formula is, the fewer correct facts about its original version can be proven. The right abstraction level for a given formula can be found by a trial-and-error process. Such a process can even be automated with an **abstraction–refinement loop**,[6] as can be seen in Algorithm 4.4.1 (this is not so much an algorithm as a framework that needs to be concretized according to the exact problem at hand). In step 2, the algorithm returns "Valid" if the abstract formula is valid. The correctness of this step is implied by (4.3). If, on the other hand, the formula is not valid and the abstract formula φ' is identical to the original one, the algorithm returns "Not valid" in the next step. The optional step that follows (step 4) is not necessary for the soundness of the algorithm, but only for its performance. This step is worth executing only if it is easier than solving φ itself.

Plenty of room for creativity is left when one is implementing such an algorithm: Which constraints to add in step 5? When to resort to the original interpreted functions? How to implement step 4? An instance of such a procedure is described, for the case of bit-vector arithmetic, in Sect. 6.3.

4.5 Two Examples of the Use of Uninterpreted Functions

Uninterpreted functions can be used for *property-based* verification, that is, proving that a certain property holds for a given model. Occasionally it happens that properties are correct regardless of the semantics of a certain function, and functional consistency is all that is needed for the proof. In such

[6] Abstraction–refinement loops [173] are implemented in many automated formal-reasoning tools. The types of abstractions used can be very different from those presented here, but the basic elements of the iterative process are the same.

Aside: Rewriting Systems

Observations such as “a multiplication by 0 is equal to 0” can be formulated with *rewriting rules*. Such rules are the basis of *rewriting systems* [100, 151], which are used in several branches of mathematics and mathematical logic. Rewriting systems, in their basic form, define a set of terms and (possibly nondeterministic) rules for transforming them. Theorem provers that are based on rewriting systems (such as ACL2 [162]) use hundreds of such rules. Many of these rules can be used in the context of the partially interpreted functions that were studied in Sect. 4.4, as demonstrated for the “multiply by 0” rule.

Rewriting systems, as a formalism, have the same power as a Turing machine. They are frequently used for defining and implementing inference systems, for simplifying formulas by replacing subexpressions with equal but simpler subexpressions, for computing results of arithmetic expressions, and so forth. Such implementations require the design of a strategy for applying the rules, and a mechanism based on pattern matching for detecting the set of applicable rules at each step.

Algorithm 4.4.1: ABSTRACTION-REFINEMENT

Input: A formula φ in a logic L, such that there is a decision procedure for L with uninterpreted functions

Output: “Valid” if φ is valid, and “Not valid” otherwise

```
1. $\varphi' := \mathcal{T}(\varphi)$.                    ▷ $\mathcal{T}$ is an abstraction function.
2. If $\varphi'$ is valid then return “Valid”.
3. If $\varphi' = \varphi$ then return “Not valid”.
4. (Optional) Let $\alpha'$ be a counterexample to the validity of $\varphi'$. If it is possible
   to derive a counterexample $\alpha$ to the validity of $\varphi$ (possibly by extending
   $\alpha'$ to those variables in $\varphi$ that are not in $\varphi'$), return “Not valid”.
5. Refine $\varphi'$ by adding more constraints as discussed in Sect. 4.4, or by re-
   placing uninterpreted functions with their original interpreted versions
   (reaching, in the worst case, the original formula $\varphi$).
6. Return to step 2.
```

cases, replacing the function with an uninterpreted function can simplify the proof.

The more common use of uninterpreted functions, however, is for proving *equivalence* between systems. In the chip design industry, proving equivalence between two versions of a hardware circuit is a standard procedure. Another application is **translation validation**, a process of proving the semantic equivalence of the input and output of a compiler. Indeed, we end this chapter with a detailed description of these two problem domains.

In both applications, it is expected that every function on one side of the equation can be mapped to a similar function on the other side. In such cases, replacing all functions with an uninterpreted version is typically sufficient for proving equivalence.

4.5.1 Proving Equivalence of Circuits

Pipelining is a technique for improving the performance of a circuit such as a microprocessor. The computation is split into phases, called pipeline stages. This allows one to speed up the computation by making use of concurrent computation, as is done in an assembly line in a factory.

The clock frequency of a circuit is limited by the length of the longest path between latches (i.e., memory components), which is, in the case of a pipelined circuit, simply the length of the longest stage. The delay of each path is affected by the gates along that path and the delay that each one of them imposes.

Figure 4.6(a) shows a pipelined circuit. The input, denoted by *in*, is processed in the first stage. We model the combinational gates within the stages with uninterpreted functions, denoted by C, F, G, H, K, and D. For the sake of simplicity, we assume that they each impose the same delay. The circuit applies function F to the inputs *in*, and stores the result in latch L_1. This can be formalized as follows:

$$L_1 = F(in) \; . \tag{4.14}$$

The second stage computes values for L_2, L_3, and L_4:

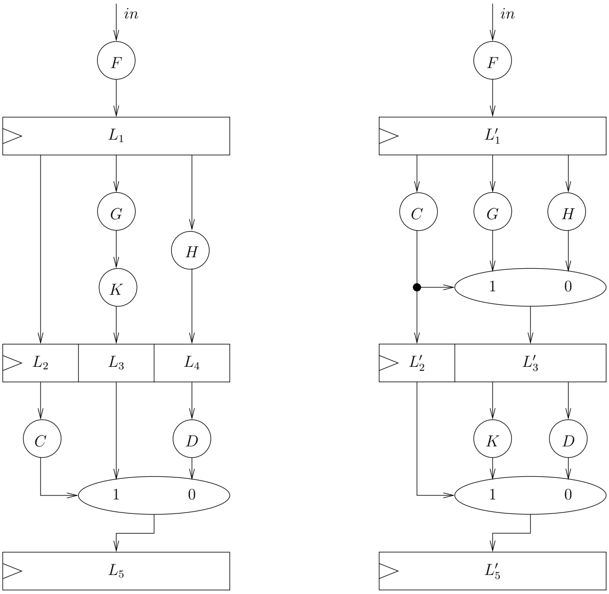
$$\begin{aligned} L_2 &= L_1 \; , \\ L_3 &= K(G(L_1)) \; , \\ L_4 &= H(L_1) \; . \end{aligned} \tag{4.15}$$

The third stage contains a *multiplexer*. A multiplexer is a circuit that selects between two inputs according to the value of a Boolean signal. In this case, this selection signal is computed by a function C. The output of the multiplexer is stored in latch L_5:

$$L_5 = C(L_2)\,?\,L_3 \;:\; D(L_4) \; . \tag{4.16}$$

Observe that the second stage contains two functions, G and K, where the output of G is used as an input for K. Suppose that this is the longest path within the circuit. We now aim to transform the circuit in order to make it work faster. This can be done in this case by moving the gates represented by K down into the third stage.

Observe also that only one of the values in L_3 and L_4 is used, as the multiplexer selects one of them depending on C. We can therefore remove one



(a) Original circuit (b) After transformation

Fig. 4.6. Showing the correctness of a transformation of a pipelined circuit using uninterpreted functions. After the transformation, the circuit has a shorter longest path between stages, and thus can be operated at a higher clock frequency

of the latches by introducing a second multiplexer in the second stage. The circuit after these changes is shown in Fig. 4.6(b). It can be formalized as follows:

$$\begin{aligned} L'_1 &= F(in)\ , \\ L'_2 &= C(L'_1)\ , \\ L'_3 &= C(L'_1)\ ?\ G(L'_1)\ :\ H(L'_1)\ , \\ L'_5 &= L'_2\ ?\ K(L'_3)\ :\ D(L'_3)\ . \end{aligned} \tag{4.17}$$

The final result of the computation is stored in L_5 in the original circuit, and in L'_5 in the modified circuit. We can show that the transformations are correct by proving that, for all inputs, the conjunction of the above equalities implies

$$L_5 = L'_5\ . \tag{4.18}$$

This proof can be automated by using a decision procedure for equalities and uninterpreted functions.

4.5.2 Verifying a Compilation Process with Translation Validation

The next example illustrates a translation validation process that relies on uninterpreted functions. Unlike the hardware example, we start from interpreted functions and replace them with uninterpreted functions.

Suppose that a source program contains the statement

$$z = (x_1 + y_1) * (x_2 + y_2) \ , \tag{4.19}$$

which the compiler that we wish to check compiles into the following sequence of three assignments:

$$u_1 = x_1 + y_1; \ u_2 = x_2 + y_2; \ z = u_1 * u_2 \ . \tag{4.20}$$

Note the two new auxiliary variables u_1 and u_2 that have been added by the compiler. To verify this translation, we construct the verification condition

$$u_1 = x_1 + y_1 \wedge u_2 = x_2 + y_2 \wedge z = u_1 * u_2 \implies z = (x_1 + y_1) * (x_2 + y_2) \ , \tag{4.21}$$

whose validity we wish to check.[7]

We now abstract the concrete functions appearing in the formula, namely addition and multiplication, by the abstract uninterpreted-function symbols F and G, respectively. The abstracted version of the implication above is

$$\begin{aligned} &(u_1 = F(x_1, y_1) \wedge u_2 = F(x_2, y_2) \wedge z = G(u_1, u_2)) \\ &\implies z = G(F(x_1, y_1), F(x_2, y_2)) \ . \end{aligned} \tag{4.22}$$

Clearly, if the abstracted version is valid, then so is the original concrete one (see (4.3)).

The success of such a process depends on how different the two sides are. Suppose that we are attempting to perform translation validation for a compiler that does not perform heavy arithmetic optimizations. In such a case, the scheme above will probably succeed. If, on the other hand, we are comparing two *arbitrary* source codes, even if they are equivalent, it is unlikely that the same scheme will be sufficient. It is possible, for example, that one side uses the function $2 * x$ while the other uses $x + x$. Since addition and multiplication are represented by two different uninterpreted functions, they are not associated with each other in any way according to the requirement of functional consistency, and hence the proof of equivalence is not able to rely on the fact that the two expressions are semantically equal.

[7] This verification condition is an implication rather than an equivalence because we are attempting to prove that the values allowed in the target code are also allowed in the source code, but not necessarily the other way. This asymmetry can be relevant when the source code is interpreted as a specification that allows multiple behaviors, only one of which is actually implemented. For the purpose of demonstrating the use of uninterpreted functions, whether we use an implication or an equivalence is immaterial.

4.6 Problems

Problem 4.1 (eliminating constants). Prove that, given an equality logic formula, Algorithm 4.1.1 returns an equisatisfiable formula without constants.

Problem 4.2 (a better way to eliminate constants?). Is the following theorem correct?

Theorem 4.5. *An equality formula φ^{E} is satisfiable if and only if the formula $\varphi^{\mathrm{E}\prime}$ generated by Algorithm 4.6.1* (Remove-Constants-Optimized) *is satisfiable.*

Prove the theorem or give a counterexample. You may use the result of Problem 4.1 in your proof.

```
Algorithm 4.6.1: Remove-Constants-Optimized

Input:  An equality logic formula $\varphi^{\mathrm{E}}$
Output: An equality logic formula $\varphi^{\mathrm{E}\prime}$ such that $\varphi^{\mathrm{E}\prime}$ contains
        no constants and $\varphi^{\mathrm{E}\prime}$ is satisfiable if and only if $\varphi^{\mathrm{E}}$ is
        satisfiable

1. $\varphi^{\mathrm{E}\prime} := \varphi^{\mathrm{E}}$.
2. Replace each constant $c$ in $\varphi^{\mathrm{E}\prime}$ with a new variable $C_c$.
3. For each pair of constants $c_i, c_j$ with an equality path between them
   ($c_i =^* c_j$) not through any other constant, add the constraint $C_{c_i} \neq$
   $C_{c_j}$ to $\varphi^{\mathrm{E}\prime}$. (Recall that the equality path is defined over $G^{\mathrm{E}}(\varphi^{\mathrm{E}})$,
   where $\varphi^{\mathrm{E}}$ is given in NNF.)
```

Problem 4.3 (deciding a conjunction of equality predicates with a graph analysis). Show a graph-based algorithm for deciding whether a given conjunction of equality predicates is satisfiable, while relying on the notion of contradictory cycles. What is the complexity of your algorithm?

Problem 4.4 (deciding a conjunction of equalities with equivalence classes).

1. Consider Algorithm 4.6.2. Present details of an efficient implementation of this algorithm, including a data structure. What is the complexity of your implementation?

```
Algorithm 4.6.2: CONJ-OF-EQUALITIES-WITH-EQUIV-CLASSES

Input:  A conjunction $\varphi^{\mathrm{E}}$ of equality predicates
Output: "Satisfiable" if $\varphi^{\mathrm{E}}$ is satisfiable, and "Unsatisfiable" otherwise

(a) Define an equivalence class for each variable. For each equality $x = y$
    in $\varphi^{\mathrm{E}}$, unite the equivalence classes of $x$ and $y$.
(b) For each disequality $u \neq v$ in $\varphi^{\mathrm{E}}$, if $u$ is in the same equivalence class
    as $v$, return "Unsatisfiable".
(c) Return "Satisfiable".
```

2. Apply your algorithm to the following formula, and determine if it is satisfiable:

$$x = f(f(f(f(f(x))))) \land x = f(f(f(x))) \land x \neq f(x)\ .$$

Problem 4.5 (a generalization of the CONGRUENCE-CLOSURE **algorithm).** Generalize Algorithm 4.3.1 to the case in which the input formula includes uninterpreted functions with multiple arguments.

Problem 4.6 (complexity of deciding equality logic). Prove that deciding equality logic is NP-complete.

Note that, to show membership in NP, it is not enough to say that every solution can be checked in P-time, because the solution itself can be arbitrarily large, and hence even reading it is not necessarily a P-time operation.

Problem 4.7 (using uninterpreted functions to encode fields of a recursive data structure). Recall the example at the second part of Sect. 4.2.2, involving pointers. The method as presented does not work if the data structure is also written to. For example, in the figure below, the code on the left results in the SSA equation on the right, which is contradictory.

```
a -> data = 1;
x = a -> data;
a -> data = 2;
x = a -> data.
```

$$\begin{array}{ll} \mathit{data}(a) = 1 & \land \\ x = \mathit{data}(a) & \land \\ \mathit{data}(a) = 2 & \land \\ x_1 = \mathit{data}(a); & \end{array}$$

Generalize the method so it also works in the presence of updates.

4.7 Bibliographic Notes

The treatment of equalities and uninterpreted functions can be divided into several eras. Solving the conjunctive fragment, for example as described in

Sect. 4.3, coupled with the DPLL(T) framework that was described in the previous chapter, is the latest of those.

In the first era, before the emergence of the first effective theorem provers in the 1970s, this logic was considered only from the point of view of mathematical logic, most notably by Ackermann [1]. In the same book, he also offered what we now call Ackermann's reduction, a procedure that we will describe in Sect. 11.2.1. Equalities were typically handled with rewriting rules, for example, substituting x with y given that $x = y$.

The second era started in the mid-1970s with the work of Downey, Sethi, and Tarjan [106], who showed that the decision problem was a variation on the common-subexpression problem; the work of Nelson and Oppen [205], who applied the union–find algorithm to compute the congruence closure and implemented it in the Stanford Pascal Verifier; and then the work of Shostak, who suggested in [258] the congruence closure method that was briefly presented in Sect. 4.3. All of this work was based on computing the congruence closure, and indicated a shift from the previous era, as it offered complete and relatively efficient methods for deciding equalities and uninterpreted functions. In its original presentation, Shostak's method relied on syntactic case-splitting (see Sect. 1.3), which is the source of the inefficiency of that algorithm. In Shostak's words, "it was found that most examples four or five lines long could be handled in just a few seconds". Even factoring in the fact that this was done on a 1978 computer (a DEC-10 computer), this statement still shows how much progress has been made since then, as nowadays many formulas with tens of thousands of variables are solved in a few seconds. Several variants on Shostak's method exist, and have been compared and described in a single theoretical framework called **abstract congruence closure** in [10]. Shostak's method and its variants are still used in theorem provers, although several improvements have been suggested to combat the practical complexity of case-splitting, namely *lazy case-splitting*, in which the formula is split only when it is necessary for the proof, and other similar techniques.

The third era will be described in Chap. 11 (see also the bibliographic notes in Sect. 11.9). It is based on the *small-model property*, namely reducing the problem to one in which only a finite set of values needs to be checked in order to determine satisfiability (this is not trivial, given that the original domain of the variables is infinite). The fourth and current era, as mentioned above, is based on solving the conjunctive fragment as part of the DPLL(T) framework.

4.8 Glossary

The following symbols were used in this chapter:

Symbol	**Refers to . . .**	**First used on page . . .**
φ^{E}	Equality formula	78
C_c	A variable used for substituting a constant c in the process of removing constants from equality formulas	78
φ^{UF}	Equality formula + uninterpreted functions	80

5

Linear Arithmetic

5.1 Introduction

This chapter introduces decision procedures for conjunctions of linear constraints. Recall that this is all that is needed for solving the more general case in which there is an arbitrary Boolean structure, based on the algorithms that were described in Chap. 3.

Definition 5.1 (linear arithmetic). *The syntax of a formula in* linear arithmetic *is defined by the following rules:*

$$\begin{aligned} \mathit{formula} &: \mathit{formula} \wedge \mathit{formula} \mid (\mathit{formula}) \mid \mathit{atom} \\ \mathit{atom} &: \mathit{sum}\ \mathit{op}\ \mathit{sum} \\ \mathit{op} &: = \ \mid\ \leq\ \mid\ < \\ \mathit{sum} &: \mathit{term} \mid \mathit{sum} + \mathit{term} \\ \mathit{term} &: \mathit{identifier} \mid \mathit{constant} \mid \mathit{constant}\ \mathit{identifier} \end{aligned}$$

The binary minus operator $a-b$ can be read as "syntactic sugar" for $a+\ -1b$. The operators $\geq$ and $>$ can be replaced by $\leq$ and $<$ if the coefficients are negated. We consider the rational numbers and the integers as domains. For the former domain the problem is polynomial, and for the latter the problem is NP-complete.

As an example, the following is a formula in linear arithmetic:

$$3x_1 + 2x_2 \leq 5x_3 \quad \wedge \quad 2x_1 - 2x_2 = 0\ . \tag{5.1}$$

Note that equality logic, as discussed in Chap. 4, is a fragment of linear arithmetic. The following example demonstrates how a compiler may use a decision procedure for arithmetic in order to optimize code.

Example 5.2. Consider the following C code fragment:

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_5

```
for(i=1; i<=10; i++)
  a[j+i]=a[j];
```

This fragment is intended to replicate the value of a[j] into the locations a[j+1] to a[j+10]. A compiler might generate the assembly code for the body of the loop as follows. Suppose variable i is stored in register R1, and variable j is stored in register R2:

```
R4 ⟵ mem[a+R2]       /* set R4 to a[j] */
R5 ⟵ R2+R1           /* set R5 to j+i */
mem[a+R5] ⟵ R4       /* set a[j+i] to a[j] */
R1 ⟵ R1+1            /* i++ */
```

Code that requires memory access is typically very slow compared with code that operates only on the internal registers of the CPU. Thus, it is highly desirable to avoid load and store instructions. A potential optimization for the code above is to move the load instruction for a[j], i.e., the first statement above, out of the loop body. After this transformation, the load instruction is executed only once at the beginning of the loop, instead of 10 times. However, the correctness of this transformation relies on the fact that the value of a[j] does not change within the loop body. We can check this condition by comparing the index of a[j+i] with the index of a[j] together with the constraint that i is between 1 and 10:

$$i \geq 1 \wedge i \leq 10 \wedge j + i = j \; . \tag{5.2}$$

This formula has no satisfying assignment, and thus, the memory accesses cannot overlap. The compiler can safely perform the read access to a[j] only once. ▟

5.1.1 Solvers for Linear Arithmetic

The **Simplex** method is one of the oldest algorithms for numerical optimization. It is used to find an optimal value for an objective function given a conjunction of linear constraints over real variables. The objective function and the constraints together are called a **linear program** (LP). However, since we are interested in the decision problem rather than the optimization problem, we cover in this chapter a variant of the Simplex method called **general Simplex** that takes as input a conjunction of linear constraints over the reals *without* an objective function, and decides whether this set is satisfiable.

Integer linear programming, or ILP, is the same problem for constraints over integers. Section 5.3 covers Branch and Bound, an algorithm for deciding such problems.

These two algorithms can solve conjunctions of a large number of constraints efficiently. We shall also describe two other methods that are considered less efficient, but can still be competitive for solving small problems.

We describe them because they are still used in practice, and they are relatively easy to implement in their basic form. The first of these methods is called **Fourier–Motzkin** variable elimination, and decides the satisfiability of a conjunction of linear constraints over the reals. The second method is called **Omega test**, and decides the satisfiability of a conjunction of linear constraints over the integers.

5.2 The Simplex Algorithm

The Simplex algorithm, originally developed by Dantzig in 1947, decides satisfiability of a conjunction of weak linear inequalities. The set of constraints is normally accompanied by a linear *objective function* in terms of the variables of the formula. If the set of constraints is satisfiable, the Simplex algorithm provides a satisfying assignment that maximizes the value of the objective function. Simplex is worst-case exponential. Although there are polynomial-time algorithms for solving this problem (the first known polynomial-time algorithm, introduced by Khachiyan in 1979, is called the **ellipsoid method**), Simplex is still considered a very efficient method in practice and the most widely used, apparently because the need for an exponential number of steps is rare in real problems.

5.2.1 A Normal Form

As we are concerned with the decision problem rather than the optimization problem, we are going to cover a variant of the Simplex algorithm called **general Simplex** that does not require an objective function. The general Simplex algorithm accepts two types of constraints as input:

1. Equalities of the form
$$a_1x_1 + \ldots + a_nx_n = 0 \; . \tag{5.3}$$
2. Lower and upper bounds on the variables:[1]
$$l_i \leq x_i \leq u_i \; , \tag{5.4}$$
where l_i and u_i are constants representing the lower and upper bounds on x_i, respectively. The bounds are optional as the algorithm supports unbounded variables. $\boxed{l_i}$ $\boxed{u_i}$

This representation of the input formula is called the **general form**. It is a normal form, which does not restrict the modeling power of weak linear constraints, as we can transform an arbitrary weak linear constraint $L \bowtie R$ with $\bowtie \in \{=, \leq, \geq\}$ into the form above as follows. Let m be the number of constraints. For the i-th constraint, $1 \leq i \leq m$: $\boxed{m}$

[1] This is in contrast to the classical Simplex algorithm, in which all variables are constrained to be nonnegative.

1. Move all addends in R to the left-hand side to obtain $L' \bowtie b$, where b is a constant.
2. Introduce a new variable s_i. Add the constraints

$$L' - s_i = 0 \quad \text{and} \quad s_i \bowtie b \; . \tag{5.5}$$

   If $\bowtie$ is the equality operator, rewrite $s_i = b$ to $s_i \geq b$ and $s_i \leq b$.

The original and the transformed conjunctions of constraints are obviously equisatisfiable.

Example 5.3. Consider the following conjunction of constraints:

$$\begin{array}{rrl} x & +y & \geq 2 \;\wedge \\ 2x & -y & \geq 0 \;\wedge \\ -x & +2y & \geq 1 \;\; . \end{array} \tag{5.6}$$

The problem is rewritten into the general form as follows:

$$\begin{array}{rrrll} x & +y & -s_1 & = 0 & \wedge \\ 2x & -y & -s_2 & = 0 & \wedge \\ -x & +2y & -s_3 & = 0 & \wedge \\ & & s_1 & \geq 2 & \wedge \\ & & s_2 & \geq 0 & \wedge \\ & & s_3 & \geq 1 \; . & \end{array} \tag{5.7}$$

▟

The new variables $s_1, \ldots, s_m$ are called the **additional variables**. The variables $x_1, \ldots, x_n$ in the original constraints are called **problem variables**. $\boxed{n}$ Thus, we have n problem variables and m additional variables. As an optimization of the procedure above, an additional variable is only introduced if L' is not already a problem variable or has been assigned an additional variable previously.

5.2.2 Basics of the Simplex Algorithm

It is common and convenient to view linear constraint satisfaction problems as geometrical problems. In geometrical terms, each variable corresponds to a dimension, and each constraint defines a convex subspace: in particular, inequalities define *half-spaces* and equalities define hyperplanes.[2] The (closed) subspace of satisfying assignments is defined by an intersection of half spaces and hyperplanes, and forms a convex polytope. This is implied by the fact that an intersection between convex subspaces is convex as well. A geometrical representation of the original problem in Example 5.3 appears in Fig. 5.1.

It is common to represent the coefficients in the input problem using an $\boxed{A}$ m-by-$(n + m)$ matrix A. The variables $x_1, \ldots, x_n, s_1, \ldots, s_m$ are written as a

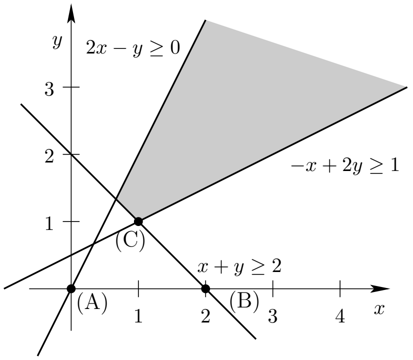


Fig. 5.1. A graphical representation of the problem in Example 5.3, projected on x and y. The shaded region corresponds to the set of satisfying assignments. The marked points (A), (B), and (C) illustrate the progress that the Simplex algorithm makes, as will be explained in the rest of this section

vector $\mathbf{x}$. Following this notation, our problem is equivalent to the existence of a vector $\mathbf{x}$ such that $\boxed{\mathbf{x}}$

$$A\mathbf{x} = 0 \quad \text{and} \quad \bigwedge_{i=1}^{m} l_i \leq s_i \leq u_i \; , \tag{5.8}$$

where $l_i \in \{-\infty\} \cup \mathbb{Q}$ is the lower bound of x_i and $u_i \in \{+\infty\} \cup \mathbb{Q}$ is the upper bound of x_i. The infinity values are for the case that a bound is not set.

Example 5.4. We continue Example 5.3. Using the variable ordering x, y, s_1, s_2, s_3, a matrix representation for the equality constraints in (5.7) is

$$\begin{pmatrix} 1 & 1 & -1 & 0 & 0 \\ 2 & -1 & 0 & -1 & 0 \\ -1 & 2 & 0 & 0 & -1 \end{pmatrix} . \tag{5.9}$$

⌟

Note that a large portion of the matrix in Example 5.4 is very regular: the columns that are added for the additional variables $s_1, \ldots, s_m$ correspond to an m-by-m diagonal matrix, where the diagonal coefficients are -1. This is a direct consequence of using the general form.

While the matrix A changes as the algorithm progresses, the number of columns of this kind is never reduced. The set of m variables corresponding

[2] A hyperplane in a d-dimensional space is a subspace with $d - 1$ dimensions. For example, in two dimensions, a hyperplane is a straight line, and in one dimension it is a point.

$\boxed{\mathcal{B}, \mathcal{N}}$

to these columns are called the **basic variables** and denoted by $\mathcal{B}$. They are also called the *dependent* variables, as their values are determined by those of the **nonbasic variables**. The nonbasic variables are denoted by $\mathcal{N}$. It is convenient to store and manipulate a representation of A called the **tableau**, which is simply A without the diagonal submatrix. The tableau is thus an m-by-n matrix, where the columns correspond to the nonbasic variables, and each row is associated with a basic variable—the same basic variable that has a "-1" entry at that row in the diagonal submatrix in A. Thus, the information originally stored in the diagonal matrix is now represented by the variables labeling the rows.

Example 5.5. We continue our running example. The tableau and the bounds for Example 5.3 are

$$
\begin{array}{c|c|c}
 & x & y \\
\hline
s_1 & 1 & 1 \\
\hline
s_2 & 2 & -1 \\
\hline
s_3 & -1 & 2
\end{array}
\qquad\qquad
\begin{array}{l}
2 \le s_1 \\
0 \le s_2 \\
1 \le s_3
\end{array}
$$

┛

The tableau is simply a different representation of A, since $A\mathbf{x} = 0$ can be rewritten into

$$\bigwedge_{x_i \in \mathcal{B}} \left(x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j \right) . \tag{5.10}$$

When written in the form of a matrix, the sums on the right-hand side of (5.10) correspond exactly to the tableau.

5.2.3 Simplex with Upper and Lower Bounds

$\boxed{\alpha}$

The general Simplex algorithm maintains, in addition to the tableau, an assignment $\alpha : \mathcal{B} \cup \mathcal{N} \longrightarrow \mathbb{Q}$. The algorithm initializes its data structures as follows:

- The set of basic variables $\mathcal{B}$ is the set of additional variables.
- The set of nonbasic variables $\mathcal{N}$ is the set of problem variables.
- For any x_i with $i \in \{1, \ldots, n+m\}$, $\alpha(x_i) = 0$.

If the initial assignment of zero to all variables (i.e., the origin) satisfies all upper and lower bounds of the basic variables, then the formula can be declared satisfiable (recall that initially the nonbasic variables do not have explicit bounds). Otherwise, the algorithm begins a process of changing this assignment.

Algorithm 5.2.1 summarizes the steps of the general Simplex procedure. The algorithm maintains two invariants:

```
Algorithm 5.2.1: General-Simplex

Input:  A linear system of constraints $S$
Output: “Satisfiable” if the system is satisfiable, and “Unsatisfiable”
        otherwise

1. Transform the system into the general form
       $A\mathbf{x} = 0$ and $\bigwedge_{i=1}^{m} l_i \leq s_i \leq u_i$ .
2. Set $\mathcal{B}$ to be the set of additional variables $s_1, \ldots, s_m$.
3. Construct the tableau for $A$.
4. Determine a fixed order on the variables.
5. If there is no basic variable that violates its bounds, return “Satisfiable”.
   Otherwise, let $x_i$ be the first basic variable in the order that violates its
   bounds.
6. Search for the first suitable nonbasic variable $x_j$ in the order for pivoting
   with $x_i$. If there is no such variable, return “Unsatisfiable”.
7. Perform the pivot operation on $x_i$ and $x_j$.
8. Go to step 5.
```

- **In-1.** $A\mathbf{x} = 0$
- **In-2.** The values of the nonbasic variables are within their bounds:

$$\forall x_j \in \mathcal{N}.\ l_j \leq \alpha(x_j) \leq u_j\ . \tag{5.11}$$

Clearly, these invariants hold initially since all the variables in $\mathbf{x}$ are set to 0, and the nonbasic variables have no bounds.

The main loop of the algorithm checks if there exists a basic variable that violates its bounds. If there is no such variable, then both the basic and nonbasic variables satisfy their bounds. Owing to invariant **In-1**, this means that the current assignment α satisfies (5.8), and the algorithm returns “Satisfiable”.

Otherwise, let x_i be a basic variable that violates its bounds, and assume, without loss of generality, that $\alpha(x_i) > u_i$, i.e., the upper bound of x_i is violated. How do we change the assignment to x_i so it satisfies its bounds? We need to find a way to reduce the value of x_i. Recall how this value is specified:

$$x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j\ . \tag{5.12}$$

The value of x_i can be reduced by decreasing the value of a nonbasic variable x_j such that $a_{ij} > 0$ and its current assignment is higher than its lower bound l_j, or by increasing the value of a variable x_j such that $a_{ij} < 0$ and its current assignment is lower than its upper bound u_j. A variable x_j fulfilling one of these conditions is said to be *suitable*. If there are no suitable variables, then the problem is unsatisfiable and the algorithm terminates.

$\boxed{\theta}$ Let θ denote by how much we have to increase (or decrease) $\alpha(x_j)$ in order to meet x_i's upper bound:

$$\theta \doteq \frac{u_i - \alpha(x_i)}{a_{ij}} \,. \tag{5.13}$$

Increasing (or decreasing) x_j by θ puts x_i within its bounds. On the other hand x_j does not necessarily satisfy its bounds anymore, and hence may violate the invariant **In-2**. We therefore swap x_i and x_j in the tableau, i.e., make x_i nonbasic and x_j basic. This requires a transformation of the tableau, which is called the **pivot operation**. The pivot operation is repeated until either a satisfying assignment is found, or the system is determined to be unsatisfiable.

The Pivot Operation

Suppose we want to swap x_i with x_j. We will need the following definition:

Definition 5.6 (pivot element, column, and row). *Given two variables x_i and x_j, the coefficient a_{ij} is called the* pivot element. *The column of x_j is called the* pivot column. *The row i is called the* pivot row.

A precondition for swapping two variables x_i and x_j is that their pivot element is nonzero, i.e., $a_{ij} \neq 0$. The pivot operation (or **pivoting**) is performed as follows:

1. Solve row i for x_j.
2. For all rows $l \neq i$, eliminate x_j by using the equality for x_j obtained from row i.

The reader may observe that the pivot operation is also the basic operation in the well-known **Gaussian variable elimination** procedure.

Example 5.7. We continue our running example. As described above, we initialize $\alpha(x_i) = 0$. This corresponds to point (A) in Fig. 5.1. Recall the tableau and the bounds:

	x	y
s_1	1	1
s_2	2	-1
s_3	-1	2

$2 \leq s_1$
$0 \leq s_2$
$1 \leq s_3$

The lower bound of s_1 is 2, which is violated. The nonbasic variable that is the lowest in the ordering is x. The variable x has a positive coefficient, but no upper bound, and is therefore suitable for the pivot operation. We need to increase s_1 by 2 in order to meet the lower bound, which means that x has to increase by 2 as well ($\theta = 2$). The first step of the pivot operation is to solve the row of s_1 for x:

$$s_1 = x + y \iff x = s_1 - y \; . \tag{5.14}$$

This equality is now used to replace x in the other two rows:

$$s_2 = 2(s_1 - y) - y \iff s_2 = 2s_1 - 3y \tag{5.15}$$

$$s_3 = -(s_1 - y) + 2y \iff s_3 = -s_1 + 3y \tag{5.16}$$

Written as a tableau, the result of the pivot operation is

	s_1	y
x	1	−1
s_2	2	−3
s_3	−1	3

$$\begin{aligned} \alpha(x) &= 2 \\ \alpha(y) &= 0 \\ \alpha(s_1) &= 2 \\ \alpha(s_2) &= 4 \\ \alpha(s_3) &= -2 \end{aligned}$$

This new state corresponds to point (B) in Fig. 5.1.

The lower bound of s_3 is violated; this is the next basic variable that is selected. The only suitable variable for pivoting is y. We need to add 3 to s_3 in order to meet the lower bound. This translates into

$$\theta = \frac{1 - (-2)}{3} = 1 \; . \tag{5.17}$$

After performing the pivot operation with s_3 and y, the final tableau is

	s_1	s_3
x	2/3	−1/3
s_2	1	−1
y	1/3	1/3

$$\begin{aligned} \alpha(x) &= 1 \\ \alpha(y) &= 1 \\ \alpha(s_1) &= 2 \\ \alpha(s_2) &= 1 \\ \alpha(s_3) &= 1 \end{aligned}$$

This assignment α satisfies the bounds, and thus $\{x \mapsto 1, y \mapsto 1\}$ is a satisfying assignment. It corresponds to point (C) in Fig. 5.1. ▟

Selecting the pivot element according to a fixed ordering for the basic and nonbasic variable ensures that no set of basic variables is ever repeated, and hence guarantees termination (no cycling can occur). For a detailed proof see [109]. This way of selecting a pivot element is called **Bland's rule**.

5.2.4 Incremental Problems

Decision problems are often constructed in an **incremental** manner, that is, the formula is strengthened with additional conjuncts. This can make a once satisfiable formula unsatisfiable. One scenario in which an incremental decision procedure is useful is the DPLL(T) framework, which we saw in Chap. 3.

The general Simplex algorithm is well suited for incremental problems. First, notice that any constraint can be disabled by removing its corresponding upper and lower bounds. The equality in the tableau is afterwards redundant, but will not render a satisfiable formula unsatisfiable. Second, the pivot operation performed on the tableau is an equivalence transformation, i.e., it preserves the set of solutions. We can therefore start the procedure with the tableau we have obtained from the previous set of bounds.

The addition of upper and lower bounds is implemented as follows:

- If a bound for a nonbasic variable was added, update the values of the nonbasic variables according to the tableau to restore **In-2**.
- Call Algorithm 5.2.1 to determine if the new problem is satisfiable. Start with step 5.

Furthermore, it is often desirable to *remove* constraints after they have been added. This is also relevant in the context of DPLL(T) because this algorithm activates and deactivates constraints. Normally constraints (or rather bounds) are removed when the current set of constraints is unsatisfiable. After removing a constraint the assignment has to be restored to a point at which it satisfied the two invariants of the general Simplex algorithm. This can be done by simply restoring the assignment α to the last known satisfying assignment. There is no need to modify the tableau.

5.3 The Branch and Bound Method

Branch and Bound is a widely used method for solving integer linear programs. As in the case of the Simplex algorithm, Branch and Bound was developed for solving the optimization problem, but the description here focuses on an adaptation of this algorithm to the decision problem.

The integer linear systems considered here have the same form as described in Sect. 5.2, with the additional requirement that the value of any variable in a satisfying assignment must be drawn from the set of integers. Observe that it is easy to support strict inequalities simply by adding 1 to or subtracting 1 from the constant on the right-hand side.

Definition 5.8 (relaxed problem). *Given an integer linear system S, its* relaxation *is S without the integrality requirement (i.e., the variables are not required to be integer).*

We denote the relaxed problem of S by relaxed(S). Assume the existence of a procedure $LP_{feasible}$, which receives a linear system S as input, and returns "Unsatisfiable" if S is unsatisfiable and a satisfying assignment otherwise. $LP_{feasible}$ can be implemented with, for example, a variation of

General-Simplex (Algorithm 5.2.1) that outputs a satisfying assignment if S is satisfiable. Using these notions, Algorithm 5.3.1 decides an integer linear system of constraints (recall that only conjunctions of constraints are considered here).

Algorithm 5.3.1: Feasibility-Branch-and-Bound

Input: An integer linear system S
Output: "Satisfiable" if S is satisfiable, and "Unsatisfiable" otherwise

```
 1. procedure SEARCH-INTEGRAL-SOLUTION($S$)
 2.     res = $LP_{feasible}$ (relaxed($S$));
 3.     if res = "Unsatisfiable" then return ;          ▷ prune branch
 4.     else
 5.         if res is integral then                      ▷ integer solution found
                abort("Satisfiable");
 6.         else
 7.             Select a variable $v$ that is assigned a nonintegral value $r$;
 8.             SEARCH-INTEGRAL-SOLUTION ($S \cup (v \le \lfloor r \rfloor)$);
 9.             SEARCH-INTEGRAL-SOLUTION ($S \cup (v \ge \lceil r \rceil)$);
10.                                         ▷ no integer solution in this branch


11. procedure FEASIBILITY-BRANCH-AND-BOUND($S$)
12.     SEARCH-INTEGRAL-SOLUTION($S$);
13.     return ("Unsatisfiable");
```

The idea of the algorithm is simple: it solves the relaxed problem with $LP_{feasible}$; if the relaxed problem is unsatisfiable, it backtracks because there is also no integer solution in this branch. If, on the other hand, the relaxed problem is satisfiable and the solution returned by $LP_{feasible}$ happens to be integral, it terminates—a satisfying integral solution has been found. Otherwise, the problem is split into two subproblems, which are then processed with a recursive call. The nature of this split is best illustrated by an example.

Example 5.9. Let $x_1, \ldots, x_4$ be the variables of S. Assume that $LP_{feasible}$ returns the solution

$$(1, 0.7, 2.5, 3) \tag{5.18}$$

in line 2. In line 7, Search-integral-solution chooses between x_2 and x_3, which are the variables that were assigned a nonintegral value. Suppose that x_2 is chosen. In line 8, S (the linear system solved at the current recursion level) is then augmented with the constraint

$$x_2 \le 0 \tag{5.19}$$

and sent for solving at a deeper recursion level. If no solution is found in this branch, S is augmented instead with

$$x_2 \geq 1 \tag{5.20}$$

and, once again, is sent to a deeper recursion level. If both these calls return, this implies that S has no satisfying solution, and hence the procedure returns (backtracks). Note that returning from the initial recursion level causes the calling function FEASIBILITY-BRANCH-AND-BOUND to return "Unsatisfiable". ▟

Algorithm 5.3.1 is not complete: there are cases for which it will branch forever. As noted in [109], the system $1 \leq 3x - 3y \leq 2$, for example, has no integer solutions but unbounded real solutions, and causes the basic branch and bound algorithm to loop forever. In order to make the algorithm complete, it is necessary to rely on the small-model property that such formulas have (we discuss this property in detail in Sect. 11.6). This means that, if there is a satisfying solution, then there is also such a solution within a finite bound, which, for this theory, is also computable. Thus, once we have computed this bound on the domain of each variable, we can stop searching for a solution once we have passed it. A detailed study of this bound in the context of optimization problems can be found in [208]. The same bounds are applicable to the feasibility problem as well. Briefly, it was shown in [208] that, given an integer linear system S with an $M \times N$ coefficient matrix A, then if there is a solution to S, then one of the extreme points of the convex hull of S is also a solution, and any such solution x^0 is bounded as follows:

$$x_j^0 \leq ((M + N) \cdot N \cdot \theta)^N \quad \text{for } j = 1, \ldots, N\ , \tag{5.21}$$

where θ is the maximal element in the problem. Thus, (5.21) gives us a bound on each of the N variables, which, by adding it as an explicit constraint, forces termination.

Finally, let us mention that BRANCH AND BOUND can be extended in a straightforward way to handle the case in which some of the variables are integers while the others are real. In the context of optimization problems, this problem is known by the name **mixed integer programming**.

5.3.1 Cutting Planes

Cutting-planes are constraints that are added to a linear system that remove only noninteger solutions; that is, all satisfying integer solutions, if they exist, remain satisfying, as demonstrated in Fig. 5.2. These new constraints improve the tightness of the relaxation in the process of solving integer linear systems, and hence can make branch and bound faster (this combination is known by the name **branch-and-cut**). Furthermore, if certain conditions are met—see Chap. 23.8 in [252] for details—Simplex + cutting planes of the type described below form a decision procedure for integer linear arithmetic.

Aside: Branch and Bound **for Integer Linear Programs**
When Branch and Bound is used for solving an optimization problem, after a first feasible solution is found, the search is continued until no smaller solution (assuming it is a minimization problem) can be found. A branch is pruned if the value of the objective according to a solution to the relaxed problem at its end is larger than the best solution found so far. The objective can also be used to guide the branching heuristic (which variable to split on next, and which side to explore first), e.g., find solutions that imply a small value of the objective function, so more future branches are pruned (bound) early.

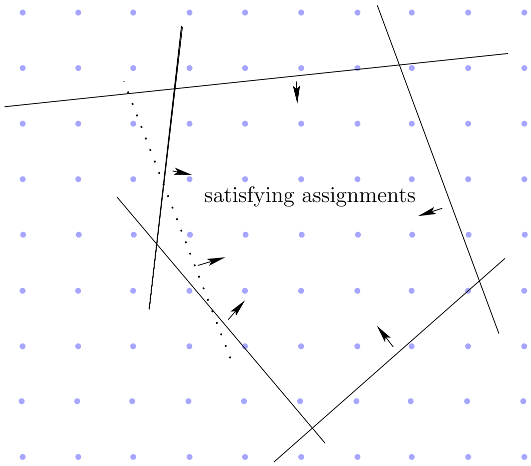


Fig. 5.2. The dots represent integer solutions. The thin dotted line represents a cutting-plane—a constraint that does not remove any integral solution

Here, we describe a family of cutting planes called **Gomory cuts**. We first illustrate this technique with an example, and then generalize it.

Suppose that our problem includes the integer variables $x_1, \ldots, x_3$, and the lower bounds $1 \leq x_1$ and $0.5 \leq x_2$. Further, suppose that the final tableau of the general Simplex algorithm includes the constraint

$$x_3 = 0.5x_1 + 2.5x_2 \ , \tag{5.22}$$

and that the solution α is $\{x_3 \mapsto 1.75, x_1 \mapsto 1, x_2 \mapsto 0.5\}$, which, of course, satisfies (5.22). Subtracting these values from (5.22) gives us

$$x_3 - 1.75 = 0.5(x_1 - 1) + 2.5(x_2 - 0.5) \ . \tag{5.23}$$

We now wish to rewrite this equation so the left-hand side is an integer:

$$x_3 - 1 = 0.75 + 0.5(x_1 - 1) + 2.5(x_2 - 0.5) \ . \tag{5.24}$$

The two right most terms must be positive because 1 and 0.5 are the lower bounds of x_1 and x_2, respectively. Since the right-hand side must add up to an integer as well, this implies that

$$0.75 + 0.5(x_1 - 1) + 2.5(x_2 - 0.5) \geq 1 \; . \tag{5.25}$$

Note, however, that this constraint is unsatisfied by α since by construction all the elements on the left other than the fraction 0.75 are equal to zero under α. This means that adding this constraint to the relaxed system will rule out this solution. On the other hand since it is implied by the integer system of constraints, it cannot remove any *integer* solution.

Let us generalize this example into a recipe for generating such cutting planes. The generalization refers also to the case of having variables assigned their upper bounds, and both negative and positive coefficients. In order to derive a Gomory cut from a constraint, the constraint has to satisfy two conditions: First, the assignment to the basic variable has to be fractional; second, the assignments to all the nonbasic variables have to correspond to one of their bounds. The following recipe, which relies on these conditions, is based on a report by Dutertre and de Moura [109].

Consider the i-th constraint:

$$x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j \; , \tag{5.26}$$

where $x_i \in \mathcal{B}$. Let α be the assignment returned by the general Simplex algorithm. Thus,

$$\alpha(x_i) = \sum_{x_j \in \mathcal{N}} a_{ij} \alpha(x_j) \; . \tag{5.27}$$

We now partition the nonbasic variables to those that are currently assigned their lower bound and those that are currently assigned their upper bound:

$$\begin{aligned} J &= \{ j \mid x_j \in \mathcal{N} \wedge \alpha(x_j) = l_j \} \; , \\ K &= \{ j \mid x_j \in \mathcal{N} \wedge \alpha(x_j) = u_j \} \; . \end{aligned} \tag{5.28}$$

Subtracting (5.27) from (5.26) taking the partition into account yields

$$x_i - \alpha(x_i) = \sum_{j \in J} a_{ij}(x_j - l_j) - \sum_{j \in K} a_{ij}(u_j - x_j) \; . \tag{5.29}$$

Let $f_0 = \alpha(x_i) - \lfloor \alpha(x_i) \rfloor$. Since we assumed that $\alpha(x_i)$ is not an integer then $0 < f_0 < 1$. We can now rewrite (5.29) as

$$x_i - \lfloor \alpha(x_i) \rfloor = f_0 + \sum_{j \in J} a_{ij}(x_j - l_j) - \sum_{j \in K} a_{ij}(u_j - x_j) \; . \tag{5.30}$$

Note that the left-hand side is an integer. We now consider two cases.

- If $\sum_{j\in J} a_{ij}(x_j - l_j) - \sum_{j\in K} a_{ij}(u_j - x_j) > 0$ then, since the right-hand side must be an integer,
$$f_0 + \sum_{j\in J} a_{ij}(x_j - l_j) - \sum_{j\in K} a_{ij}(u_j - x_j) \geq 1 \ . \tag{5.31}$$
We now split J and K as follows:
$$\begin{aligned} J^+ &= \{j \mid j \in J \wedge a_{ij} > 0\} \ , \\ J^- &= \{j \mid j \in J \wedge a_{ij} < 0\} \ , \\ K^+ &= \{j \mid j \in K \wedge a_{ij} > 0\} \ , \\ K^- &= \{j \mid j \in K \wedge a_{ij} < 0\} \ . \end{aligned} \tag{5.32}$$
Gathering only the positive elements in the left-hand side of (5.31) gives us
$$\sum_{j\in J^+} a_{ij}(x_j - l_j) - \sum_{j\in K^-} a_{ij}(u_j - x_j) \geq 1 - f_0 \ , \tag{5.33}$$
or, equivalently,
$$\sum_{j\in J^+} \frac{a_{ij}}{1-f_0}(x_j - l_j) - \sum_{j\in K^-} \frac{a_{ij}}{1-f_0}(u_j - x_j) \geq 1 \ . \tag{5.34}$$
- If $\sum_{j\in J} a_{ij}(x_j - l_j) - \sum_{j\in K} a_{ij}(u_j - x_j) \leq 0$ then again, since the right-hand side must be an integer,
$$f_0 + \sum_{j\in J} a_{ij}(x_j - l_j) - \sum_{j\in K} a_{ij}(u_j - x_j) \leq 0 \ . \tag{5.35}$$
Equation (5.35) implies that
$$\sum_{j\in J^-} a_{ij}(x_j - l_j) - \sum_{j\in K^+} a_{ij}(u_j - x_j) \leq -f_0 \ . \tag{5.36}$$
Dividing by $-f_0$ gives us
$$- \sum_{j\in J^-} \frac{a_{ij}}{f_0}(x_j - l_j) + \sum_{j\in K^+} \frac{a_{ij}}{f_0}(u_j - x_j) \geq 1 \ . \tag{5.37}$$

Note that the left-hand side of both (5.34) and (5.37) is greater than zero. Therefore these two equations imply
$$\begin{aligned} &\sum_{j\in J^+} \frac{a_{ij}}{1-f_0}(x_j - l_j) - \sum_{j\in J^-} \frac{a_{ij}}{f_0}(x_j - l_j) \\ &+ \sum_{j\in K^+} \frac{a_{ij}}{f_0}(u_j - x_j) - \sum_{j\in K^-} \frac{a_{ij}}{1-f_0}(u_j - x_j) \geq 1 \ . \end{aligned} \tag{5.38}$$

Since each of the elements on the left-hand side is equal to zero under the current assignment α, this assignment α is ruled out by the new constraint. In other words, the solution to the linear problem augmented with the constraint is guaranteed to be different from the previous one.

5.4 Fourier–Motzkin Variable Elimination

Similarly to the Simplex method (Sect. 5.2), the Fourier–Motzkin variable elimination algorithm takes a conjunction of linear constraints over real variables and decides their satisfiability. It is not as efficient as Simplex, but it can still be competitive for small formulas. In practice it is used mostly as a method for eliminating existential quantifiers, a topic that we will only cover later, in Sect. 9.2.4.

Let m denote the number of such constraints, and let $x_1, \ldots, x_n$ denote the variables used by these constraints. We begin by eliminating equalities.

5.4.1 Equality Constraints

As a first step, equality constraints of the following form are eliminated:

$$\sum_{j=1}^{n} a_{i,j} \cdot x_j = b_i \ . \tag{5.39}$$

We choose a variable x_j that has a nonzero coefficient $a_{i,j}$ in an equality constraint i. Without loss of generality, we assume that x_n is the variable that is to be eliminated. The constraint (5.39) can be rewritten as

$$x_n \ = \ \frac{b_i}{a_{i,n}} - \sum_{j=1}^{n-1} \frac{a_{i,j}}{a_{i,n}} \cdot x_j \ . \tag{5.40}$$

Now we substitute the right-hand side of (5.40) for x_n into all the other constraints, and remove constraint i. This is iterated until all equalities are removed. We are left with a system of inequalities of the form

$$\bigwedge_{i=1}^{m} \sum_{j=1}^{n} a_{i,j} x_j \leq b_i \ . \tag{5.41}$$

5.4.2 Variable Elimination

The basic idea of the variable elimination algorithm is to heuristically choose a variable and then to eliminate it by projecting its constraints onto the rest of the system, resulting in new constraints.

Example 5.10. Consider Fig. 5.3(a): the constraints

$$0 \leq x \leq 1,\ 0 \leq y \leq 1,\ \frac{3}{4} \leq z \leq 1 \tag{5.42}$$

form a cuboid. Projecting these constraints onto the x and y axes, and thereby eliminating z, results in a square which is given by the constraints

$$0 \leq x \leq 1,\ 0 \leq y \leq 1\ . \tag{5.43}$$

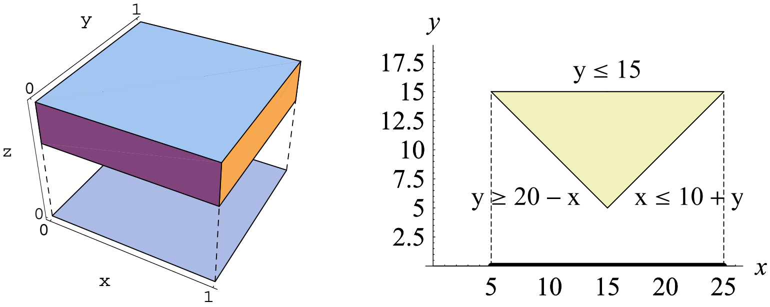
Figure 5.3(b) shows a triangle formed by the constraints

$$x \leq y + 10,\ y \leq 15,\ y \geq -x + 20\ . \tag{5.44}$$

The projection of the triangle onto the x axis is a line given by the constraints

$$5 \leq x \leq 25\ . \tag{5.45}$$

▟



(a) (b)

Fig. 5.3. Projection of constraints: (a) a cuboid is projected onto the x and y axes; (b) a triangle is projected onto the x axis

Thus, the projection forms a new problem with one variable fewer, but possibly more constraints. This is done iteratively until all variables but one have been eliminated. The problem with one variable is trivially decidable.

The order in which the variables are eliminated may be predetermined, or adjusted dynamically to the current set of constraints. There are various heuristics for choosing the elimination order. A standard greedy heuristic gives priority to variables that produce fewer new constraints when eliminated.

Once again, assume that x_n is the variable chosen to be eliminated. The constraints are partitioned according to the coefficient of x_n. Consider the constraint with index i:

$$\sum_{j=1}^{n} a_{i,j} \cdot x_j \leq b_i\ . \tag{5.46}$$

By splitting the sum, (5.46) can be rewritten into

$$a_{i,n} \cdot x_n\ \leq\ b_i - \sum_{j=1}^{n-1} a_{i,j} \cdot x_j\ . \tag{5.47}$$

If $a_{i,n}$ is zero, the constraint can be disregarded when we are eliminating x_n. Otherwise, we divide by $a_{i,n}$. If $a_{i,n}$ is positive, we obtain

$$x_n \;\leq\; \frac{b_i}{a_{i,n}} - \sum_{j=1}^{n-1} \frac{a_{i,j}}{a_{i,n}} \cdot x_j \; . \tag{5.48}$$

$\boxed{\beta_i}$

Thus, if $a_{i,n} > 0$, the constraint is an *upper bound* on x_n. If $a_{i,n} < 0$, the constraint is a *lower bound.* We denote the right-hand side of (5.48) by β_i.

Unbounded Variables

It is possible that a variable is not bounded both ways, i.e., it has either only upper bounds or only lower bounds. Such variables are called **unbounded variables**. Unbounded variables can be simply removed from the system together with all constraints that use them. Removing these constraints can make other variables unbounded. Thus, this simplification stage iterates until no such variables remain.

Bounded Variables

If x_n has both an upper and a lower bound, the algorithm enumerates all pairs of lower and upper bounds. Let $u \in \{1, \ldots, m\}$ denote the index of an upper-bound constraint, and $l \in \{1, \ldots, m\}$ denote the index of a lower-bound constraint for x_n, where $l \neq u$. For each such pair, we have

$$\beta_l \quad \leq \quad x_n \quad \leq \quad \beta_u \; . \tag{5.49}$$

The following new constraint is added:

$$\beta_l \quad \leq \quad \beta_u \; . \tag{5.50}$$

The Formula (5.50) may simplify to $0 \leq b_k$, where b_k is some constant smaller than 0. In this case, the algorithm has found a *conflicting* pair of constraints and concludes that the problem is unsatisfiable. Otherwise, all constraints that involve x_n are removed. The new problem is solved recursively as before.

Example 5.11. Consider the following set of constraints:

$$\begin{array}{rrrcr}
x_1 & -x_2 & & \leq & 0 \\
x_1 & & -x_3 & \leq & 0 \\
-x_1 & +x_2 & +2x_3 & \leq & 0 \\
 & & -x_3 & \leq & -1 \; .
\end{array} \tag{5.51}$$

Suppose we decide to eliminate the variable x_1 first. There are two upper bounds on x_1, namely $x_1 \leq x_2$ and $x_1 \leq x_3$, and one lower bound, which is $x_2 + 2x_3 \leq x_1$.

Using $x_1 \leq x_2$ as the upper bound, we obtain a new constraint $2x_3 \leq 0$, and using $x_1 \leq x_3$ as the upper bound, we obtain a new constraint $x_2 + x_3 \leq 0$. Constraints involving x_1 are removed from the problem, which results in the following new set:

$$\begin{array}{rr} 2x_3 \leq & 0 \\ x_2 + x_3 \leq & 0 \\ -x_3 \leq & -1 \ . \end{array} \tag{5.52}$$

Next, observe that x_2 is unbounded (as it has no lower bound), and hence the second constraint can be eliminated, which simplifies the formula. We therefore progress by eliminating x_2 and all the constraints that contain it:

$$\begin{array}{rr} 2x_3 \leq & 0 \\ -x_3 \leq & -1 \ . \end{array} \tag{5.53}$$

Only the variable x_3 remains, with a lower and an upper bound. Combining the two into a new constraint results in $1 \leq 0$, which is a contradiction. Thus, the system is unsatisfiable. ▟

The Simplex method in its basic form, as described in Sect. 5.2, allows only nonstrict ($\leq$) inequalities.[3] The Fourier–Motzkin method, on the other hand, can easily be extended to handle a combination of strict ($<$) and nonstrict inequalities: if either the lower or the upper bound is a strict inequality, then so is the resulting constraint.

5.4.3 Complexity

In each iteration, the number of constraints can increase in the worst case from m to $m^2/4$, which results overall in $m^{2^n}/4^n$ constraints. Thus, as a decision procedure, Fourier–Motzkin variable elimination is only suitable for a relatively small set of constraints and a small number of variables.

5.5 The Omega Test

5.5.1 Problem Description

The Omega test is an algorithm to decide the satisfiability of a conjunction of linear constraints over integer variables. It can be seen as a variant of the Fourier–Motzkin algorithm (Sect. 5.4). Both are not considered to be the fastest decision procedures, but they are used for existential quantifier elimination, a topic that will only be covered later, in Chap. 9.

Each conjunct is assumed to be either an equality of the form

[3] There are extensions of Simplex to strict inequalities. See, for example, [108].

$$\sum_{i=1}^{n} a_i x_i = b \tag{5.54}$$

or a nonstrict inequality of the form

$$\sum_{i=1}^{n} a_i x_i \leq b \; . \tag{5.55}$$

The coefficients a_i are assumed to be integers; if they are not, by making use of the assumption that the coefficients are rational, the problem can be transformed into one with integer coefficients by multiplying the constraints by the least common multiple of the denominators. In Sect. 5.6, we show how strict inequalities can be transformed into nonstrict inequalities.

The run time of the Omega test depends on the size of the coefficients a_i. It is therefore desirable to transform the constraints such that small coefficients are obtained. This can be done by dividing the coefficients $a_1, \ldots, a_n$ of each constraint by their greatest common divisor g. The resulting constraint is called *normalized.* If the constraint is an equality constraint, this results in

$$\sum_{i=1}^{n} \frac{a_i}{g} \, x_i = \frac{b}{g} \; . \tag{5.56}$$

If g does not divide b exactly, the system is unsatisfiable. If the constraint is an inequality, one can tighten the constraint by rounding down the constant:

$$\sum_{i=1}^{n} \frac{a_i}{g} \, x_i \leq \left\lfloor \frac{b}{g} \right\rfloor . \tag{5.57}$$

More simplifications of this kind are described in Sect. 5.6.

Example 5.12. The equality $3x + 3y = 2$ can be normalized to $x + y = 2/3$, which is unsatisfiable. The constraint $8x + 6y \leq 0$ can be normalized to obtain $4x + 3y \leq 0$. The constraint $1 \leq 4y$ can be tightened to obtain $1 \leq y$. ▟

As in the case of Fourier–Motzkin, equality and inequality constraints are treated separately; all equality constraints are removed before inequalities are considered.

5.5.2 Equality Constraints

In order to eliminate an equality of the form of (5.54), we first check if there is a variable x_j with a coefficient 1 or -1, i.e., $|a_j| = 1$. If yes, we transform the constraint as follows. Without loss of generality, assume $j = n$. We isolate x_n:

$$x_n = \frac{b}{a_n} - \sum_{i=1}^{n-1} \frac{a_i}{a_n} x_i \; . \tag{5.58}$$

The variable x_n can now be substituted by the right-hand side of (5.58) in all constraints.

If there is no variable with a coefficient 1 or -1, we cannot simply divide by the coefficient, as this would result in nonintegral coefficients. Instead, the algorithm proceeds as follows: it determines the variable that has the nonzero coefficient with the smallest absolute value. Assume again that x_n is chosen, and that $a_n > 0$. The Omega test transforms the constraints iteratively until some coefficient becomes 1 or -1. The variable with that coefficient can then be eliminated as above.

$\boxed{a \,\widehat{\mathrm{mod}}\, b}$

For this transformation, a new binary operator $\widehat{\mathrm{mod}}$, called **symmetric modulo**, is defined as follows:

$$a \,\widehat{\mathrm{mod}}\, b \doteq\ a - b \cdot \left\lfloor \frac{a}{b} + \frac{1}{2} \right\rfloor \ . \tag{5.59}$$

The symmetric modulo operator is very similar to the usual modular arithmetic operator. If $a \bmod b < b/2$, then $a \,\widehat{\mathrm{mod}}\, b = a \bmod b$. If $a \bmod b$ is greater than or equal to $b/2$, b is deducted, and thus

$$a \,\widehat{\mathrm{mod}}\, b = \begin{cases} a \bmod b & : a \bmod b < b/2 \\ (a \bmod b) - b & : \text{otherwise} \ . \end{cases} \tag{5.60}$$

We leave the proof of this equivalence as an exercise (see Problem 5.12).

Our goal is to derive a term that can replace x_n. For this purpose, we define $m \doteq\ a_n + 1$, introduce a new variable σ, and add the following new constraint:

$$\sum_{i=1}^{n} (a_i \,\widehat{\mathrm{mod}}\, m)\, x_i = m\sigma + b \,\widehat{\mathrm{mod}}\, m \ . \tag{5.61}$$

We split the sum on the left-hand side to obtain

$$(a_n \,\widehat{\mathrm{mod}}\, m) x_n = m\sigma + b \,\widehat{\mathrm{mod}}\, m - \sum_{i=1}^{n-1} (a_i \,\widehat{\mathrm{mod}}\, m)\, x_i \ . \tag{5.62}$$

Since $a_n \,\widehat{\mathrm{mod}}\, m = -1$ (see Problem 5.14), this simplifies to

$$x_n = -m\sigma - b \,\widehat{\mathrm{mod}}\, m + \sum_{i=1}^{n-1} (a_i \,\widehat{\mathrm{mod}}\, m)\, x_i \ . \tag{5.63}$$

The right-hand side of (5.63) is used to replace x_n in all constraints. Any equality from the original problem (5.54) is changed as follows:

$$\sum_{i=1}^{n-1} a_i x_i + a_n \left(-m\sigma - b \,\widehat{\mathrm{mod}}\, m + \sum_{i=1}^{n-1} (a_i \,\widehat{\mathrm{mod}}\, m)\, x_i \right) = b \ , \tag{5.64}$$

which can be rewritten as

$$-a_n m\sigma + \sum_{i=1}^{n-1} (a_i + a_n(a_i \mathbin{\widehat{\mathrm{mod}}} m))\, x_i = b + a_n(b \mathbin{\widehat{\mathrm{mod}}} m)\ . \tag{5.65}$$

Since $a_n = m - 1$, this simplifies to

$$\begin{array}{l} -a_n m\sigma + \sum_{i=1}^{n-1} ((a_i - (a_i \mathbin{\widehat{\mathrm{mod}}} m)) + m(a_i \mathbin{\widehat{\mathrm{mod}}} m))\, x_i = \\ b - (b \mathbin{\widehat{\mathrm{mod}}} m) + m(b \mathbin{\widehat{\mathrm{mod}}} m)\ . \end{array} \tag{5.66}$$

Note that $a_i - (a_i \mathbin{\widehat{\mathrm{mod}}} m)$ is equal to $m\lfloor a_i/m + 1/2 \rfloor$, and thus all terms are divisible by m. Dividing (5.66) by m results in

$$-a_n\sigma + \sum_{i=1}^{n-1} (\lfloor a_i/m + 1/2 \rfloor + (a_i \mathbin{\widehat{\mathrm{mod}}} m))\, x_i = \lfloor b/m + 1/2 \rfloor + (b \mathbin{\widehat{\mathrm{mod}}} m)\ . \tag{5.67}$$

The absolute value of the coefficient of σ is the same as the absolute value of the original coefficient a_n, and it seems that nothing has been gained by this substitution. However, observe that the coefficient of x_i can be bounded as follows (see Problem 5.13):

$$|\lfloor a_i/m + 1/2 \rfloor + (a_i \mathbin{\widehat{\mathrm{mod}}} m)| \le \frac{5}{6}|a_i|\ . \tag{5.68}$$

Thus, the absolute values of the coefficients in the equality are strictly smaller than their previous values. As the coefficients are always integral, repeated application of equality elimination eventually generates a coefficient of 1 or -1 on some variable. This variable can then be eliminated directly, as described earlier (see (5.58)).

Example 5.13. Consider the following formula:

$$\begin{array}{rrl} -3x_1 & +2x_2 & = 0 \\ 3x_1 & +4x_2 & = 3\ . \end{array} \tag{5.69}$$

The variable x_2 has the coefficient with the smallest absolute value ($a_2 = 2$). Thus, $m = a_2 + 1 = 3$, and we add the following constraint (see (5.61)):

$$(-3 \mathbin{\widehat{\mathrm{mod}}}\ 3)x_1 + (2 \mathbin{\widehat{\mathrm{mod}}}\ 3)x_2 = 3\sigma\ . \tag{5.70}$$

This simplifies to $x_2 = -3\sigma$. Substituting -3σ for x_2 results in the following problem:

$$\begin{array}{rrl} -3x_1 & -6\sigma & = 0 \\ 3x_1 & -12\sigma & = 3\ . \end{array} \tag{5.71}$$

Division by m results in

$$\begin{array}{rrl} -x_1 & -2\sigma & = 0 \\ x_1 & -4\sigma & = 1\ . \end{array} \tag{5.72}$$

As expected, the coefficient of x_1 has decreased. We can now substitute x_1 by $4\sigma + 1$, and obtain $-6\sigma = 1$, which is unsatisfiable. ▟

5.5.3 Inequality Constraints

Once all equalities have been eliminated, the algorithm attempts to find a solution for the remaining inequalities. The control flow of Algorithm 5.5.1 is illustrated in Fig. 5.4. As in the Fourier–Motzkin procedure, the first step is to choose a variable to be eliminated. Subsequently, the three subprocedures *Real-Shadow*, *Dark-Shadow*, and *Gray-Shadow* produce new constraint sets, which are solved recursively.

Note that many of the subproblems generated by the recursion are actually identical. An efficient implementation uses a hash table that stores the solutions of previously solved problems.

Algorithm 5.5.1: Omega-Test

Input: A conjunction of constraints C
Output: "Satisfiable" if C is satisfiable, and "Unsatisfiable" otherwise

```
if $C$ only contains one variable then
    Solve and return result;                    ▷ (solving this problem is trivial)

Otherwise, choose a variable $v$ that occurs in $C$;
$C_R$ := Real-Shadow($C, v$);
if Omega-Test($C_R$) = "Unsatisfiable" then                ▷ Recursive call
    return "Unsatisfiable";

$C_D$ := Dark-Shadow($C, v$);
if Omega-Test($C_D$) = "Satisfiable" then                  ▷ Recursive call
    return "Satisfiable";

if $C_R = C_D$ then                                        ▷ Exact projection?
    return "Unsatisfiable";

$C_G^1, \ldots, C_G^n$ := Gray-Shadow($C, v$);
for all $i \in \{1, \ldots, n\}$ do
    if Omega-Test($C_G^i$) = "Satisfiable" then            ▷ Recursive call
        return "Satisfiable";

return "Unsatisfiable";
```

Checking the Real Shadow

Even though the Omega test is concerned with constraints over integers, the first step is to check if there are integer solutions in the relaxed problem,

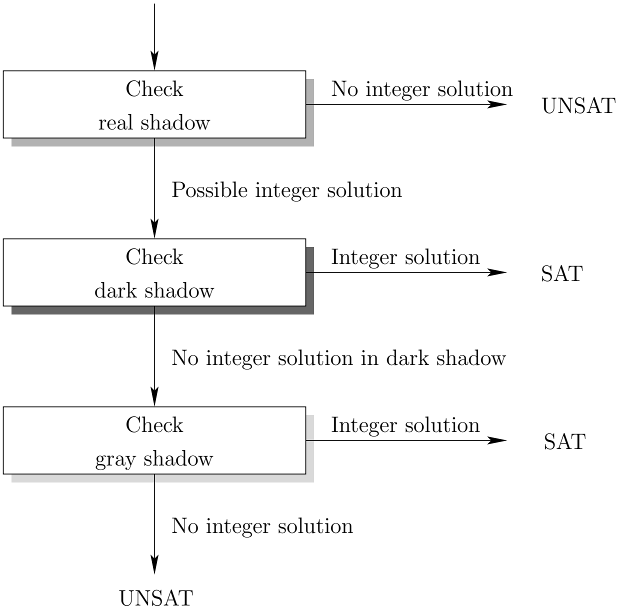


Fig. 5.4. Overview of the Omega test

which is called the *real shadow*. The real shadow is the same projection that the Fourier–Motzkin procedure uses. The Omega test is then called recursively to check if the projection contains an integer. If there is no such integer, then there is no integer solution to the original system either, and the algorithm concludes that the system is unsatisfiable.

Assume that the variable to be eliminated is denoted by z. As in the case of the Fourier–Motzkin procedure, all pairs of lower and upper bounds have to be considered. Variables that are not bounded both ways can be removed, together with all constraints that contain them.

Let $\beta \leq bz$ and $cz \leq \gamma$ be constraints, where c and b are positive integer constants and γ and β denote the remaining linear expressions. Consequently, β/b is a lower bound on z, and γ/c is an upper bound on z. The new constraint is obtained by multiplying the lower bound by c and the upper bound by b:

$$\begin{array}{cc} \text{Lower bound} & \text{Upper bound} \\ \hline \beta \leq bz & cz \leq \gamma \\ c\beta \leq cbz & cbz \leq b\gamma \end{array} \qquad (5.73)$$

The existence of such a variable z implies

$$c\beta \leq b\gamma \; . \qquad (5.74)$$

Example 5.14. Consider the following set of constraints:

$$\begin{aligned} 2y &\leq x \\ 8y &\geq 2 + x \\ 2y &\leq 3 - x \ . \end{aligned} \tag{5.75}$$

The triangle spanned by these constraints is depicted in Fig. 5.5. Assume that we decide to eliminate x. In this case, the combination of the two constraints $2y \leq x$ and $8y \geq 2 + x$ results in $8y - 2 \geq 2y$, which simplifies to $y \geq 1/3$. The two constraints $2y \leq x$ and $2y \leq 3 - x$ combine into $2y \leq 3 - 2y$, which simplifies to $y \leq 3/4$. Thus, $1/3 \leq y \leq 3/4$ must hold, which has no integer solution. The set of constraints is therefore unsatisfiable. ▟

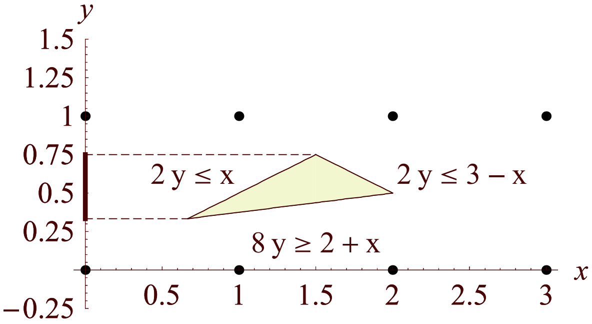


Fig. 5.5. Computing the real shadow: eliminating x

The converse of this observation does not hold, i.e., if we find an integer solution within the real shadow, this does not guarantee that the original set of constraints has an integer solution. This is illustrated by the following example.

Example 5.15. Consider the same set of constraints as in Example 5.14. This time, eliminate y instead of x. This projection is depicted in Fig. 5.6. We obtain $2/3 \leq x \leq 2$, which has two integer solutions. The triangle, on the other hand, contains no integer solution. ▟

The real shadow is an overapproximating projection, as it contains more solutions than does the original problem. The next step in the Omega test is to compute an underapproximating projection, i.e., if that projection contains an integer solution, so does the original problem. This projection is called the *dark* shadow.

Checking the Dark Shadow

The name *dark shadow* is motivated by optics. Assume that the object we are projecting is partially translucent. Places that are "thicker" will project

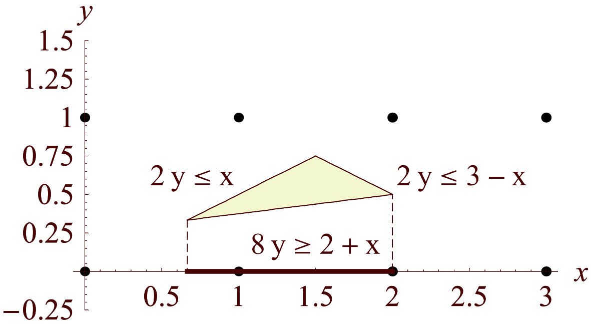


Fig. 5.6. Computing the real shadow: eliminating y

a darker shadow. In particular, a dark area in the shadow where the object is thicker than 1 must have at least one integer above it.

After the first phase of the algorithm, we know that there is a solution to the real shadow, i.e., $c\beta \leq b\gamma$. We now aim at determining if there is an integer z such that $c\beta \leq cbz \leq b\gamma$, which is equivalent to

$$\exists z \in \mathbb{Z}.\ \frac{\beta}{b} \leq z \leq \frac{\gamma}{c}\ . \tag{5.76}$$

Assume that (5.76) does not hold. Let i denote $\lfloor \beta/b \rfloor$, i.e., the largest integer that is smaller than β/b. Since we have assumed that there is no integer between β/b and γ/c,

$$i < \frac{\beta}{b} \leq \frac{\gamma}{c} < i+1 \tag{5.77}$$

holds. This situation is illustrated in Fig. 5.7.

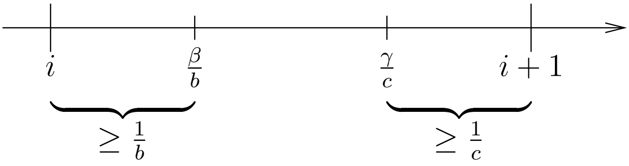


Fig. 5.7. Computing the dark shadow

Since β/b and γ/c are not integers themselves, the distances from these points to the closest integer are greater than the fractions $1/b$ and $1/c$, respectively:

$$\frac{\beta}{b} - i \geq \frac{1}{b}\ , \tag{5.78}$$

$$i + 1 - \frac{\gamma}{c} \geq \frac{1}{c}\ . \tag{5.79}$$

The proof is left as an exercise (Problem 5.11). By summing (5.78) and (5.79), we obtain

$$\frac{\beta}{b} + 1 - \frac{\gamma}{c} \geq \frac{1}{c} + \frac{1}{b}\ , \tag{5.80}$$

which is equivalent to

$$c\beta - b\gamma \geq -cb + c + b\ . \tag{5.81}$$

By multiplying this inequality by -1, we obtain

$$b\gamma - c\beta \leq cb - c - b\ . \tag{5.82}$$

In order to show a contradiction to our assumption, we need to show the negation of (5.82). Exploiting the fact that c, b are integers, the negation of (5.82) is

$$b\gamma - c\beta \geq cb - c - b + 1\ , \tag{5.83}$$

or simply

$$b\gamma - c\beta \geq (c-1)(b-1)\ . \tag{5.84}$$

Thus, if (5.84) holds, our assumption is wrong, which means that we have a guarantee that there exists an integer solution.

Observe that, if either $c = 1$ or $b = 1$, the formula (5.84) is identical to the real shadow (5.74), i.e., the dark and real shadow are the same. In this case, the projection is exact, and it is sufficient to check the *real shadow*. When choosing variables to eliminate, preference should be given to variables that result in an exact projection, that is, to variables with coefficient 1.

Checking the Gray Shadow

We know that any integer solution must also be in the real shadow. Let **R** denote this area. Now assume that we have found no integer in the dark shadow. Let **D** denote the area of the dark shadow. $\boxed{\mathbf{R}}$ $\boxed{\mathbf{D}}$

Thus, if **R** and **D** do not coincide, there is only one remaining area in which an integer solution can be found: an area around the dark shadow, which, staying within the optical analogy, is called the *gray shadow*.

Any solution must satisfy

$$c\beta \leq cbz \leq b\gamma\ . \tag{5.85}$$

Furthermore, we already know that the dark shadow does not contain an integer, and thus we can exclude this area from the search. Therefore, besides (5.85), any solution has to satisfy (5.82):

$$c\beta \leq cbz \leq b\gamma \quad \wedge \quad b\gamma - c\beta \leq cb - c - b \; . \tag{5.86}$$

This is equivalent to

$$c\beta \leq cbz \leq b\gamma \quad \wedge \quad b\gamma \leq cb - c - b + c\beta \; , \tag{5.87}$$

which implies

$$c\beta \leq cbz \leq cb - c - b + c\beta \; . \tag{5.88}$$

Dividing by c, we obtain

$$\beta \leq bz \leq \beta + \frac{cb - c - b}{c} \; . \tag{5.89}$$

The Omega test proceeds by simply trying possible values of bz between these two bounds. Thus, a new constraint

$$bz = \beta + i \tag{5.90}$$

is formed and combined with the original problem for each integer i in the range $0, \ldots, (cb - c - b)/c$. If any one of the resulting new problems has a solution, so does the original problem.

The number of subproblems can be reduced by determining the largest coefficient c of z in any upper bound for z. The new constraints generated for the other upper bounds are already covered by the constraints generated for the upper bound with the largest c.

5.6 Preprocessing

In this section, we examine several simple preprocessing steps for both linear and integer linear systems without objective functions. Preprocessing the set of constraints can be done regardless of the decision procedure chosen.

5.6.1 Preprocessing of Linear Systems

Two simple preprocessing steps for linear systems are the following:

1. Consider the set of constraints

$$x_1 + x_2 \leq 2, \; x_1 \leq 1, \; x_2 \leq 1 \; . \tag{5.91}$$

The first constraint is redundant. In general, for a set

$$S = \left\{ a_0x_0 + \sum_{j=1}^{n} a_jx_j \le b,\ l_j \le x_j \le u_j \text{ for } j = 0, \ldots, n \right\} , \tag{5.92}$$

the constraint

$$a_0x_0 + \sum_{j=1}^{n} a_jx_j \le b \tag{5.93}$$

is redundant if

$$\sum_{j | a_j > 0} a_ju_j + \sum_{j | a_j < 0} a_jl_j \le b \ . \tag{5.94}$$

To put this in words, a "$\le$" constraint in the above form is redundant if assigning values equal to their upper bounds to all of its variables that have a positive coefficient, and assigning values equal to their lower bounds to all of its variables that have a negative coefficient, results in a value less than or equal to b, the constant on the right-hand side of the inequality.

2. Consider the following set of constraints:

$$2x_1 + x_2 \le 2,\ x_2 \ge 4,\ x_1 \le 3 \ . \tag{5.95}$$

From the first and second constraints, $x_1 \le -1$ can be derived, which means that the bound $x_1 \le 3$ can be tightened. In general, if $a_0 > 0$, then

$$x_0 \le \left(b - \sum_{j | j > 0, a_j > 0} a_jl_j - \sum_{j | a_j < 0} a_ju_j \right) / a_0 \ , \tag{5.96}$$

and if $a_0 < 0$, then

$$x_0 \ge \left(b - \sum_{j | a_j > 0} a_jl_j - \sum_{j | j > 0, a_j < 0} a_ju_j \right) / a_0 \ . \tag{5.97}$$

5.6.2 Preprocessing of Integer Linear Systems

The following preprocessing steps are applicable to integer linear systems:

1. Multiply every constraint by the smallest common multiple of the coefficients and constants in this constraint, in order to obtain a system with integer coefficients.[4]
2. After the previous preprocessing has been applied, strict inequalities can be transformed into nonstrict inequalities as follows:

[4] This assumes that the coefficients and constants in the system are rational. The case in which the coefficients can be nonrational is of little value and is rarely considered in the literature.

$$\sum_{1 \leq i \leq n} a_i x_i < b \tag{5.98}$$

is replaced with

$$\sum_{1 \leq i \leq n} a_i x_i \leq b - 1 \; . \tag{5.99}$$

The case in which b is fractional is handled by the previous preprocessing step.

For the special case of **0–1 linear systems** (integer linear systems in which all the variables are constrained to be either 0 or 1), some preprocessing steps are illustrated by the following examples:

1. Consider the constraint

$$5x_1 \; - \; 3x_2 \leq 4 \; , \tag{5.100}$$

from which we can conclude that

$$x_1 = 1 \implies x_2 = 1 \; . \tag{5.101}$$

Hence, the constraint

$$x_1 \leq x_2 \tag{5.102}$$

can be added.

2. From

$$x_1 + x_2 \leq 1, \;\; x_2 \geq 1 \; , \tag{5.103}$$

we can conclude $x_1 = 0$.

Generalization of these examples is left for Problem 5.8.

5.7 Difference Logic

5.7.1 Introduction

A popular fragment of linear arithmetic is called **difference logic**.

Definition 5.16 (difference logic). *The syntax of a formula in* difference logic *is defined by the following rules:*

$$\begin{aligned} \mathit{formula} &: \mathit{formula} \wedge \mathit{formula} \mid \mathit{atom} \\ \mathit{atom} &: \mathit{identifier} - \mathit{identifier}\ \mathit{op}\ \mathit{constant} \\ \mathit{op} &: \; \leq \; \mid \; < \end{aligned}$$

Here, we consider the case in which the variables are defined over $\mathbb{Q}$, the rationals. A similar definition exists for the case in which the variables are defined over $\mathbb{Z}$ (see Problem 5.18). Solving both variants is polynomial, whereas, recall, linear arithmetic over $\mathbb{Z}$ is NP-complete.

Some other convenient operands can be modeled with the grammar above:

- $x - y = c$ is the same as $x - y \leq c \wedge y - x \leq -c$.
- $x - y \geq c$ is the same as $y - x \leq -c$.
- $x - y > c$ is the same as $y - x < -c$.
- A constraint with one variable such as $x < 5$ can be rewritten as $x - x_0 < 5$, where x_0 is a special variable not used so far in the formula, called the "zero variable". In any satisfying assignment, its value must be 0.

As an example,

$$x < y + 5 \wedge y \leq 4 \wedge x = z - 1 \tag{5.104}$$

can be rewritten in difference logic as

$$x - y < 5 \wedge y - x_0 \leq 4 \wedge x - z \leq -1 \wedge z - x \leq 1. \tag{5.105}$$

A more important variant, however, is one in which an arbitrary Boolean structure is permitted. We describe one application of this variant by the following example.

Example 5.17. We are given a finite set of n jobs, each of which consists of a chain of operations. There is a finite set of m machines, each of which can handle at most one operation at a time. Each operation needs to be performed during an uninterrupted period of given length on a given machine. The **job-shop scheduling** problem is to find a schedule, that is, an allocation of the operations to time intervals on the machines that has a minimal total length.

More formally, given a set of machines

$$M = \{m_1, \ldots, m_m\} , \tag{5.106}$$

job J^i with $i \in \{1, \ldots, n\}$ is a sequence of n_i pairs of the form (machine, duration):

$$J^i = (m^i_1, d^i_1), \ldots, (m^i_{n_i}, d^i_{n_i}) , \tag{5.107}$$

such that $m^i_1, \ldots, m^i_{n_i}$ are elements of M. The durations can be assumed to be rational numbers. We denote by O the multiset of all operations from all jobs. For an operation $v \in O$, we denote its machine by $M(v)$ and its duration by $\tau(v)$.

A schedule is a function that defines, for each operation v, its starting time $S(v)$ on its specified machine $M(v)$. A schedule S is *feasible* if the following three constraints hold:

1. First, the starting time of all operations is greater than or equal to 0:

$$\forall v \in O.\ S(v) \geq 0 . \tag{5.108}$$

2. Second, for every pair of consecutive operations $v_i, v_j \in O$ in the same job, the second operation does not start before the first ends:

$$S(v_i) + \tau(v_i) \leq S(v_j) . \tag{5.109}$$

3. Finally, every pair of different operations $v_i, v_j \in O$ scheduled on the same machine ($M(v_i) = M(v_j)$) is mutually exclusive:

$$S(v_i) + \tau(v_i) \leq S(v_j) \vee S(v_j) + \tau(v_j) \leq S(v_i) \; . \tag{5.110}$$

The length of the schedule S is defined as

$$\max_{v \in O} S(v) + \tau(v) \; . \tag{5.111}$$

The objective is to find a feasible schedule S that minimizes this length. As usual, we can define the decision problem associated with this optimization problem by removing the objective function and adding a constraint that forces the value of this function to be smaller than some constant.

It should be clear that a job-shop scheduling problem can be formulated with difference logic. Note the disjunction in (5.110). ▟

5.7.2 A Decision Procedure for Difference Logic

Recall that in this chapter we present only decision procedures for conjunctive fragments. The Boolean structure is dealt with by DPLL(T), as described in Chap. 3.

Definition 5.18 (inequality graph for nonstrict inequalities). *Let S be a set of difference predicates and let the* inequality graph $G(V, E)$ *be the graph comprising one edge (x, y) with weight c for every constraint of the form $x - y \leq c$ in S.*

Given a difference logic formula φ with nonstrict inequalities only, the inequality graph corresponding to the set of difference predicates in φ can be used for deciding φ, on the basis of the following theorem:

Theorem 5.19. *Let φ be a conjunction of difference constraints, and let G be the corresponding inequality graph. Then φ is satisfiable if and only if there is no negative cycle in G.*

The proof of this theorem is left as an exercise (Problem 5.15). The extension of Definition 5.18 and Theorem 5.19 to general difference logic (which includes both strict and nonstrict inequalities) is left as an exercise as well (see Problem 5.16).

By Theorem 5.19, deciding a difference logic formula amounts to searching for a negative cycle in a graph. This can be done with the **Bellman–Ford algorithm** [82] for finding the single-source shortest paths in a directed weighted graph, in time $O(|V| \cdot |E|)$ (to make the graph single-source, we introduce a new node and add an edge with weight 0 from this node to each of the roots of the original graph). Although finding the shortest paths is not our goal, we exploit a side-effect of this algorithm: if there exists a negative cycle in the graph, the algorithm finds it and aborts.

5.8 Problems

5.8.1 Warm-up Exercises

Problem 5.1 (linear systems). Consider the following linear system, which we denote by S:

$$\begin{array}{rl} x_1 \geq & -x_2 + \frac{11}{5} \\ x_1 \leq & x_2 + \frac{1}{2} \\ x_1 \geq & 3x_2 - 3 \ . \end{array} \tag{5.112}$$

(a) Check with Simplex whether S is satisfiable, as described in Sect. 5.2.
(b) Using the Fourier–Motzkin procedure, compute the range within which x_2 has to lie in a satisfying assignment.
(c) Consider a problem S', similar to S, but where the variables are forced to be integer. Check with branch and bound whether S' is satisfiable. To solve the relaxed problem, you can use a Simplex implementation (there are many of these on the Web).

5.8.2 The Simplex Method

Problem 5.2 (Simplex). Compute a satisfying assignment for the following problem using the general Simplex method:

$$\begin{array}{rrrrrr} 2x_1 & +2x_2 & +2x_3 & +2x_4 & \leq & 2 \\ 4x_1 & +x_2 & +x_3 & -4x_4 & \leq & -2 \\ x_1 & +2x_2 & +4x_3 & +2x_4 & = & 4 \ . \end{array} \tag{5.113}$$

Problem 5.3 (complexity). Give a conjunction of linear constraints over reals with n variables (that is, the size of the instance is parameterized) such that the number of iterations of the general Simplex algorithm is exponential in n.

Problem 5.4 (difference logic with Simplex). What is the worst-case run time of the general Simplex algorithm if applied to a conjunction of difference logic constraints?

Problem 5.5 (strict inequalities with Simplex). Extend the general Simplex algorithm with strict inequalities.

Problem 5.6 (soundness). Assume that the general Simplex algorithm returns “UNSAT”. Show a method for deriving a proof of unsatisfiability.

5.8.3 Integer Linear Systems

Problem 5.7 (complexity of ILP-feasibility). Prove that the feasibility problem for integer linear programming is NP-hard.[5]

Problem 5.8 (0–1 ILP). A 0–1 integer linear system is an integer linear system in which all variables are constrained to be either 0 or 1. Show how a 0–1 integer linear system can be translated to a Boolean formula. What is the complexity of the translation?

Problem 5.9 (simplifications for 0–1 ILP). Generalize the simplification demonstrated in (5.100)–(5.103).

Problem 5.10 (Gomory cuts). Find Gomory cuts corresponding to the following results from the general Simplex algorithm:

1. $x_4 = x_1 - 2.5x_2 + 2x_3$ where $\alpha := \{x_4 \mapsto 3.25, x_1 \mapsto 1, x_2 \mapsto -0.5, x_3 \mapsto 0.5\}$, x_2 and x_3 are at their upper bound, and x_1 is at its lower bound.
2. $x_4 = -0.5x_1 - 2x_2 + 3.5x_3$ where $\alpha := \{x_4 \mapsto 0.25, x_1 \mapsto 1, x_2 \mapsto 0.5, x_3 \mapsto 0.5\}$, x_1 and x_3 are at their lower bound, and x_2 is at its upper bound.

5.8.4 Omega Test

Problem 5.11 (integer fractions). Recall the definition $i = \lfloor \frac{\beta}{b} \rfloor$ in Sect. 5.5.3. Show that

- $\frac{\beta}{b} - i \geq \frac{1}{b}$, and
- $i + 1 - \dfrac{\gamma}{c} \geq \dfrac{1}{c}$

Recall that all coefficients are assumed to be integers.

Problem 5.12 (eliminating equalities). Show that

$$a \mathbin{\widehat{\bmod}} b = \begin{cases} a \bmod b & : a \bmod b < b/2 \\ (a \bmod b) - b & : \text{otherwise} \end{cases} \tag{5.114}$$

holds. Use the fact that

$$a/b = \lfloor a/b \rfloor + \frac{a \bmod b}{b} \ .$$

Problem 5.13 (eliminating equalities). Show that the absolute values of the coefficients of the variables x_i are reduced to at most 5/6 of their previous values after substituting σ:

[5] In fact it is NP-complete, but membership in NP is more difficult to prove. The proof makes use of a small-model-property argument.

$$|\lfloor a_i/m + 1/2 \rfloor + (a_i \mathbin{\widehat{\bmod}} m)| \leq 5/6|a_i| \,. \tag{5.115}$$

Problem 5.14 (eliminating equalities). The elimination of x_n relies on the fact that the coefficient of x_n in the newly added constraint is -1. Let a_n denote the coefficient of x_n in the original constraint. Let $m = a_n + 1$, and assume that $a_n \geq 2$. Show that $a_n \mathbin{\widehat{\bmod}} m = -1$.

5.8.5 Difference Logic

Problem 5.15 (difference logic). Prove Theorem 5.19.

Problem 5.16 (inequality graphs for difference logic). Extend Definition 5.18 and Theorem 5.19 to general difference logic formulas (i.e., where both strong and weak inequalities are allowed).

Problem 5.17 (difference logic). Give a reduction of difference logic to SAT. What is the complexity of the reduction?

Problem 5.18 (integer difference logic). Show a reduction from the problem of integer difference logic to difference logic.

Problem 5.19 (theory propagation for difference logic). Recall the notion of exhaustive theory propagation that was studied in Sect. 3.4.2. Suggest an efficient procedure that performs exhaustive theory propagation for the case of difference logic (difference logic is presented in Sect. 5.7).

5.9 Bibliographic Notes

The Fourier–Motzkin variable elimination algorithm is the earliest documented method for solving linear inequalities. It was discovered in 1826 by Fourier, and rediscovered by Motzkin in 1936. A somewhat more efficient way to eliminate variables called **virtual substitution** was suggested by Loos and Weispfenning [183].

The Simplex method was introduced by Dantzig in 1947 [84]. There are several variations of and improvements on this method, most notably the *revised Simplex method*, which most industrial implementations use. This variant has an apparent advantage on large and sparse LP problems, which seem to characterize LP problems in practice. The variant of the general Simplex algorithm that we presented in Sect. 5.2 was proposed by Dutertre and de Moura [108] in the context of DPLL(T), a technique that we saw in Chap. 3. Its main advantage is that it works efficiently with incremental operations, i.e., constraints can be added and removed with little effort.

Linear programs are a very popular modeling formalism for solving a wide range of problems in science and engineering, finance, logistics, and so on.

Consider, for example, how LP is used for computing an optimal placement of gates in an integrated circuit [152]. The popularity of this method led to a large industry of LP solvers, some of which are sold for tens of thousands of dollars per copy. A classical reference to linear and integer linear programming is the book by Schrijver [252]. Other resources on the subject that we found useful include publications by Wolsey [288], Hillier and Lieberman [142], and Vanderbei [278].

Gomory cutting planes are due to a paper published by Ralph Gomory in 1963 [134]. For many years, the operations research community considered Gomory cuts impractical for large problems. There were several refinements of the original method and empirical studies that revived this technique, especially in the context of the related optimization problem. See, for example, the work of Balas et al. [11]. The variant we described is suitable for working with the general Simplex algorithm, and its description here is based on [109].

The Omega test was introduced by Pugh as a method for deciding integer linear arithmetic within an optimizing compiler [233]. It is an extension of the Fourier–Motzkin variable elimination. For an example of an application of the Omega test inside a Fortran compiler, see [2]. A much earlier work following similar lines to those of the Omega test is by Paul Williams [283]. Williams' work, in turn, is inspired by Presburger's paper from 1929 [232].

Difference logic was recognized as an interesting fragment of linear arithmetic by Pratt [231]. He considered "separation theory", which is the conjunctive fragment of what we call difference logic. He observed that most inequalities in verification conditions are of this form. Disjunctive difference logic was studied in M. Mahfoudh's PhD thesis [185] and in [186], among other places. A reduction of difference logic to SAT was studied in [267] (in this particular paper and some later papers, this theory fragment is called "separation logic", after Pratt's separation theory—not to be confused with the separation logic that is discussed in Chap. 8). The main reason for the renewed interest in this fragment is due to interest in **timed automata**: the verification conditions arising in this problem domain are difference logic formulas.

In general, the amount of research and writing on linear systems is immense, and in fact most universities offer courses dedicated to this subject. Most of the research was and still is conducted in the operations research community.

5.10 Glossary

The following symbols were used in this chapter:

Symbol	**Refers to ...**	**First used on page ...**
l_i, u_i	Constants bounding the i-th variable from below and above	99
m	The number of linear constraints in the original problem formulation	99
n	The number of variables in the original problem formulation	100
A	Coefficient matrix	100
$\mathbf{x}$	The vector of the variables in the original problem formulation	101
$\mathcal{B}, N$	The sets of basic and nonbasic variables, respectively	102
α	A full assignment (to both basic and nonbasic variables)	102
θ	See (5.13)	104
β_i	Upper or lower bound	114
$\widehat{\mathrm{mod}}$	Symmetric modulo	117

6

Bit Vectors

6.1 Bit-Vector Arithmetic

The design of computer systems is error-prone, and, thus, decision procedures for reasoning about such systems are highly desirable. A computer system uses *bit vectors* to encode information, for example, numbers. Owing to the finite domain of these bit vectors, the semantics of operations such as addition no longer matches what we are used to when reasoning about unbounded types, for example, the natural numbers.

6.1.1 Syntax

The subset of bit-vector arithmetic that we consider is defined by the following grammar:

$$\begin{aligned}
\mathit{formula} &: \mathit{formula} \wedge \mathit{formula} \mid \neg \mathit{formula} \mid (\,\mathit{formula}\,) \mid \mathit{atom} \\
\mathit{atom} &: \mathit{term}\ \mathit{rel}\ \mathit{term} \mid \mathit{Boolean\text{-}Identifier} \mid \mathit{term}[\,\mathit{constant}\,] \\
\mathit{rel} &: < \ \mid \ = \\
\mathit{term} &: \mathit{term}\ \mathit{op}\ \mathit{term} \mid \mathit{identifier} \mid \ \sim \mathit{term} \mid \mathit{constant} \mid \mathit{atom}\,?\,\mathit{term} : \mathit{term} \mid \\
&\quad\ \mathit{term}[\,\mathit{constant} : \mathit{constant}\,] \mid \mathit{ext}(\,\mathit{term}\,) \\
\mathit{op} &: + \mid - \mid \cdot \mid / \mid \ll \mid \gg \mid \& \mid \ \shortmid \ \mid \oplus \mid \circ
\end{aligned}$$

As usual, other useful operators such as "$\vee$", "$\neq$", and "$\geq$" can be obtained using Boolean combinations of the operators that appear in the grammar. Most operators have a straightforward meaning, but a few operators are unique to bit-vector arithmetic. The unary operator "$\sim$" denotes bitwise negation. The function *ext* denotes sign and zero extension (the meanings of these operators are explained in Sect. 6.1.3). The ternary operator $c?a\!:\!b$ is a case-split: the operator evaluates to a if c holds, and to b otherwise. The operators "$\ll$" and "$\gg$" denote left and right shifts, respectively. The operator "$\oplus$" denotes bitwise XOR. The binary operator "$\circ$" denotes concatenation of bit vectors.

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_6

Motivation

As an example to describe our motivation, the following formula obviously holds over the integers:

$$(x - y > 0) \iff (x > y)\ . \tag{6.1}$$

If x and y are interpreted as finite-width bit vectors, however, this equivalence no longer holds, owing to possible **overflow** of the subtraction operation. As another example, consider the following small C program:

```
unsigned char number = 200;
number = number + 100;
printf("Sum: %d\n", number);
```

This program may return a surprising result, as most architectures use eight bits to represent variables with type `unsigned char`:

$$\begin{array}{r}
\texttt{11001000} = 200\\
+\ \texttt{01100100} = 100\\
\hline
= \texttt{00101100} = 44
\end{array}$$

When represented with eight bits by a computer, 200 is stored as 11001000. Adding 100 results in an overflow, as the ninth bit of the result is discarded.

The meaning of operators such as "+" is therefore defined by means of *modular* arithmetic. However, the problem of reasoning about bit vectors extends beyond that of overflow and modular arithmetic. For efficiency reasons, programmers use bit-level operators to encode as much information as possible into the number of bits available.

As an example, consider the implementation of a propositional SAT solver. Recall the definition of a *literal* (Definition 1.11): a literal is a variable or its negation. Propositional SAT solvers that operate on formulas in CNF have to store a large number of such literals. We assume that we have numbered the variables that occur in the formula, and denote the variables by $x_1, x_2, \ldots$.

The DIMACS standard for CNF uses signed numbers to encode a literal, e.g., the literal $\neg x_3$ is represented as -3. The fact that we use signed numbers for the encoding avoids the use of one bit vector to store the sign. On the other hand, it reduces the possible number of variables to $2^{31} - 1$ (the index 0 cannot be used any more), but this is still more than sufficient for any practical purpose.

In order to extract the index of a variable, we have to perform a case-split on the sign of the bit vector, for example, as follows:

```
unsigned variable_index(int literal) {
  if(literal < 0)
    return -literal;
  else
    return literal;
}
```

The branch needed to implement the `if` statement in the program above slows down the execution of the program, as it is hard to predict for the branch prediction mechanisms of modern processors. Most SAT solvers therefore use a different encoding: the least significant bit of the bit vector is used to encode the sign of the literal, and the remaining bits encode the variable. The index of the variable can then be extracted by means of a bit-vector right-shift operation:

```
unsigned variable_index(unsigned literal) {
  return literal >> 1;
}
```

Similarly, the sign can be obtained by means of a bitwise AND operation:

```
bool literal_sign(unsigned literal) {
  return literal & 1;
}
```

The bitwise right-shift operation and the bitwise AND are implemented in most microprocessors, and both can be executed efficiently. Such bitwise operators also frequently occur in hardware design. Reasoning about such artifacts requires *bit-vector arithmetic*.

6.1.2 Notation

We use a simple variant of Church's **$\boldsymbol{\lambda}$-notation** in order to define vectors easily. A lambda expression for a bit vector with l bits has the form

$$\lambda i \in \{0, \ldots, l-1\}.\, f(i)\ , \tag{6.2}$$

where $f(i)$ is an expression that denotes the value of the i-th bit.

The use of the λ-operator to denote bit vectors is best explained by an example.

Example 6.1. Consider the following expressions:

- The expression
$$\lambda i \in \{0, \ldots, l-1\}.\, 0 \tag{6.3}$$
denotes the l-bit bit vector that consists only of zeros.
- A λ-expression is simply another way of defining a function *without* giving it a name. Thus, instead of defining a function z with
$$z(i) \doteq\ 0\ , \tag{6.4}$$
we can simply write $\lambda i \in \{0, \ldots, l-1\}.\, 0$ for z.
- The expression
$$\lambda i \in \{0, \ldots, 7\}.\, \begin{cases} 0 : i \text{ is even} \\ 1 : \text{otherwise} \end{cases} \tag{6.5}$$
denotes the bit vector 10101010.

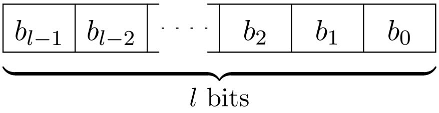


Fig. 6.1. A bit vector b with l bits. The bit number i is denoted by b_i

- The expression
$$\lambda i \in \{0, \ldots, l-1\}. \neg x_i \tag{6.6}$$
denotes the bitwise negation of the vector x.

▟

We omit the domain of i from the lambda expression if the number of bits is clear from the context.

6.1.3 Semantics

We now give a formal definition of the meaning of a bit-vector arithmetic formula. We first clarify what a bit vector is.

Definition 6.2 (bit vector). *A* bit vector b *is a vector of bits with a given length* l (*or dimension*)*:*

$$b : \{0, \ldots, l-1\} \longrightarrow \{0, 1\} . \tag{6.7}$$

$\boxed{bvec_l}$

The set of all 2^l *bit vectors of length* l *is denoted by* $bvec_l$*. The* i*-th bit of the bit vector* b *is denoted by* b_i *(Fig. 6.1).*

The meaning of a bit-vector formula obviously depends on the width of the bit-vector variables in it. This applies even if no arithmetic is used. As an example,

$$x \neq y \wedge x \neq z \wedge y \neq z \tag{6.8}$$

is unsatisfiable for bit vectors x, y, and z that are one bit wide, but satisfiable for larger widths.

We sometimes use bit vectors that encode positive numbers only (unsigned bit vectors), and also bit vectors that encode both positive and negative numbers (signed bit vectors). Thus, each expression is associated with a **type**. The type of a bit-vector expression is

1. the width of the expression in bits, and
2. whether it is signed or unsigned.

We restrict the presentation to bit vectors that have a fixed, given length, as bit-vector arithmetic becomes undecidable as soon as arbitrary-width bit vectors are permitted. The width is known in most problems that arise in practice.

In order to clarify the type of an expression, we add indices in square brackets to the operator and operands in order to denote the bit width (this is not to be confused with b_l, which denotes bit l of b). As an example, $a_{[32]} \cdot_{[32]} b_{[32]}$ denotes the multiplication of a and b. Both the result and the operands are 32 bits wide, and the remaining 32 bits of the result are discarded. The expression $a_{[8]} \circ_{[24]} b_{[16]}$ denotes the concatenation of a and b and is in total 24 bits wide. In most cases, the width is clear from the context, and we therefore usually omit the subscript.

Bitwise Operators

The meanings of bitwise operators can be defined through the bit vectors that they yield. The binary bitwise operators take two l-bit bit vectors as arguments and return an l-bit bit vector. As an example, the signature of the bitwise OR operator "|" is

$$|_{[l]} \; : (bvec_l \times bvec_l) \longrightarrow bvec_l \; . \tag{6.9}$$

Using the λ-notation, the bitwise OR operator is defined as follows:

$$a \,|\, b \doteq \; \lambda i. \, (a_i \vee b_i) \; . \tag{6.10}$$

All the other bitwise operators are defined in a similar manner. In the following, we typically provide both the signature and the definition together.

Arithmetic Operators

The meaning of a bit-vector formula with arithmetic operators depends on the *interpretation* of the bit vectors that it contains. There are many ways to encode numbers using bit vectors. The most commonly used encodings for integers are the **binary encoding** for unsigned integers and **two's complement** for signed integers.

Definition 6.3 (binary encoding). *Let x denote a natural number, and $b \in bvec_l$ a bit vector. We call b a* binary encoding *of x iff*

$$x = \langle b \rangle_U \; , \tag{6.11}$$

where $\langle b \rangle_U$ is defined as follows: $\boxed{\langle\cdot\rangle_U}$

$$\begin{aligned} \langle\cdot\rangle_U : bvec_l &\longrightarrow \{0, \ldots, 2^l - 1\} \; , \\ \langle b \rangle_U &\doteq \; \textstyle\sum_{i=0}^{l-1} b_i \cdot 2^i . \end{aligned} \tag{6.12}$$

The bit b_0 is called the **least significant bit***, and the bit b_{l-1} is called the* **most significant bit***.*

Binary encoding can be used to represent nonnegative integers only. One way of encoding negative numbers as well is to use one of the bits as a **sign bit**.

A naive way of using a sign bit is to simply negate the number if a designated bit is set, for example, the most significant bit. As an example, 1001 could be interpreted as -1 instead of 1. This encoding is hardly ever used in practice.[1] Instead, most microprocessor architectures implement the **two's complement** encoding.

Definition 6.4 (two's complement). *Let x denote a natural number, and $b \in bvec_l$ a bit vector. We call b the* two's complement *of x iff*

$$x = \langle b \rangle_S \ , \tag{6.13}$$

$\boxed{\langle \cdot \rangle_S}$

where $\langle b \rangle_S$ is defined as follows:

$$\begin{aligned} \langle \cdot \rangle_S : bvec_l &\longrightarrow \{-2^{l-1}, \ldots, 2^{l-1} - 1\} \ , \\ \langle b \rangle_S &:= -2^{l-1} \cdot b_{l-1} + \textstyle\sum_{i=0}^{l-2} b_i \cdot 2^i \ . \end{aligned} \tag{6.14}$$

The bit with index $l-1$ is called the sign bit *of b.*

Example 6.5. Some encodings of integers in binary and two's complement are

$$\begin{aligned} \langle 11001000 \rangle_U &= 200 \ , \\ \langle 11001000 \rangle_S &= -128 + 64 + 8 = -56 \ , \\ \langle 01100100 \rangle_S &= 100 \ . \end{aligned}$$

Note that the meanings of the relational operators ">", "<", "≤", "≥", the multiplicative operators "·", "/", and the right-shift operator ">>" depend on whether a binary encoding or a two's complement encoding is used for the operands, which is why the encoding of the bit vectors is part of the type. We use the subscript U for a binary encoding (unsigned) and the subscript S for a two's complement encoding (signed). We may omit this subscript if the encoding is clear from the context, or if the meaning of the operator does not depend on the encoding (this is the case for most operators).

As suggested by the example at the beginning of this chapter, arithmetic on bit vectors has a wraparound effect: if the number of bits required to represent the result exceeds the number of bits available, the additional bits of the result are discarded, i.e., the result is truncated. This corresponds to a **modulo** operation, where the base is 2^l. We write

$$x = y \mod b \tag{6.15}$$

to denote that x and y are equal modulo b. The use of modulo arithmetic allows a straightforward definition of the interpretation of all arithmetic operators:

[1] The main reason for this is the fact that it makes the implementation of arithmetic operators such as addition more complicated, and that there are two encodings for 0, namely 0 and -0.

- Addition and subtraction:

$$a_{[l]} +_U b_{[l]} = c_{[l]} \iff \langle a \rangle_U + \langle b \rangle_U = \langle c \rangle_U \mod 2^l \,, \tag{6.16}$$

$$a_{[l]} -_U b_{[l]} = c_{[l]} \iff \langle a \rangle_U - \langle b \rangle_U = \langle c \rangle_U \mod 2^l \,, \tag{6.17}$$

$$a_{[l]} +_S b_{[l]} = c_{[l]} \iff \langle a \rangle_S + \langle b \rangle_S = \langle c \rangle_S \mod 2^l \,, \tag{6.18}$$

$$a_{[l]} -_S b_{[l]} = c_{[l]} \iff \langle a \rangle_S - \langle b \rangle_S = \langle c \rangle_S \mod 2^l \,. \tag{6.19}$$

Note that $a +_U b = a +_S b$ and $a -_U b = a -_S b$ (see Problem 6.7), and thus the U/S subscript can be omitted from the addition and subtraction operands. A semantics for mixed-type expressions is also easily defined, as shown in the following example:

$$a_{[l]U} +_U b_{[l]S} = c_{[l]U} \iff \langle a \rangle + \langle b \rangle_S = \langle c \rangle \mod 2^l \,. \tag{6.20}$$

- Unary minus:

$$-a_{[l]} = b_{[l]} \iff -\langle a \rangle_S = \langle b \rangle_S \mod 2^l \,. \tag{6.21}$$

- Relational operators:

$$a_{[l]U} < b_{[l]U} \iff \langle a \rangle_U < \langle b \rangle_U \,, \tag{6.22}$$

$$a_{[l]S} < b_{[l]S} \iff \langle a \rangle_S < \langle b \rangle_S \,, \tag{6.23}$$

$$a_{[l]U} < b_{[l]S} \iff \langle a \rangle_U < \langle b \rangle_S \,, \tag{6.24}$$

$$a_{[l]S} < b_{[l]U} \iff \langle a \rangle_S < \langle b \rangle_U \,. \tag{6.25}$$

The semantics for the other relational operators such as "$\geq$" follows the same pattern. Note that ANSI-C compilers do not implement the relational operators on operands with mixed encodings the way they are formalized above (see Problem 6.6). Instead, the signed operand is converted to an unsigned operand, which does not preserve the meaning expected by many programmers.

- Multiplication and division:

$$a_{[l]} \cdot_U b_{[l]} = c_{[l]} \iff \langle a \rangle_U \cdot \langle b \rangle_U = \langle c \rangle_U \mod 2^l \,, \tag{6.26}$$

$$a_{[l]} /_U b_{[l]} = c_{[l]} \iff \langle a \rangle_U / \langle b \rangle_U = \langle c \rangle_U \mod 2^l \,, \tag{6.27}$$

$$a_{[l]} \cdot_S b_{[l]} = c_{[l]} \iff \langle a \rangle_S \cdot \langle b \rangle_S = \langle c \rangle_S \mod 2^l \,, \tag{6.28}$$

$$a_{[l]} /_S b_{[l]} = c_{[l]} \iff \langle a \rangle_S / \langle b \rangle_S = \langle c \rangle_S \mod 2^l \,. \tag{6.29}$$

The semantics of multiplication is independent of whether the arguments are interpreted as unsigned or two's complement (see Problem 6.8), and thus the U/S subscript can be omitted. This does not hold in the case of division.

- The extension operator: converting a bit vector to a bit vector with more bits is called **zero extension** in the case of an unsigned bit vector, and **sign extension** in the case of a signed bit vector. Let $l \leq m$. The value that is encoded does not change:

$$ext_{[m]U}(a_{[l]}) = b_{[m]U} \iff \langle a \rangle_U = \langle b \rangle_U \ , \tag{6.30}$$

$$ext_{[m]S}(a_{[l]}) = b_{[m]S} \iff \langle a \rangle_S = \langle b \rangle_S \ . \tag{6.31}$$

- Shifting: the left-shift operator "<<" takes two operands and shifts the first one to the left as many times as is given by the respective value of the second operand. The width of the left-hand-side operand is called the *width of the shift*, whereas the width of the right-hand-side operator is the *width of the shift distance*. The vector is filled up with zeros from the right:

$$a_{[l]} << b_U = \lambda i \in \{0, \ldots, l-1\}. \begin{cases} a_{i-\langle b \rangle_U} & : i \geq \langle b \rangle_U \\ 0 & : \text{otherwise} \ . \end{cases} \tag{6.32}$$

  See also Problem 6.5. The meaning of the right-shift ">>" operator depends on the encoding of the first operand: if it uses binary encoding (which, recall, is for unsigned bit vectors), zeros are inserted from the left. This is called a **logical right shift**:

$$a_{[l]U} >> b_U = \lambda i \in \{0, \ldots, l-1\}. \begin{cases} a_{i+\langle b \rangle_U} & : i < l - \langle b \rangle_U \\ 0 & : \text{otherwise} \ . \end{cases} \tag{6.33}$$

  If the first operand uses two's complement encoding, the sign bit of a is replicated. This is also called an **arithmetic right shift**:

$$a_{[l]S} >> b_U = \lambda i \in \{0, \ldots, l-1\}. \begin{cases} a_{i+\langle b \rangle_U} & : i < l - \langle b \rangle_U \\ a_{l-1} & : \text{otherwise} \ . \end{cases} \tag{6.34}$$

  The shift operators are rarely defined for a signed shift distance. An option could be to flip the direction of the shift in case b is negative; e.g., a left shift with distance -1 is a right shift with distance 1.

6.2 Deciding Bit-Vector Arithmetic with Flattening

6.2.1 Converting the Skeleton

$\boxed{\mathcal{B}}$

The most commonly used decision procedure for bit-vector arithmetic is called *flattening*.[2] Algorithm 6.2.1 implements this technique. For a given bit-vector arithmetic formula φ, the algorithm computes an equisatisfiable propositional formula $\mathcal{B}$, which is then passed to a SAT solver.

$\boxed{At(\varphi)}$

Let $At(\varphi)$ denote the set of atoms in φ. As a first step, the algorithm replaces the atoms in φ with new Boolean variables. We denote the variable that replaces an atom $a \in At(\varphi)$ by $e(a)$, and call this the **propositional encoder** of a. The resulting formula is denoted by $e(\varphi)$. We call it the **propositional skeleton** of φ. The propositional skeleton is the expression that is assigned to $\mathcal{B}$ initially.

$\boxed{e(\varphi)}$

Let $T(\varphi)$ denote the set of terms in φ. The algorithm then assigns a vector of new Boolean variables to each bit-vector term in $T(\varphi)$. We use $e(t)$ to denote this vector of variables for a given $t \in T(\varphi)$, and $e(t)_i$ to denote the variable for the bit with index i of the term t. The width of $e(t)$ matches the width of the term t. Note that, so far, we have used e to denote three different, but related things: a propositional encoder of an atom, a propositional formula resulting from replacing all atoms of a formula with their respective propositional encoders, and a propositional encoder of a term.

$\boxed{T(\varphi)}$

$\boxed{e(t)}$

The algorithm then iterates over the terms and atoms of φ, and computes a constraint for each of them. The constraint is returned by the function BV-Constraint, and is added as a conjunct to $\mathcal{B}$.

Algorithm 6.2.1: BV-Flattening

Input: A formula φ in bit-vector arithmetic
Output: An equisatisfiable Boolean formula $\mathcal{B}$

```
function BV-Flattening
    $\mathcal{B}$:=$e(\varphi)$;                        ▷ the propositional skeleton of $\varphi$
    for each $t_{[l]} \in T(\varphi)$ do
        for each $i \in \{0, \ldots, l-1\}$ do
            set $e(t)_i$ to a new Boolean variable;
    for each $a \in At(\varphi)$ do
        $\mathcal{B}$:=$\mathcal{B} \wedge$ BV-Constraint($e, a$);
    for each $t_{[l]} \in T(\varphi)$ do
        $\mathcal{B}$:=$\mathcal{B} \wedge$ BV-Constraint($e, t$);
    return $\mathcal{B}$;
```

The constraint that is needed for a particular atom a or term t depends on the atom or term, respectively. In the case of a bit vector or a Boolean variable, no constraint is needed, and BV-Constraint returns TRUE. If t is a bit-vector constant $C_{[l]}$, the following constraint is generated:

$$\bigwedge_{i=0}^{l-1} (C_i \iff e(t)_i) \; . \tag{6.35}$$

Otherwise, t must contain a bit-vector operator. The constraint that is needed depends on this operator. The constraints for the bitwise operators are

[2] In colloquial terms, this technique is sometimes referred to as "*bit-blasting*".

straightforward. As an example, consider bitwise OR, and let $t = a \,|_{[l]} b$. The constraint returned by BV-Constraint is

$$\bigwedge_{i=0}^{l-1} ((a_i \vee b_i) \iff e(t)_i) \ . \tag{6.36}$$

The constraints for the other bitwise operators follow the same pattern.

6.2.2 Arithmetic Operators

The constraints for the arithmetic operators often follow implementations of these operators as a *circuit*. There is an abundance of literature on how to build efficient circuits for various arithmetic operators. However, experiments with various alternative circuits have shown that the simplest ones usually burden the SAT solver the least. We begin by defining a one-bit adder, also called a **full adder**.

Definition 6.6 (full adder). *A* full adder *is defined using the two functions carry and sum. Both of these functions take three input bits a, b, and cin as arguments. The function carry calculates the carry-out bit of the adder, and the function sum calculates the sum bit:*

$$sum(a, b, cin) \doteq (a \oplus b) \oplus cin \ , \tag{6.37}$$
$$carry(a, b, cin) \doteq (a \wedge b) \vee ((a \oplus b) \wedge cin) \ . \tag{6.38}$$

We can extend this definition to adders for bit vectors of arbitrary length.

Definition 6.7 (carry bits). *Let x and y denote two l-bit bit vectors and cin a single bit. The* carry bits *c_0 to c_l are defined recursively as follows:*

$$c_i \doteq \begin{cases} cin & : i = 0 \\ carry(x_{i-1}, y_{i-1}, c_{i-1}) & : otherwise \ . \end{cases} \tag{6.39}$$

Definition 6.8 (adder). *An l-bit* adder *maps two l-bit bit vectors x, y and a carry-in bit cin to their sum and a carry-out bit. Let c_i denote the i-th carry bit as in Definition 6.7. The function add is defined using the carry bits c_i:*

$$add(x, y, cin) \doteq \langle result, cout \rangle \ , \tag{6.40}$$
$$result_i \doteq sum(x_i, y_i, c_i) \quad for\ i \in \{0, \ldots, l-1\} \ , \tag{6.41}$$
$$cout \doteq c_n \ . \tag{6.42}$$

The circuit equivalent of this construction is called a *ripple carry adder*. One can easily implement the constraint for $t = a + b$ using an adder with $cin = 0$:

$$\bigwedge_{i=0}^{l-1} (add(a, b, 0).result_i \iff e(t)_i) \ . \tag{6.43}$$

One can prove by induction on l that (6.43) holds if and only if $\langle a\rangle_U + \langle b\rangle_U = \langle e(t)\rangle_U \bmod 2^l$, which shows that the constraint complies with the semantics.

Subtraction, where $t = a - b$, is implemented with the same circuit by using the following constraint (recall that $\sim b$ is the bitwise negation of b):

$$\bigwedge_{i=0}^{l-1} (\mathit{add}(a, {\sim} b, 1).\mathit{result}_i \iff e(t)_i) \ . \tag{6.44}$$

This implementation makes use of the fact that $\langle ({\sim} b) + 1\rangle_S = -\langle b\rangle_S \bmod 2^l$ (see Problem 6.9).

Relational Operators

The equality $a =_{[l]} b$ is implemented using simply a conjunction:

$$\bigwedge_{i=0}^{l-1} a_i = b_i \iff e(t) \ . \tag{6.45}$$

The relation $a < b$ is transformed into $a - b < 0$, and an adder is built for the subtraction, as described above. Thus, b is negated and the carry-in bit of the adder is set to TRUE. The result of the relation $a < b$ depends on the encoding. In the case of unsigned operands, $a < b$ holds if the carry-out bit *cout* of the adder is FALSE:

$$\langle a\rangle_U < \langle b\rangle_U \iff \neg \mathit{add}(a, {\sim} b, 1).\mathit{cout} \ . \tag{6.46}$$

In the case of signed operands, $a < b$ holds if and only if $(a_{l-1} = b_{l-1}) \neq \mathit{cout}$:

$$\langle a\rangle_S < \langle b\rangle_S \iff (a_{l-1} \iff b_{l-1}) \oplus \mathit{add}(a,\ b, 1).\mathit{cout} \ . \tag{6.47}$$

Comparisons involving mixed encodings are implemented by extending both operands by one bit, followed by a signed comparison.

Shifts

Recall that we call the width of the left-hand-side operand of a shift (the vector that is to be shifted) the *width of the shift*, whereas the width of the right-hand-side operand is the *width of the shift distance.*

We restrict the left and right shifts as follows: the width l of the shift must be a power of two, and the width of the shift distance n must be $\log_2 l$.

With this restriction, left and right shifts can be implemented by using the following construction, which is called the *barrel shifter*. The shifter is split into n stages. Stage s can shift the operand by 2^s bits or leave it unaltered. The function *ls* is defined recursively for $s \in \{-1, \ldots, n-1\}$:

$$ls(a_{[l]}, b_{[n]U}, -1) \doteq a \, , \tag{6.48}$$

$$ls(a_{[l]}, b_{[n]U}, s) \doteq \lambda i \in \{0, \ldots, l-1\}. \begin{cases} (ls(a,b,s-1))_{i-2^s} & : i \geq 2^s \wedge b_s \\ (ls(a,b,s-1))_i & : \neg b_s \\ 0 & : \text{otherwise} \, . \end{cases} \tag{6.49}$$

The barrel shifter construction needs only $O(n \log n)$ logical operators, in contrast to the naive implementation, which requires $O(n^2)$ operators.

Multiplication and Division

Multipliers can be implemented following the most simplistic circuit design, which uses the *shift-and-add* idea. The function *mul* is defined recursively for $s \in \{-1, \ldots, n-1\}$, where n denotes the width of the second operand:

$$mul(a, b, -1) \doteq 0, \tag{6.50}$$

$$mul(a, b, s) \doteq mul(a, b, s-1) + (b_s ? (a << s) : 0) \, . \tag{6.51}$$

A division $a /_U b$ is implemented by adding two constraints:

$$b \neq 0 \implies e(t) \cdot b + r = a \, , \tag{6.52}$$

$$b \neq 0 \implies r < b \, . \tag{6.53}$$

The variable r is a new bit vector of the same width as b, and contains the remainder. The signed-division and modulo operations are done in a similar way.

6.3 Incremental Bit Flattening

6.3.1 Some Operators Are Hard

For some operators, the size of the formula generated by BV-Constraint may be large. As an example, consider the formula for a single multiplier with n bits. The table in Fig. 6.2 shows the number of variables and the number of CNF clauses that are generated from the formula using Tseitin's encoding (see Sect. 1.3).

In addition to the sheer size of these formulas, their symmetry and connectivity is a burden on the decision heuristic of state-of-the-art propositional SAT solvers. As a consequence, formulas with multipliers are often very hard to solve. Similar observations hold for other arithmetic operators such as division and modulo.

As an example, consider the following bit-vector formula:

$$a \cdot b = c \ \wedge \ b \cdot a \neq c \ \wedge \ x < y \ \wedge \ x > y \, . \tag{6.54}$$

n	Number of variables	Number of clauses
8	313	1001
16	1265	4177
24	2857	9529
32	5089	17057
64	20417	68929

Fig. 6.2. The size of the constraint for an n-bit multiplier expression after Tseitin's transformation

When this formula is encoded into CNF, a SAT instance with about 11 000 variables is generated for a width of 32 bits. This formula is obviously unsatisfiable. There are two reasons for this: the first two conjuncts are inconsistent, and independently, the last two conjuncts are inconsistent. The decision heuristics of most SAT solvers (see Chap. 2) are biased towards splitting first on variables that are used frequently, and thus favor decisions on a, b, and c. Consequently, they attempt to show unsatisfiability of the formula on the hard part, which includes the two multipliers. The "easy" part of the formula, which contains only two relational operators, is ignored. Most propositional SAT solvers cannot solve this formula in a reasonable amount of time.

In many cases, it is therefore beneficial to build the flattened formula $\mathcal{B}$ *incrementally.* Algorithm 6.3.1 is a realization of this idea: as before, we start with the propositional skeleton of φ. We then add constraints for the "inexpensive" operators, and omit the constraints for the "expensive" operators. The bitwise operators are typically inexpensive, whereas arithmetic operators are expensive. The encodings with missing constraints can be considered an *abstraction* of φ, and thus the algorithm is an instance of the abstraction–refinement procedure introduced in Sect. 4.4.

The current flattening $\mathcal{B}$ is passed to a propositional SAT solver. If $\mathcal{B}$ is unsatisfiable, so is the original formula φ. Recall the formula (6.54): as soon as the constraints for the second half of the formula are added to $\mathcal{B}$, the encoding becomes unsatisfiable, and we may conclude that (6.54) is unsatisfiable without considering the multipliers.

On the other hand, if $\mathcal{B}$ is satisfiable, one of two cases applies:

1. The original formula φ is unsatisfiable, but one (or more) of the omitted constraints is needed to show this.
2. The original formula φ is satisfiable.

In order to distinguish between these two cases, we can check whether the satisfying assignment produced by the SAT solver satisfies the constraints that we have omitted. As we might have removed variables, the satisfying assignment might have to be extended by setting the missing values to some constant, for example, zero. If this assignment satisfies all constraints, the second case applies, and the algorithm terminates.

Algorithm 6.3.1: Incremental-BV-Flattening

Input: A formula φ in bit-vector logic
Output: "Satisfiable" if the formula is satisfiable, and "Unsatisfiable" otherwise

```
function Incremental-BV-Flattening($\varphi$)
    $\mathcal{B} := e(\varphi)$;                              ▷ propositional skeleton of $\varphi$
    for each $t_{[l]} \in T(\varphi)$ do
        for each $i \in \{0, \ldots, l-1\}$ do
            set $e(t)_i$ to a new Boolean variable;
    while (TRUE) do
        $\alpha :=$ SAT-Solver($\mathcal{B}$);
        if $\alpha=$"Unsatisfiable" then
            return "Unsatisfiable";
        else
            Let $I \subseteq T(\varphi)$ be the set of terms that are inconsistent with the
                satisfying assignment;
            if $I = \emptyset$ then
                return "Satisfiable";
            else
                Select "easy" $F' \subseteq I$;
                for each $t_{[l]} \in F'$ do
                    $\mathcal{B}:=\mathcal{B} \wedge$ BV-Constraint($e, t$);
```

If this is not so, one or more of the terms for which the constraints were omitted is inconsistent with the assignment provided by the SAT solver. We denote this set of terms by I. The algorithm proceeds by selecting some of these terms, adding their constraints to $\mathcal{B}$, and reiterating. The algorithm terminates, as we strictly add more constraints with each iteration. In the worst case, all constraints from $T(\varphi)$ are added to the encoding.

6.3.2 Abstraction with Uninterpreted Functions

In many cases, omitting constraints for particular operators may result in a flattened formula that is too weak, and thus is satisfied by too many spurious models. On the other hand, the full constraint may burden the SAT solver too much. A compromise between the maximum strength of the full constraint and omitting the constraint altogether is to replace functions over bit vectors by uninterpreted functions (see Sect. 4.2). This technique is particularly effective when one is checking the equivalence of two models.

For example, let $a_1 \mathit{op}\, b_1$ and $a_2 \mathit{op}\, b_2$ be two terms, where *op* is some binary operator (for simplicity, assume that these are the only terms in the input formula that use *op*). Replace *op* with a new uninterpreted-function symbol G

to obtain instead $G(a_1, b_1)$ and $G(a_2, b_2)$. The resulting formula is abstract, and does not contain constraints that correspond to the flattening of *op*.

6.4 Fixed-Point Arithmetic

6.4.1 Semantics

Many applications, for example, in scientific computing, require arithmetic on numbers with a fractional part. High-end microprocessors offer support for **floating-point arithmetic** for this purpose. However, fully featured floating-point arithmetic is too heavyweight for many applications, such as control software embedded in vehicles, and computer graphics. In these domains, **fixed-point arithmetic** is a reasonable compromise between accuracy and complexity. Fixed-point arithmetic is also commonly supported by database systems, for example, to represent amounts of currency.

In fixed-point arithmetic, the representation of a number is partitioned into two parts, the *integer part* (also called the *magnitude*) and the *fractional part* (Fig. 6.3). The number of digits in the fractional part is fixed—hence the name "fixed point arithmetic". The number 1.980, for example, is a fixed-point number with a three-digit fractional part.

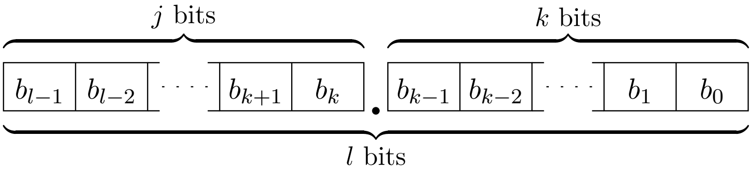


Fig. 6.3. A fixed-point bit vector b with a total of $j + k = l$ bits. The dot is called the radix point. The j bits before the dot represent the magnitude (the integer part), whereas the k bits after the dot represent the fractional part

The same principle can be applied to binary arithmetic, as captured by the following definition. Recall the definition of $\langle\cdot\rangle_S$ (two's complement) from Sect. 6.1.3.

Definition 6.9. *Given two bit vectors M and F with m and f bits, respectively, we define the rational number that is represented by $M.F$ as follows and denote it by $\langle M.F\rangle$:*

$$\begin{aligned} \langle\cdot\rangle &: \{0,1\}^{m+f} \longrightarrow \mathbb{Q}\,, \\ \langle M.F\rangle &:= \frac{\langle M \circ F\rangle_S}{2^f}\,. \end{aligned}$$

Example 6.10. Some encodings of rational numbers as fixed-point numbers with base 2 are

$$\begin{aligned}\langle 0.10\rangle &= 0.5\ ,\\ \langle 0.01\rangle &= 0.25\ ,\\ \langle 01.1\rangle &= 1.5\ ,\\ \langle 11111111.1\rangle &= -0.5\ .\end{aligned}$$

Some rational numbers are not precisely representable using fixed-point arithmetic in base 2: they can only be approximated. As an example, for $m = f = 4$, the two numbers that are closest to 1/3 are

$$\begin{aligned}\langle 0000.0101\rangle &= 0.3125\ ,\\ \langle 0000.0110\rangle &= 0.375\ .\end{aligned}$$

▟

Definition 6.9 gives us the semantics of fixed-point arithmetic. For example, the meaning of addition on bit vectors that encode fixed-point numbers can be defined as follows:

$$\begin{aligned}&a_M.a_F + b_M.b_F = c_M.c_F \iff\\ &\quad \langle a_M.a_F\rangle \cdot 2^f + \langle b_M.b_F\rangle \cdot 2^f = \langle c_M.c_F\rangle \cdot 2^f \mod 2^{m+f}\ .\end{aligned}$$

There are variants of fixed-point arithmetic that implement **saturation** instead of overflow semantics, that is, instead of wrapping around, the result remains at the highest or lowest number that can be represented with the given precision. Both the semantics and the flattening procedure are straightforward for this case.

6.4.2 Flattening

Fixed-point arithmetic can be flattened just as well as arithmetic using binary encoding or two's complement. We assume that the numbers on the left- and right-hand sides of a binary operator have the same numbers of bits, before and after the radix point. If this is not so, missing bits after the radix point can be added by padding the fractional part with zeros from the right. Missing bits before the radix point can be added from the left using sign extension.

The operators are encoded as follows:

- The bitwise operators are encoded exactly as in the case of binary numbers. Addition, subtraction, and the relational operators can also be encoded as in the case of binary numbers.
- Multiplication requires an alignment. The result of a multiplication of two numbers with f_1 and f_2 bits in the fractional part, respectively, is a number with $f_1 + f_2$ bits in the fractional part. Note that, most commonly, fewer bits are needed, and thus, the extra bits of the result have to be rounded off using a separate **rounding** step.

Example 6.11. Addition and subtraction are straight-forward, but note the need for sign-extension in the second sum:

$$\langle 00.1\rangle + \langle 00.1\rangle = \langle 01.0\rangle$$
$$\langle 000.0\rangle + \langle 1.0\rangle = \langle 111.0\rangle$$

The following examples illustrate multiplication without any subsequent rounding:

$$\langle 0.1\rangle \cdot \langle 1.1\rangle = \langle 0.11\rangle$$
$$\langle 1.10\rangle \cdot \langle 1.1\rangle = \langle 10.010\rangle$$

If needed, rounding towards zero, towards the next even number, or towards $\pm\infty$ can be applied in order to reduce the size of the fractional part; see Problem 6.10. ▟

There are many other encodings of numbers, which we do not cover here, e.g., binary-coded decimals (BCDs), or fixed-point formats with sign bit.

6.5 Problems

6.5.1 Semantics

Problem 6.1 (operators that depend on the encoding). Provide an example (with values of operands) that illustrates that the semantics depend on the encoding (signed vs. unsigned) for each of the following three operators: $>$, $/$, and $>>$.

Problem 6.2 (λ-notation). Define the meaning of $a_l \circ b_l$ using the λ-notation.

Problem 6.3 (negation). What is -10000000_S if the operand of the unary minus is a bit-vector constant?

Problem 6.4 (λ-notation). Define the meaning of $a_{[l]U} >>_{[l]U} b_{[m]S}$ and $a_{[l]S} >>_{[l]S} b_{[m]S}$ using modular arithmetic. Prove these definitions to be equivalent to the definition given in Sect. 6.1.3.

Problem 6.5 (shifts in hardware). What semantics of the left shift does the processor in your computer implement? You can use a program to test this, or refer to the specification of the CPU. Formalize the semantics.

Problem 6.6 (relations in hardware). What semantics of the $<$ operator does the processor in your computer implement if a signed integer is compared with an unsigned integer? Try this for the ANSI-C types `int`, `unsigned`, `char`, and `unsigned char`. Formalize the semantics, and specify the vendor and model of the CPU.

Problem 6.7 (two's complement addition). Prove

$$a_{[l]} +_U b_{[l]} = a_{[l]} +_S b_{[l]} . \tag{6.55}$$

Problem 6.8 (two's complement multiplication). Prove

$$a_{[l]} \cdot_U b_{[l]} = a_{[l]} \cdot_S b_{[l]} . \tag{6.56}$$

6.5.2 Bit-Level Encodings of Bit-Vector Arithmetic

Problem 6.9 (negation). Prove $\langle (\sim b) + 1 \rangle_S = -\langle b \rangle_S \mod 2^l$.

Problem 6.10 (relational operators). Prove the correctness of the flattening for "<" as given in Sect. 6.2, for:

(a) Unsigned operands
(b) Signed operands
(c) An unsigned and a signed operand

Problem 6.11 (rounding for fixed-point arithmetic). Formally specify the operator for rounding a fixed-point number with a fractional part of size f_1 to a fractional part of size $f_2 < f_1$ for the following cases:

(a) Rounding to zero
(b) Rounding to $-\infty$
(c) Rounding to the nearest even number

Problem 6.12 (flattening fixed-point arithmetic). Provide a flattening for the three rounding operators above.

6.5.3 Using Solvers for Linear Arithmetic

$\boxed{\llbracket b \rrbracket}$

We introduced decision procedures for linear arithmetic in Chap. 5. A restricted subset of bit-vector arithmetic can be translated into linear arithmetic over the integers. As preparation, we perform a number of transformations on the terms contained in a. We write $\llbracket b \rrbracket$ for the result of the transformation of any bit-vector arithmetic term b.

- Let $b >> d$ denote a bitwise right-shift term that is contained in a, where b is a term and d is a constant. It is replaced by $\llbracket b \rrbracket / 2^{\langle d \rangle}$, i.e.,

  $$\llbracket b >> d \rrbracket \doteq \llbracket b \rrbracket / 2^{\langle d \rangle} . \tag{6.57}$$

  Bitwise left shifts are handled in a similar manner.

- The bitwise negation of a term b is replaced by $-[\![b]\!]-1$:

$$[\![\sim b]\!] \doteq -[\![b]\!]-1\;. \tag{6.58}$$

- A bitwise AND term $b_{[l]}\&1$, where b is any term, is replaced by a new integer variable x subject to the following constraints over x and a second new integer variable σ:

$$0 \le x \le 1 \;\;\wedge\;\; [\![b]\!] = 2\sigma + x \;\;\wedge\;\; 0 \le \sigma < 2^{l-1}\;. \tag{6.59}$$

  A bitwise AND with other constants can be replaced using shifts. This can be optimized further by joining together groups of adjacent one-bits in the constant on the right-hand side.
- The bitwise OR is replaced with bitwise negation and bitwise AND.

We are now left with addition, subtraction, multiplication by a constant, and division by a constant.

As the next step, the division operators are removed from the constraints. As an example, the constraint $a/_{[32]}3 = b$ becomes $a = b \cdot_{[34]} 3$. Note that the bit width of the multiplication has to be increased in order to take overflow into account. The operands a and b are sign-extended if signed, and zero-extended if unsigned. After this preparation, we can assume the following form of the atoms without loss of generality:

$$c_1 \cdot t_1 +_{[l]} c_2 \cdot t_2 \;\; rel \;\; b\;, \tag{6.60}$$

where rel is one of the relational operators as defined in Sect. 6.1, c_1, c_2, and b are constants, and t_1 and t_2 are bit-vector identifiers with l bits. Sums with more than two addends can be handled in a similar way.

As we can handle additions efficiently, all scalar multiplications $c \cdot_{[l]} a$ with a small constant c are replaced by c additions. For example, $3 \cdot a$ becomes $a + a + a$.

At this point, we are left with predicates of the following form:

$$t_1 +_{[l]} t_2 \;\; rel \;\; b\;. \tag{6.61}$$

Given that t_1 and t_2 are l-bit unsigned vectors, we have $t_1 \in \{0, \ldots, 2^l - 1\}$ and $t_2 \in \{0, \ldots, 2^l - 1\}$, and, thus, $t_1 + t_2 \in \{0, \ldots, 2^{l+1} - 2\}$. Recall that the bit-vector addition in (6.61) will overflow if $t_1 + t_2$ is larger than $2^l - 1$. We use a case split to adjust the value of the sum in the case of an overflow and transform (6.61) into

$$((t_1 + t_2 \le 2^l - 1)\ ?\ t_1 + t_2 \;:\; (t_1 + t_2 - 2^l)) \;\; op \;\; b\;. \tag{6.62}$$

Based on this description, answer the following questions:

Problem 6.13 (translation to integer arithmetic). Translate the following bit-vector formula into a formula over integers:

$$x_{[8]} +_{[8]} 100 \leq 10_{[8]} \; . \tag{6.63}$$

Problem 6.14 (bitwise AND). Give a translation of

$$x_{[32]U} = y_{[32]U} \& 0\text{xffff0000} \tag{6.64}$$

into disjunctive integer linear arithmetic that is more efficient than that suggested by (6.59).

Problem 6.15 (scalar multiplications). Rewriting scalar multiplications $c \cdot_{[l]} a$ into c additions is inefficient if c is large owing to the cost of the subsequent splitting. Suggest an alternative that uses a new variable.

Problem 6.16 (addition without splitting). Can you propose a different translation for addition that does not use case splitting but uses a new integer variable instead?

Problem 6.17 (removing the conditional operator). Our grammar for integer linear arithmetic does not permit the conditional operator. Propose a linear-time method for removing them. Note that the conditional operators may be nested.

6.6 Bibliographic Notes

Tools and Applications

Bit-vector arithmetic was identified as an important logic for verification and equivalence checking in the hardware industry in [263]. The notation we use to annotate the type of the bit-vector expressions is taken from [50].

Early decision procedures for bit-vector arithmetic can be found in tools such as SVC [16] and ICS [113]. ICS used BDDs in order to decide properties of arithmetic operators, whereas SVC was based on a combination of a canonizer and a solver [18]. SVC has been superseded by CVC, and then CVC-Lite [14] and STP, both of which use a propositional SAT solver to decide the satisfiability of a circuit-based flattening of a bit-vector formula. ICS was superseded by YICES, which also uses flattening and a SAT solver.

Bit-vector arithmetic is now primarily used to model the semantics of programming languages. COGENT [75] decides the validity of ANSI-C expressions. ANSI-C expressions are drawn from a fragment of bit-vector arithmetic, extended with pointer logic (see Chap. 8). COGENT and related procedures have many applications besides checking verification conditions. As an example, see [37, 38] for an application of COGENT to database testing. In addition to deciding the validity of ANSI-C expressions, C32SAT [51], developed by Brummayer and Biere, is also able to determine if an expression always has a well-defined meaning according to the ANSI-C standard.

Bounded model checking (BMC) is a common source of bit-vector arithmetic decision problems [33]. BMC was designed originally for synchronous models, as frequently found in the hardware domain, for example. BMC has been adopted in other domains that result in bit-vector formulas, for example, software programs given in ANSI-C [72]. Further applications for decision procedures for bit-vector arithmetic are discussed in Chap. 12.

Translation to Integer Linear Arithmetic

Translations to integer linear arithmetic as described in Sect. 6.5.3 have been used for bit-vector decision problems found in the hardware verification domain. Brinkmann and Drechsler [50] translated a fragment of bit-vector arithmetic into ILP and used the Omega test as a decision procedure for the ILP problem. However, the work in [50] was aimed only at the data paths, and thus did not allow a Boolean part within the original formula. This was mended by Parthasarathy et al. [218] using an incremental encoding similar to the one described in Chap. 3.

IEEE Floating-Point Arithmetic

Decision procedures for IEEE floating-point arithmetic are useful for generating tests for software that uses such arithmetic. A semantics for formulas using IEEE binary floating-point arithmetic is given in the definition of the SMT-LIB FPA theory. IEEE floating-point arithmetic can be flattened into propositional logic by using circuits that implement floating-point units. Beyond flattening, approaches based on incremental refinement [49] and interval arithmetic [45] have been proposed. A theory for SMT solvers is proposed in [47].

State-of-the-Art Solvers

Current state-of-the-art decision procedures for bit-vector arithmetic apply heavy preprocessing to the formula, but ultimately rely on flattening a formula to propositional SAT [54, 189, 136]. The preprocessing is especially beneficial if the formula also contains large arrays, for example, for modeling memories [118, 188], or very expensive bit-vector operators such as multiplication or division. A method for generating encodings that are particularly well-suited for BCP is explained in [46]. The bit-vector category in the 2015 SMT competition was won by Boolector [52], which features an efficient decision procedure for the combination of bit-vector arithmetic with arrays.

6.7 Glossary

The following symbols were used in this chapter:

Symbol	**Refers to** ...	**First used on page** ...
$c?a : b$	Case split on condition c	135
λ	Lambda expressions	137
$bvec_l$	Set of bit vectors with l bits	138
$\langle \cdot \rangle_U$	Number encoded by binary encoding	139
$\langle \cdot \rangle_S$	Number encoded by two's complement	140
$A(\varphi)$	Set of atoms in φ	142
$T(\varphi)$	Set of terms in φ	143
c_i	Carry bit i	144
$[\![b]\!]$	Result of translation of bit-vector term b into linear arithmetic	152

7

Arrays

7.1 Introduction

The array is a basic datatype that is supported by most programming languages, and is consequently prevalent in software. It is also used for modeling the memory components of hardware. It is clear, then, that analysis of software or hardware requires the ability to decide formulas that contain arrays. This chapter introduces an array theory and two decision procedures for specific fragments thereof.

Let us begin with an example that illustrates the use of array theory for verifying an invariant of a loop.

Example 7.1. Consider the pseudocode fragment in Fig. 7.1. The main step of the correctness argument is to show that the assertion in line 7 follows from the assertion in line 5 when executing the assignment in line 6. A common way to do so is to generate **verification conditions**, e.g., using Hoare's axiom system. We obtain the following verification condition for the claim:

$$\begin{aligned} &(\forall x \in \mathbb{N}_0.\ x < i \implies \mathsf{a}[x] = 0) \\ \land\ &\mathsf{a}' = \mathsf{a}\{i \leftarrow 0\} \\ \implies &(\forall x \in \mathbb{N}_0.\ x \leq i \implies \mathsf{a}'[x] = 0)\ . \end{aligned} \tag{7.1}$$

The formula above contains two symbols that are specific to arrays: the *array index* operator $\mathsf{a}[x]$ and the *array update* operator $\mathsf{a}\{i \leftarrow 0\}$. We will explain the meaning of these operators later. The validity of (7.1) implies that the loop invariant is maintained. Our goal is to prove such formulas automatically, and indeed later in this chapter we will show how this can be done.

The array theory permits expressions over arrays, which are formalized as maps from an *index type* to an *element type*. We denote the index type by T_I, and the element type by T_E. The type of the arrays themselves is denoted by T_A, which is a shorthand for $T_I \longrightarrow T_E$, i.e., the set of functions that map an

$\boxed{T_I}$

$\boxed{T_E}$

$\boxed{T_A}$

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_7

```
a: array 0..99 of integer;
i: integer;

for i:=0 to 99 do
        assert($\forall x \in \mathbb{N}_0.\ x < i \implies$ a[$x$] $= 0$);
        a[i]:=0;
        assert($\forall x \in \mathbb{N}_0.\ x \leq i \implies$ a[$x$] $= 0$);
done;
assert($\forall x \in \mathbb{N}_0.\ x \leq 99 \implies$ a[$x$] $= 0$);
```

Fig. 7.1. Pseudocode fragment that initializes an array of size 100 with zeros, annotated with assertions

element of T_I to an element of T_E. Note that neither the set of indices nor the set of elements are required to be finite.

Let $a \in T_A$ denote an array. There are two basic operations on arrays:

$\boxed{a[i]}$

1. *Reading* an element with index $i \in T_I$ from a. The value of the element that has index i is denoted by $a[i]$. This operator is called the *array index* operator.

$\boxed{a\{i \leftarrow e\}}$

2. *Writing* an element with index $i \in T_I$. Let $e \in T_E$ denote the value to be written. The array a where element i has been replaced by e is denoted by $a\{i \leftarrow e\}$. This operator is called the *array update* or *array store* operator.

We call the theories used to reason about the indices and the elements the *index theory* and the *element theory*, respectively. The array theory is *parameterized* with the index and element theories. We can obtain multidimensional arrays by recursively defining $T_A(n)$ for n-dimensional arrays. For $n \geq 2$, we simply add $T_A(n-1)$ to the element type of $T_A(n)$.

The choice of the index and element theories will affect the expressiveness of the resulting array theory. As an instance, the index theory needs to permit existential and universal quantification in order to model properties such as "there exists an array element that is zero" or "all elements of the array are greater than zero". An example of a suitable index theory is Presburger arithmetic, i.e., linear arithmetic over integers (Chap. 5) with quantification (Chap. 9).

We start with a very general definition of the array theory. This theory is in general not decidable, however, and we therefore consider restrictions later on in order to obtain decision procedures.

7.1.1 Syntax

We define the syntax of the array theory as an extension to the combination of the index and element theories. Let $term_I$ and $term_E$ denote a term in these two theories, respectively. We begin by defining an array term $term_A$:

$$term_A : \textit{array-identifier} \mid term_A\{term_I \leftarrow term_E\}\ .$$

Next, we extend element terms to include array elements, i.e.,

$$\mathit{term}_E : \mathit{term}_A \,[\, \mathit{term}_I \,] \mid (\mathit{previous\ rules}) \; ,$$

where *previous rules* denote the grammatical rules that define term_E before this extension. Finally, we extend the possible predicates in the formula by allowing equalities between array terms:

$$\mathit{formula} : \mathit{term}_A = \mathit{term}_A \mid (\mathit{previous\ rules}) \; ,$$

where here *previous rules* refer to the grammatical rules defining *formula* before this extension. The extension of the grammar with explicit equality between arrays is redundant if the index theory includes quantification, since $a_1 = a_2$ for arrays a_1 and a_2 can also be written as $\forall i.\ a_1[i] = a_2[i]$.

7.1.2 Semantics

The meaning of the new atoms and terms in the array theory is given using three axioms.

The first axiom gives the obvious meaning to the array index operator. Two array index terms have the same value if the array is the same and if the index is the same.

$$\forall a_1 \in T_A.\ \forall a_2.\ \in T_A.\ \forall i \in T_I.\ \forall j \in T_I.\ (a_1 = a_2 \wedge i = j) \implies a_1[i] = a_2[j] \; . \tag{7.2}$$

The axiom used to define the meaning of the array update operator is the **read-over-write axiom**: after the value e has been written into array a at index i, the value of this array at index i is e. The value at any index $j \neq i$ matches that in the array before the write operation at index j:

$$\forall a \in T_A.\ \forall e \in T_E.\ \forall i \in T_I.\ \forall j \in T_I.\ a\{i \leftarrow e\}[j] = \begin{cases} e & : i = j \\ a[j] & : \text{otherwise} \; . \end{cases} \tag{7.3}$$

This axiom is necessary, for example, for proving (7.1).

Finally, we give the obvious meaning to equality over arrays with the **extensionality rule**:

$$\forall a_1 \in T_A.\ \forall a_2 \in T_A.\ (\forall i \in T_I.\ a_1[i] = a_2[i]) \implies a_1 = a_2 \; . \tag{7.4}$$

The array theory that includes the rule above is called the **extensional theory of arrays**.

7.2 Eliminating the Array Terms

We now present a method to translate a formula in the array theory into a formula that is a combination of the index theory and the element theory.

Aside: Array Bounds Checking in Programs

While the array theory uses arrays of unbounded size, array data structures in programs are of bounded size. If an index variable exceeds the size of an array in a program, the value returned may be undefined or a crash might occur. This situation is called an **array bounds violation**. In the case of a write operation, other data might be overwritten, which is often exploitable to gain control over a computer system from a remote location over a network. Checking that a program never violates any of its array bounds is therefore highly desirable.

Note, however, that checking array bounds in programs does not require the array theory; the question of whether an array index is within the bounds of a finite-size array requires one only to keep track of the *size* of the array, not of its contents.

As an example, consider the following program fragment, which is meant to move the elements of an array:

```
int a[N];

for(int i=0; i<N; i++)
  a[i]=a[i+1];
```

Despite the fact that the program contains an array, the verification condition for the array-bounds property does not require the array theory:

$$i < N \implies (i < N \wedge i + 1 < N) \, . \tag{7.5}$$

The translation is applicable if this combined theory includes uninterpreted functions and quantifiers over indices.

Consider Axiom (7.2), which defines the semantics of the array index operator. Now recall the definition of functional consistency, which we saw in Sect. 4.2.1. Informally, functional consistency requires that two applications of the same function must yield an equal result if their arguments are the same. It is evident that Axiom (7.2) is simply a special case of functional consistency.

We can therefore replace the array index operator by an uninterpreted function, as illustrated in the following example:

Example 7.2. Consider the following array theory formula, where a is an array with element type `char`:

$$(i = j \wedge a[j] = \texttt{'z'}) \implies a[i] = \texttt{'z'} \, . \tag{7.6}$$

The character constant `'z'` is a member of the element type. Let F_a denote the uninterpreted function introduced for the array a:

$$(i = j \wedge F_a(j) = \texttt{'z'}) \implies F_a(i) = \texttt{'z'} \, . \tag{7.7}$$

This formula can be shown to be valid with a decision procedure for equality and uninterpreted functions (Chap. 4). ▟

What about the array update operator? One way to model the array update is to replace each expression of the form $a\{i \leftarrow e\}$ by a fresh variable a' of type array. We then add two constraints that correspond directly to the two cases of the read-over-write axiom:

1. $a'[i] = e$ for the value that is written,
2. $\forall j \neq i.\ a'[j] = a[j]$ for the values that are unchanged.

This is called the **write rule**, and is an equivalence-preserving transformation on array theory formulas.

Example 7.3. The formula

$$a\{i \leftarrow e\}[i] \geq e \tag{7.8}$$

is transformed by introducing a new array identifier a' to replace $a\{i \leftarrow e\}$. Additionally, we add the constraint $a'[i] = e$, and obtain

$$a'[i] = e \implies a'[i] \geq e\ , \tag{7.9}$$

which shows the validity of (7.8). The second part of the read-over-write axiom is needed to show the validity of a formula such as

$$a[0] = 10 \implies a\{1 \leftarrow 20\}[0] = 10\ . \tag{7.10}$$

As before, the formula is transformed by replacing $a\{1 \leftarrow 20\}$ with a new identifier a' and adding the two constraints described above:

$$(a[0] = 10 \wedge a'[1] = 20 \wedge (\forall j \neq 1.\ a'[j] = a[j])) \implies a'[0] = 10\ . \tag{7.11}$$

Again as before, we transform this formula by replacing a and a' with uninterpreted-function symbols F_a and $F_{a'}$:

$$(F_a(0) = 10 \wedge F_{a'}(1) = 20 \wedge (\forall j \neq 1.\ F_{a'}(j) = F_a(j))) \implies F_{a'}(0) = 10\ .$$

▟

This simple example shows that array theory can be reduced to combinations of the index theory and uninterpreted functions, provided that the index theory offers quantifiers. The problem is that this combination is not necessarily decidable. A convenient index theory with quantifiers is Presburger arithmetic, and indeed its combination with uninterpreted functions is known to be undecidable. As mentioned above, the array theory is undecidable even if the combination of the index theory and the element theory is decidable (see Problem 7.2). We therefore need to restrict the set of formulas that we consider in order to obtain a decision procedure. This is the approach used by the reduction algorithm in the following section.

7.3 A Reduction Algorithm for a Fragment of the Array Theory

7.3.1 Array Properties

We define here a restricted class of array theory formulas in order to obtain decidability. We consider formulas that are Boolean combinations of **array properties**.

Definition 7.4 (array property). *An array theory formula is called an* array property *if and only if it is of the form*

$$\forall i_1 \ldots \forall i_k \in T_I .\ \phi_I(i_1, \ldots, i_k) \implies \phi_V(i_1, \ldots, i_k)\ , \tag{7.12}$$

$\boxed{\phi_I}$ *and satisfies the following conditions:*

$\boxed{\phi_V}$ *1. The predicate ϕ_I, called the* index guard*, must follow the grammar*

$$\begin{aligned} \mathit{iguard} &: \mathit{iguard} \wedge \mathit{iguard} \mid \mathit{iguard} \vee \mathit{iguard} \mid \mathit{iterm} \leq \mathit{iterm} \mid \mathit{iterm} = \mathit{iterm} \\ \mathit{iterm} &: i_1 \mid \ldots \mid i_k \mid \mathit{term} \\ \mathit{term} &: \textit{integer-constant} \mid \textit{integer-constant} \cdot \textit{index-identifier} \mid \mathit{term} + \mathit{term} \end{aligned}$$

The "index-identifier" used in "term" must not be one of $i_1, \ldots, i_k$.

2. The index variables $i_1, \ldots, i_k$ can only be used in array read expressions of the form $a[i_j]$.

The predicate ϕ_V is called the value constraint*.*

Example 7.5. Recall Axiom (7.4), which defines the equality of two arrays a_1 and a_2 as element-wise equality. Extensionality is an array property:

$$\forall i.\ a_1[i] = a_2[i]\ . \tag{7.13}$$

The index guard is simply TRUE in this case.

Recall the array theory formula (7.1). The first and the third conjunct are obviously array properties, but recall the second conjunct,

$$\mathsf{a}' = \mathsf{a}\{i \leftarrow 0\}\ . \tag{7.14}$$

Is this an array property as well? As illustrated in Example 7.3, an array update expression can be replaced by adding two constraints. In our example, the first constraint is $a'[i] = 0$, which is obviously an array property. The second constraint is

$$\forall j \neq i.\ a'[j] = a[j]\ , \tag{7.15}$$

which does not comply with the syntax constraints for index guards as given in Definition 7.4. However, it can be rewritten as

$$\forall j.\ (j \leq i - 1 \vee i + 1 \leq j) \implies a'[j] = a[j] \tag{7.16}$$

to match the syntactic constraints. ▟

7.3.2 The Reduction Algorithm

We now describe an algorithm that accepts a formula from the array property fragment of array theory and reduces it to an equisatisfiable formula that uses the element and index theories combined with equalities and uninterpreted functions. Techniques for uninterpreted functions are given in Chap. 4.

Algorithm 7.3.1 takes an array theory formula from the array property fragment as input. Note that the transformation of array properties to NNF may turn a universal quantification over the indices into an existential quantification. The formula does not contain explicit quantifier alternations, owing to the syntactic restrictions.

As a first step, the algorithm applies the write rule (see Sect. 7.2) to remove all array update operators. The resulting formula contains quantification over indices, array reads, and subformulas from the element and index theories.

As the formula is in NNF, an existential quantification can be replaced by a new variable (which is implicitly existentially quantified). The universal quantifiers $\forall i \in T_I .\ P(i)$ are replaced by the conjunction $\bigwedge_{i \in \mathcal{I}(\phi)} P(i)$, where the set $\mathcal{I}(\phi)$ denotes the index expressions that i might possibly be equal to in the formula ϕ. This set contains the following elements:

$\boxed{\mathcal{I}(\phi)}$

1. All expressions used as an array index in ϕ that are not quantified variables.
2. All expressions used inside index guards in ϕ that are not quantified variables.
3. If ϕ contains none of the above, $\mathcal{I}(\phi)$ is $\{0\}$ in order to obtain a nonempty set of index expressions.

Finally, the array read operators are replaced by uninterpreted functions, as described in Sect. 7.2.

Example 7.6. In order to illustrate Algorithm 7.3.1, we continue the introductory example by proving the validity of (7.1):

$$\begin{aligned} &(\forall x \in \mathbb{N}_0 .\ x < i \implies \mathsf{a}[x] = 0) \\ &\wedge\ \mathsf{a}' = \mathsf{a}\{i \leftarrow 0\} \\ \implies &(\forall x \in \mathbb{N}_0 .\ x \leq i \implies \mathsf{a}'[x] = 0)\ . \end{aligned}$$

That is, we aim to show unsatisfiability of

$$\begin{aligned} &(\forall x \in \mathbb{N}_0 .\ x < i \implies \mathsf{a}[x] = 0) \\ \wedge\ &\mathsf{a}' = \mathsf{a}\{i \leftarrow 0\} \\ \wedge\ &(\exists x \in \mathbb{N}_0 .\ x \leq i \wedge \mathsf{a}'[x] \neq 0)\ . \end{aligned} \tag{7.17}$$

By applying the write rule, we obtain

$$\begin{aligned} &(\forall x \in \mathbb{N}_0 .\ x < i \implies \mathsf{a}[x] = 0) \\ \wedge\ &\mathsf{a}'[i] = 0 \wedge \forall j \neq i .\ \mathsf{a}'[j] = \mathsf{a}[j] \\ \wedge\ &(\exists x \in \mathbb{N}_0 .\ x \leq i \wedge \mathsf{a}'[x] \neq 0)\ . \end{aligned} \tag{7.18}$$

```
Algorithm 7.3.1: Array-Reduction

Input:  An array property formula $\phi_A$ in NNF
Output: A formula $\phi_{UF}$ in the index and element theories with uninterpreted functions

1. Apply the write rule to remove all array updates from $\phi_A$.
2. Replace all existential quantifications of the form $\exists i \in T_I.\ P(i)$ by $P(j)$,
      where $j$ is a fresh variable.
3. Replace all universal quantifications of the form $\forall i \in T_I.\ P(i)$ by

      $\bigwedge_{i \in \mathcal{I}(\phi)} P(i)$ .

4. Replace the array read operators by uninterpreted functions and obtain
      $\phi_{UF}$;
5. return $\phi_{UF}$;
```

In the second step of Algorithm 7.3.1, we instantiate the existential quantifier with a new variable $z \in \mathbb{N}_0$:

$$\begin{aligned} &(\forall x \in \mathbb{N}_0.\ x < i \implies \mathsf{a}[x] = 0) \\ &\wedge \mathsf{a}'[i] = 0 \wedge \forall j \neq i.\ \mathsf{a}'[j] = \mathsf{a}[j] \\ &\wedge z \leq i \wedge \mathsf{a}'[z] \neq 0\ . \end{aligned} \tag{7.19}$$

The set $\mathcal{I}$ for our example is $\{i, z\}$. We therefore replace the two universal quantifications as follows:

$$\begin{aligned} &(i < i \implies \mathsf{a}[i] = 0) \wedge (z < i \implies \mathsf{a}[z] = 0) \\ &\wedge \mathsf{a}'[i] = 0 \wedge (i \neq i \implies \mathsf{a}'[i] = \mathsf{a}[i]) \wedge (z \neq i \implies \mathsf{a}'[z] = \mathsf{a}[z]) \\ &\wedge z \leq i \wedge \mathsf{a}'[z] \neq 0\ . \end{aligned} \tag{7.20}$$

Let us remove the trivially satisfied conjuncts to obtain

$$\begin{aligned} &(z < i \implies \mathsf{a}[z] = 0) \\ &\wedge \mathsf{a}'[i] = 0 \wedge (z \neq i \implies \mathsf{a}'[z] = \mathsf{a}[z]) \\ &\wedge z \leq i \wedge \mathsf{a}'[z] \neq 0\ . \end{aligned} \tag{7.21}$$

We now replace the two arrays a and a' by uninterpreted functions F_{a} and $F_{\mathsf{a}'}$ and obtain

$$\begin{aligned} &(z < i \implies F_{\mathsf{a}}(z) = 0) \\ &\wedge F_{\mathsf{a}'}(i) = 0 \wedge (z \neq i \implies F_{\mathsf{a}'}(z) = F_{\mathsf{a}}(z)) \\ &\wedge z \leq i \wedge F_{\mathsf{a}'}(z) \neq 0\ . \end{aligned} \tag{7.22}$$

By distinguishing the three cases $z < i$, $z = i$, and $z > i$, it is easy to see that this formula is unsatisfiable. ∎

7.4 A Lazy Encoding Procedure

7.4.1 Incremental Encoding with DPLL(T)

The reduction procedure given in the previous section performs an encoding from the array theory into the underlying index and element theories. In essence, it does so by adding instances of the read-over-write rule and the extensionality rule. In practice, most of the instances that the algorithm generates are unnecessary, which increases the computational cost of the decision problem.

In this section, we discuss a procedure that generates the instances of the read-over-write (7.3) and extensionality (7.4) rules *incrementally*, which typically results in far fewer constraints. The algorithm we describe in this section follows [70] and is designed for integration into the DPLL(T) procedure (Chap. 3). It performs a lazy encoding of the array formula into equality logic with uninterpreted functions (Chap. 4). The algorithm assumes that the index theory is quantifier-free, but does permit equalities between arrays.

Preprocessing

We perform a preprocessing step before the main phase of the algorithm. The preprocessing step instantiates the first half of (7.3) exhaustively, i.e., for all expressions $a\{i \leftarrow e\}$ present in the formula, add the constraint

$$a\{i \leftarrow e\}[i] = e \,. \tag{7.23}$$

This generates a linear number of constraints. The axiom given as (7.2) is handled using the encoding into uninterpreted functions that we have explained in the previous section. The second case of (7.3) and the extensionality rule will be implemented incrementally.

Before we discuss the details of the incremental encoding we will briefly recall the basic principle of DPLL(T), as described in Chap. 3. In DPLL(T), a propositional SAT solver is used to obtain a (possibly partial) truth assignment to the theory atoms in the formula. This assignment is passed to the theory solver, which determines whether the assignment is T-consistent. The theory solver can pass additional propositional constraints back to the SAT solver in order to implement theory propagation and theory learning. These constraints are added to the clause database maintained by the SAT solver. Afterwards, the procedure reiterates, either determining that the formula is UNSAT or generating a new (possibly partial) truth assignment.

7.4.2 Lazy Instantiation of the Read-Over-Write Axiom

Algorithm 7.4.1 takes as input a set of array formula literals (array theory atoms or their negation). The conjunction of the literals is denoted by $\hat{Th}$. The algorithm returns TRUE if $\hat{Th}$ is consistent in the array theory; otherwise, it returns a formula t that is valid in the array theory and blocks the assignment $\hat{Th}$. The formula t is initialized with TRUE, and then strengthened as the algorithm proceeds.

In line 2 the equivalence classes of the terms mentioned in $\hat{Th}$ are computed. In Sect. 4.3 we described the congruence closure algorithm for computing such classes. We denote by $t_1 \approx t_2$ the fact that terms t_1 and t_2 are in the same equivalence class.

$\boxed{\approx}$

Algorithm 7.4.1: ARRAY-ENCODING-PROCEDURE

Input: A conjunction of array literals $\hat{Th}$
Output: TRUE, or a valid array formula t that blocks $\hat{Th}$

```
 1. $t :=$ TRUE;
 2. Compute equivalence classes of terms in $\hat{Th}$;
 3. Construct the weak equivalence graph $G$ from $\hat{Th}$;
 4. for $a, b, i, j$ such that $a[i]$ and $b[j]$ are terms in $\hat{Th}$ do
 5.     if $i \approx j$ then
 6.         if $a[i] \not\approx b[j]$ then
 7.             for each simple path $p \in G$ from $a$ to $b$ do
 8.                 if each label $l$ on $p$'s edges satisfies $l \not\approx i$ then
 9.                     $t := t \wedge ((i = j \wedge Cond_i(p)) \implies a[i] = b[j])$;
10. return $t$;
```

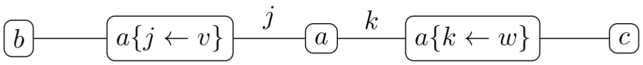
In line 3 we construct a labeled undirected graph $G(V, E)$ called the **weak equivalence graph**. The vertices V correspond to the array terms in $\hat{Th}$. The edges have an optional label, and are added as follows:

1. For each equality $a = b$ between array terms, add an unlabeled edge between a and b.
2. For each array term a and an update of that term $a\{i \leftarrow v\}$, add an edge labeled with i between their vertices.

Example 7.7. Consider the formula

$$\hat{Th} \doteq i \neq j \wedge i \neq k \wedge a\{j \leftarrow v\} = b \wedge a\{k \leftarrow w\} = c \wedge b[i] \neq c[i] . \quad (7.24)$$

The weak equivalence graph corresponding to $\hat{Th}$ is



∎

Two arrays a and b are called *weakly equivalent* whenever there is a path from a to b in G. This means that they are equal on all array elements except, possibly, those that are updated on the path. Arrays a, b, and c in the example above are all weakly equivalent.

Lines 4–9 generate constraints that enforce equality of array elements. This is relevant for any pair of array element terms $a[i]$ and $b[j]$ in $\hat{Th}$ where the index terms i and j are forced to be equal, according to the equivalence classes, but $a[i]$ and $b[j]$ are not. The idea is to determine whether the arrays a and b are connected by a chain of array updates where the index i is *not* used. If there is a chain with this property, then $a[i]$ must be equal to $b[j]$.

We will check whether this chain exists using our weak equivalence graph G as follows. We will consider all paths p from a to b. The path can be discarded if any of its edge labels has an index that is equal to i according to our equivalence classes. Otherwise, we have found the desired chain, and add

$$(i = j \wedge Cond_i(p)) \implies a[i] = b[j] \tag{7.25}$$

as a constraint to t. The expression $Cond_i(p)$ is a conjunction of the following constraints: $\boxed{Cond_i(p)}$

1. For an unlabeled edge from a to b, add the constraint $a = b$.
2. For an edge labeled with k, add the constraint $i \neq k$.

Example 7.8. Continuing Example 7.7, we have two nontrivial equivalence classes: $\{a\{j \leftarrow v\}, b\}$ and $\{a\{k \leftarrow w\}, c\}$. Hence the terms $b[i], c[i]$ satisfy $b[i] \not\approx c[i]$ and their index is trivially equal. There is one path p from b to c on the graph G, and none of its edges is labeled with an index in the same equivalence class as i, i.e., $j \not\approx i, k \not\approx i$. For this path p, we obtain

$$Cond_i(p) \;=\; i \neq j \wedge i \neq k \tag{7.26}$$

and subsequently update t in line 9 to

$$t := \;(i = i \wedge i \neq j \wedge i \neq k) \implies b[i] = c[i]\;. \tag{7.27}$$

Now t is added to (7.24). The left-hand side of t holds trivially, and thus, we obtain a contradiction to $b[i] \neq c[i]$. Hence, we proved that (7.24) is unsatisfiable. ▟

Note that the constraint returned by Algorithm 7.4.1 is TRUE when no chain is found. In this case, $\hat{Th}$ is satisfiable in the array theory. Otherwise t is a **blocking clause**, i.e., its propositional skeleton is inconsistent with the propositional skeleton of $\hat{Th}$. This ensures progress in the propositional part of the DPLL(T) procedure.

7.4.3 Lazy Instantiation of the Extensionality Rule

The constraints generated by Algorithm 7.4.1 are sufficient to imply the required equalities between individual array elements. In order to obtain a complete decision procedure for the extensional array theory, we need to add constraints that imply equalities between entire arrays.

Algorithm 7.4.2 is intended to be executed in addition to Algorithm 7.4.1. It generates further constraints that imply the equality of array terms.

Algorithm 7.4.2: EXTENSIONAL-ARRAY-ENCODING

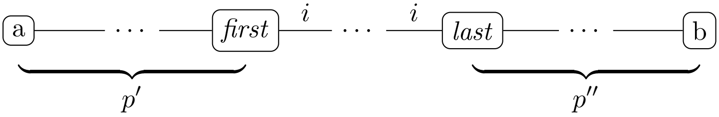
Input: A conjunction of array literals $\hat{Th}$
Output: TRUE, or a valid array formula t that blocks $\hat{Th}$

```
t := TRUE;
Compute equivalence classes of terms in $\hat{Th}$;
Construct the weak equivalence graph $G$ from $\hat{Th}$;
for $a, b$ such that $a$ and $b$ are array terms in $\hat{Th}$ do
    if $a \not\approx b$ then
        for each simple path $p \in G$ from $a$ to $b$ do
            Let $S$ be the set of edge labels of $p$;
            $t := t \wedge (\bigwedge_{i \in S} Cond_i^u(p) \implies a = b)$;
return $t$;
```

An equality between two array terms is deduced as follows: Consider all pairs a, b of array terms in $\hat{Th}$ that are not equal and any chain of equalities $\boxed{S}$ between a and b. Choose one such chain, which we call p, and let S be the set of all distinct indices that are used in array updates in the chain. For all indices $i \in S$, do the following:

1. Find the array term just *before* the first edge on p labeled with i or with an index j such that $j \approx i$. Denote this term by *first*, and denote the prefix of p up to the edge with p'.
2. Find the array term just *after* the last update on p labeled with i or with an index k such that $k \approx i$. Denote this term by *last*, and denote the suffix of the path p after this edge with p''.
3. Check that *first*$[i]$ is equal to *last*$[i]$.

If this holds for all indices, then a must be equal to b. A chain of this kind in G has the following form:



Algorithm 7.4.2 checks whether such a chain exists using our graph G as follows: It considers all paths p from a to b. For each path p it computes the set S. It then adds

$$\bigwedge_{i \in S} Cond_i^u(p) \implies a = b \tag{7.28}$$

$\boxed{Cond_i^u(p)}$ as a constraint to t, where $Cond_i^u(p)$ is defined as follows: If there is no edge with an index label that is equal to i in p, then

$$Cond_i^u(p) := \; Cond_i(p) \; .$$

Otherwise, it is the condition under which the updates of index i on p satisfy the constraints explained above, which is formalized as follows:

$$Cond_i^u(p) := \; Cond_i(p') \wedge \mathit{first}[i] = \mathit{last}[i] \wedge Cond_i(p'') \; . \tag{7.29}$$

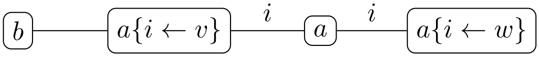
Example 7.9. Consider the following input to Algorithm 7.4.2:

$$\hat{Th} := \; v = w \quad \wedge \quad b = a\{i \leftarrow v\} \quad \wedge \quad b \neq a\{i \leftarrow w\} \; , \tag{7.30}$$

which is inconsistent. The preprocessing step (Sect. 7.4.1) is to add the instances of the first part of the read-over-write axiom (7.3). For the theory literals in $\hat{Th}$, we obtain

$$a\{i \leftarrow v\}[i] = v \quad \text{and} \quad a\{i \leftarrow w\}[i] = w \; . \tag{7.31}$$

Next, we construct the following weak equivalence graph:



Algorithm 7.4.2 will, among others, identify b and $a\{i \leftarrow w\}$ as array terms. There is one path between them, and the set S for this path is the singleton $\{i\}$. The array term *first* is $a\{i \leftarrow v\}$, and the array term *last* is $a\{i \leftarrow w\}$. Note that p' is the path from b to $a\{i \leftarrow v\}$ and that p'' is empty. We obtain

$$Cond_i^u(p) \; = \; (a\{i \leftarrow v\}[i] = a\{i \leftarrow w\}[i]) \tag{7.32}$$

and subsequently add the constraint

$$a\{i \leftarrow v\}[i] = a\{i \leftarrow w\}[i] \implies b = a\{i \leftarrow w\} \tag{7.33}$$

to our formula. Recall that we have added the constraints $a\{i \leftarrow v\}[i] = v$ and $a\{i \leftarrow w\}[i] = w$ and suppose that $v = w$ in all models of the formula. The decision procedure for equality logic will determine that $a\{i \leftarrow v\}[i] = a\{i \leftarrow w\}[i]$ holds, and thus, DPLL(T) will deduce that $b = a\{i \leftarrow w\}$ must be true in any model of the formula, which contradicts the third literal of $\hat{Th}$ in (7.30). ▟

7.5 Problems

Problem 7.1 (manual proofs for array logic). Show the validity of (7.1) using the read-over-write axiom.

Problem 7.2 (undecidability of array logic). A two-counter machine M consists of

- A finite set L of labels for instructions, which includes the two special labels *start* and *halt*
- An instruction for each label, which has one of the following two forms, where m and n are labels in L:
  - $c_i := c_i + 1$; **goto** m

  -
    ```
    if $c_i = 0$ then
        $c_i := c_i + 1;$ goto $m$
    else
        $c_i := c_i - 1;$ goto $n$
    endif
    ```

A configuration of M is a triple $\langle \ell, c_1, c_2 \rangle$ from $S := (L \times \mathbb{N} \times \mathbb{N})$, where ℓ is the label of the instruction that is to be executed next, and c_1 and c_2 are the current values of the two counters. The instructions permitted and their semantics vary. We will assume that $R(s, s')$ denotes a predicate that holds if M can make a transition from state s to state s'. The definition of R is straightforward. The initial state of M is $\langle \mathit{start}, 0, 0 \rangle$. We write $I(s)$ if s is the initial state. A computation of M is any sequence of states that begin in the initial state and where two adjacent states are related by R. We say that the machine *terminates* if there exists a computation that reaches a state in which the instruction has label *halt*. The problem of whether a given two-counter machine M terminates is undecidable in general.

Show that the satisfiability of an array logic formula is undecidable by performing a reduction of the termination problem for a two-counter machine to an array logic formula: given a two-counter machine M, generate an array logic formula φ that is valid if M terminates.

Problem 7.3 (quantifiers and NNF). The transformation steps 3 and 4 of Algorithm 7.3.1 rely on the fact that the formula is in NNF. Provide one example for each of these steps that shows that the step is unsound if the formula is not in NNF.

7.6 Bibliographic Notes

The read-over-write axiom (7.3) is due to John McCarthy, who used it to show the correctness of a compiler for arithmetic expressions [191]. The reads and writes correspond to loads and stores in a computer memory. Hoare and Wirth introduced the notation $(a, i : e)$ for $a\{i \leftarrow e\}$ and used it to define the meaning of assignments to array elements in the PASCAL programming language [145].

Automatic decision procedures for arrays have been found in automatic theorem provers since the very beginning. In the context of program verification, array logic is often combined with application-specific predicates, for example, to specify properties such as “the array is sorted” or to specify ranges of indices [241]. Greg Nelson's theorem prover SIMPLIFY [101] has McCarthy's read-over-write axiom and appropriate instantiation heuristics built in.

The reduction of array logic to fragments of Presburger arithmetic with uninterpreted functions is commonplace [272, 190, 156]. While this combination is in general undecidable [105], many restrictions of Presburger arithmetic with uninterpreted functions have been shown to be decidable. Stump et al. [269] present an algorithm that first eliminates the array update expressions from the formula by identifying matching writes. The resulting formula can be decided with an EUF decision procedure (Chap. 4). Armando et al. [6] give a decision procedure for the extensional theory of arrays based on rewriting techniques and a preprocessing phase to implement extensionality.

Most modern SMT solvers implement a variant of the incremental encoding described in Sect. 7.4. Specifically, Brummayer et al. [53] used lazy introduction of functional consistency constraints in their tool BOOLECTOR, which solves combinations of arrays and bit vectors. Such a lazy procedure was used in the past also in the context of deciding arrays via quantifier elimination [97], and in the context of translation validation [229]. The definition of *weak equivalence* and the construction of the corresponding graph are given in [70].

The array property fragment that we used in this chapter was identified by Bradley, Manna, and Sipma [44]. The idea of computing "sufficiently large" sets of instantiation values is also used in other procedures. For instance, Ghilardi et al. computed such sets separately for the indices and array elements [126]. In [127], the authors sketch how to integrate the decision procedure into a state-of-the-art SMT solver. There are also many procedures for other logics with quantifiers that are based on this approach; some of these are discussed in Sect. 9.5.

7.7 Glossary

The following symbols were used in this chapter:

Symbol	**Refers to ...**	**First used on page ...**
T_I	Index type	157
T_E	Element type	157
T_A	Array type (a map from T_I to T_E)	157
$a[i]$	The element with index i of an array a	158

continued on next page

continued from previous page

Symbol	**Refers to** ...	**First used on page** ...
$a\{i \leftarrow e\}$	The array a, where the element with index i has been replaced by e	158
ϕ_I	The index guard in an array property	162
ϕ_V	The value constraint in an array property	162
$\mathcal{I}(\phi)$	Index set	163
$t_1 \approx t_2$	The terms t_1, t_2 are in the same equivalence class	166
$Cond_i(p)$	A constraint added as part of Algorithm 7.4.1	167
S	The set of indices that are used in array updates in a path	168
$Cond_i^u(p)$	A constraint added as part of Algorithm 7.4.2	168

8

Pointer Logic

8.1 Introduction

8.1.1 Pointers and Their Applications

This chapter introduces a theory for reasoning about programs that use pointers, and describes decision procedures for it. We assume that the reader is familiar with pointers and their use in programming languages.

A **pointer** is a program variable whose sole purpose is to refer to some other program construct. This other construct could be a variable, a procedure or label, or yet another pointer. Among other things, pointers allow a piece of code to operate on different sets of data, which avoids inefficient copying of data.

As an example, consider a program that maintains two arrays of integers, named `A` and `B`, and that both arrays need to be sorted at some point within the program. Without pointers, the programmer needs to maintain two implementations of the sorting algorithm, one for `A` and one for `B`. Using pointers, a single implementation of sorting is implemented as a procedure that accepts a pointer to the first element of an array as an argument. It is called twice, with the addresses of `A` and `B`, respectively, as the argument.

As pointers are a common source of programming errors, most modern programming languages try to offer alternatives, e.g., in the form of references or abstract data containers. Nevertheless, low-level programming languages with explicit pointers are still frequently used, for example, for embedded systems or operating systems.

The implementation of pointers relies on the fact that the memory cells of a computer have *addresses*, i.e., each cell has a unique number. The value of a pointer is then nothing but such a number. The way the memory cells are addressed is captured by the concept of the **memory model** of the architecture that executes the program.

Definition 8.1 (memory model). *A* memory model *describes the assumptions that are made about the way memory cells are addressed. We assume*

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_8

$\boxed{M, A}$ $\boxed{D}$

that the architecture provides a continuous, uniform address space, i.e., the set of addresses A is a subinterval of the integers $\{0, \ldots, N-1\}$. Each address corresponds to a memory cell that is able to store one data word. The set of data words is denoted by D. A **memory valuation** *$M : A \longrightarrow D$ is a mapping from a set of addresses A into the domain D of data words.*

$\boxed{\sigma}$

A variable may require more than one data word to be stored in memory. For example, this is the case when the variable is of type struct, array, or double-precision floating point. Let $\sigma(v)$ with $v \in V$ denote the size (in data words) of v.

$\boxed{V}$

The compiler assigns a particular memory location (address) to each global, and thus, static variable.[1] This mapping is called the **memory layout**, and is formalized as follows. Let V denote the set of variables.

$\boxed{L}$

Definition 8.2 (memory layout). *A* memory layout *$L : V \longrightarrow A$ is a mapping from each variable $v \in V$ to an address $a \in A$. The address of v is also called the* memory location *of v.*

The memory locations of the statically allocated variables are usually assigned such that they are *nonoverlapping* (we explain later on how to model dynamically allocated data structures). Note that the memory layout is not necessarily continuous, i.e., compilers may generate a layout that contains "holes".[2]

Example 8.3. Figure 8.1 illustrates a memory layout for a fragment of an ANSI-C program. The program has six objects, which are named `var_a`, `var_b`, `var_c`, `S`, `array`, and `p`. The first five objects either are integer variables or are composed of integer variables. The object named `p` is a pointer variable, which we assume to be as wide as an integer.[3] The program initializes `p` to the address of the variable `var_c`, which is denoted by `&var_c`. Besides the variable definitions, the program also has a function `main()`, which sets the value of the variable pointed to by `p` to 100. ▟

8.1.2 Dynamic Memory Allocation

Pointers also enable the creation of *dynamic data structures*. Dynamic data structures rely on an area of memory that is designated for use by objects that

[1] Statically allocated variables are variables that are allocated space during the entire run time of the program. In contrast, the addresses of dynamically allocated data such as local variables or data on the heap are determined at run time once the object has been created.

[2] A possible reason for such holes is the need for proper alignment. As an example, many 64-bit architectures are unable to read double-precision floating-point values from addresses that are not a multiple of 8.

[3] This is not always the case; for example, in the x86 16-bit architecture, integers have 16 bits, whereas pointers are 32 bits wide. In some 64-bit architectures, integers have 32 bits, whereas pointers have 64 bits.

```
int var_a, var_b, var_c;
struct { int x; int y; } S;
int array[4];
int *p = &var_c;

int main() {
  *p=100;
}
```

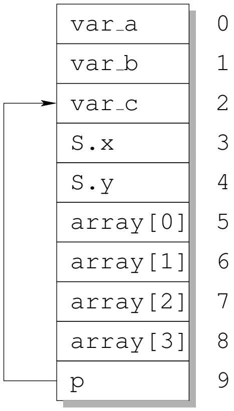


Fig. 8.1. A fragment of an ANSI-C program and a possible memory layout for it

Aside: Pointers and References in Object-Oriented Programming
Separation of data and algorithms is promoted by the concept of *object-oriented programming* (OOP). In modern programming languages such as Java and C++, the explicit use of pointer variables is deprecated. Instead, the procedures that are associated with an object (the *methods*) implicitly receive a pointer to the data members (the *fields*) of the object instance as an argument. In C++, the pointer is accessible using the keyword `this`. All accesses to the data members are performed indirectly by means of the `this` pointer variable.

References, just like pointers, are program variables that refer to a variable or object. The difference between references and pointers is often only syntactic. As an example, the fact that dereferencing is performed is usually hidden. In program analysis, references can be treated just as pointers.

are created at the run time of the program. A run time library maintains a list of the memory regions that are unused. A function, which is part of this library, allocates a region of given size and returns a pointer to the beginning (lowest address) of the region. The memory layout therefore *changes during the run time of the program*. Memory allocation may be performed an unbounded number of times (provided enough space is deallocated as well), and thus, there is no bound on the number of objects that a program can generate.

The function that performs the allocation is called `malloc()` in C, and is provided as an operator called `new` in C++, C#, and Java. In either case, the size of the region that is requested is passed as an argument. In order to reuse memory occupied by data structures that are no longer needed, C programmers call `free`, C++ programmers use `delete`, while Java and C# provide an automatic garbage collection mechanism. The **lifetime** of a dynamic object is the time between its allocation and its deallocation.

8.1.3 Analysis of Programs with Pointers

All but trivial programs rely on pointers or references in order to separate between data and algorithms. Decision procedures that are used for program analysis therefore often need to include reasoning about pointers.

As a simple example, consider the following program fragment, which computes the sum of an array of size 10:

```
void f(int *sum) {
  *sum = 0;

  for(i=0; i<10; i++)
    *sum = *sum + array[i];
}
```

The sum is stored in an integer variable that is pointed to by a pointer called `sum`. Any analysis method that aims at validating the correctness of this fragment has to take the value of the pointer into account. In particular, the program is likely to fail if the address held by `sum` is equal to the address of `i`. In this case, we say that `*sum` is an **alias** for `i`. Aliasing that is not anticipated by the programmer is a common source of problems.

The use of pointers gives rise to program properties that are of high interest. It is well known that many programs fail owing to incorrect use of pointer variables. A very common problem in programs is dereferencing of pointer variables that do not point to a proper object. The value 0 is typically reserved as a designated `NULL` pointer. It is guaranteed that no object, either statically or dynamically allocated, has this address. This value can therefore be used to indicate special cases, for example, the end of a linked list. However, if such a pointer is—by mistake—dereferenced, modern architectures typically generate an exception, which terminates the program.

Programming languages that offer explicit deallocation face another problem. In the following program fragment, an array-type object is allocated and deallocated:

```
int *p, *q;

p = new int[10];
q = &p[3];
delete p;
*q = 2;
```

Note that the address of the fourth element of the array is stored in `q`, and that this pointer is dereferenced after the deallocation of the array. In a variant of the program above, the library that manages the dynamically allocated memory may have reassigned the space used for the array by that time, and thus another object might be overwritten by writing to `*q`. Such errors are hard to reproduce, as they depend on the exact memory layout of the architecture.

They often remain undetected despite extensive testing. The detection of such errors is therefore an important application for static program analysis tools.

Aside: Alias Analysis

Alias analysis has a significant role in pointer-related reasoning about software, such as the analysis performed by optimizing compilers. Alias analysis may be performed at various levels of precision. For example, alias analysis may be field sensitive or insensitive, interprocedural or intraprocedural, and may or may not be sensitive to the control flow. Alias analysis is a special case of *static analysis*, and is typically performed as a *may-analysis*, that is, it determines the set of variables that a given pointer *may* point to — this is called the "points-to" set. In other words, variables that are *not* in this set cannot be pointed to by this pointer. For example, given an instruction such as

```
*p=0;
```

may-analysis permits us to conclude that any variable that is *not* in the points-to set of `p` is also not modified by this assignment. In the case of an optimizing compiler, this permits us to determine the set of variables that can be cached safely in processor registers.

Alias analysis is performed by maintaining a points-to set for each pointer (and, if desired, for each program location), and updating these sets according to the program statements. The algorithm terminates once the sets have saturated, i.e., do not change anymore.

As an example, consider a control-flow-insensitive analysis of a program with three statements:

```
p=q;
q=&i;
p=&j;
```

The points-to sets of `p` and `q` are initially empty. Processing the first statement results in no change. The second statement adds `i` to the points-to set of `q`, and the third adds `j` to the points-to set of `p`. Owing to the first statement, the set of `q` is added to that of `p` and, thereafter, the two sets are saturated.

8.2 A Simple Pointer Logic

8.2.1 Syntax

There are many variants of pointer logic, each with a different syntax and meaning. The more complex ones are often undecidable. We define a simple logic here, with the goal of making the problem of deciding formulas in this logic easier to solve.

Definition 8.4 (pointer logic). *The syntax of a formula in* pointer logic *is defined by the following rules:*

$$\begin{aligned}
\mathit{formula} &: \mathit{formula} \wedge \mathit{formula} \mid \neg\mathit{formula} \mid (\mathit{formula}) \mid \mathit{atom} \\
\mathit{atom} &: \mathit{pointer} \;=\; \mathit{pointer} \mid \mathit{term} \;=\; \mathit{term} \mid \\
& \mathit{pointer} \;<\; \mathit{pointer} \mid \mathit{term} \;<\; \mathit{term} \\
\mathit{pointer} &: \mathit{pointer\text{-}identifier} \mid \mathit{pointer} + \mathit{term} \mid (\mathit{pointer}) \mid \\
& \&\mathit{identifier} \mid \& * \mathit{pointer} \mid * \mathit{pointer} \mid \mathrm{NULL} \\
\mathit{term} &: \mathit{identifier} \mid * \mathit{pointer} \mid \mathit{term}\ \mathit{op}\ \mathit{term} \mid (\mathit{term}) \mid \\
& \mathit{integer\text{-}constant} \mid \mathit{identifier}\ [\ \mathit{term}\ ] \\
\mathit{op} &: + \mid -
\end{aligned}$$

The variables represented by *pointer-identifier* are assumed to be of pointer type, whereas the variables represented by *identifier* are assumed to be integers or an array of integers.[4] Note that the grammar allows pointer arithmetic, whereas it prohibits a direct conversion of an integer into a pointer or vice versa. This is motivated by the fact that the conversion of a pointer to an integer may actually fail in a number of architectures, owing to the fact that pointers are wider than the standard integer type.[5]

Example 8.5. Let p, q denote pointer identifiers, and let i, j denote integer identifiers. The following expressions are well formed according to the grammar above:

- $*(p+i) = 1$,
- $*(p+*p) = 0$,
- $p = q \wedge *p = 5$,
- $*****p = 1$,
- $p < q$.

The following expressions are not permitted by the grammar:

- $p+i$,
- $p = i$,
- $*(p+q)$,
- $*1 = 1$,
- $p < i$. ▟

Note that the grammar above encompasses all of integer linear arithmetic (Chap. 5) and also a fragment of array logic (Chap. 7). In practice, a logic for pointers is typically combined with a logic for the program expressions, such as bit-vector arithmetic.

[4] The syntax is clearly inspired by that of ANSI-C. Note, however, that we deviate from the ANSI-C syntax in a few points. As an example, in ANSI-C, an array identifier is synonymous with its address.

[5] Much as in C/C++, an indirect conversion by means of the dereferencing operator is still possible.

8.2.2 Semantics

There are numerous ways to assign a meaning to the expressions defined above. We define the semantics by referring to a specific memory layout L (Definition 8.2) and a specific memory valuation M (Definition 8.1), that is, pointer logic formulas are predicates on M, L pairs. The definition uses a reduction to integer arithmetic and array logic, and thus we treat M and L as array types. We also assume that D (the set of data words) is contained in the set of integers.

Definition 8.6 (semantics of pointer logic). *As before let L denote a memory layout and let M denote a valuation of the memory. Let $\mathcal{L}_P$ denote the set of pointer logic expressions, and let $\mathcal{L}_D$ denote the set of expressions permitted by the logic for the data words. We define a meaning for $e \in \mathcal{L}_P$ using the function $\llbracket \cdot \rrbracket : \mathcal{L}_P \longrightarrow \mathcal{L}_D$. The function $\llbracket e \rrbracket$ is defined recursively as given in Fig. 8.2. The expression $e \in \mathcal{L}_P$ is valid if and only if $\llbracket e \rrbracket$ is valid.*

$$\begin{array}{rcll}
\llbracket f_1 \wedge f_2 \rrbracket & \doteq & \llbracket f_1 \rrbracket \wedge \llbracket f_2 \rrbracket & \\
\llbracket \neg f \rrbracket & \doteq & \neg \llbracket f \rrbracket & \\
\llbracket p_1 = p_2 \rrbracket & \doteq & \llbracket p_1 \rrbracket = \llbracket p_2 \rrbracket & \text{where } p_1,\, p_2 \text{ are pointer expressions} \\
\llbracket p_1 < p_2 \rrbracket & \doteq & \llbracket p_1 \rrbracket < \llbracket p_2 \rrbracket & \text{where } p_1,\, p_2 \text{ are pointer expressions} \\
\llbracket t_1 = t_2 \rrbracket & \doteq & \llbracket t_1 \rrbracket = \llbracket t_2 \rrbracket & \text{where } t_1,\, t_2 \text{ are terms} \\
\llbracket t_1 < t_2 \rrbracket & \doteq & \llbracket t_1 \rrbracket < \llbracket t_2 \rrbracket & \text{where } t_1,\, t_2 \text{ are terms} \\
\llbracket p \rrbracket & \doteq & M[L[p]] & \text{where } p \text{ is a pointer identifier} \\
\llbracket p + t \rrbracket & \doteq & \llbracket p \rrbracket + \llbracket t \rrbracket & \text{where } p \text{ is a pointer expression, and } t \text{ is a term} \\
\llbracket \& v \rrbracket & \doteq & L[v] & \text{where } v \in V \text{ is a variable} \\
\llbracket \& * p \rrbracket & \doteq & \llbracket p \rrbracket & \text{where } p \text{ is a pointer expression} \\
\llbracket \mathrm{NULL} \rrbracket & \doteq & 0 & \\
\llbracket v \rrbracket & \doteq & M[L[v]] & \text{where } v \in V \text{ is a variable} \\
\llbracket *p \rrbracket & \doteq & M[\llbracket p \rrbracket] & \text{where } p \text{ is a pointer expression} \\
\llbracket t_1 \mathbin{op} t_2 \rrbracket & \doteq & \llbracket t_1 \rrbracket \mathbin{op} \llbracket t_2 \rrbracket & \text{where } t_1,\, t_2 \text{ are terms} \\
\llbracket c \rrbracket & \doteq & c & \text{where } c \text{ is an integer constant} \\
\llbracket v[t] \rrbracket & \doteq & M[L[v] + \llbracket t \rrbracket] & \text{where } v \text{ is an array identifier, and } t \text{ is a term}
\end{array}$$

Fig. 8.2. Semantics of pointer expressions

Observe that a pointer p points to a variable x if $M[L[p]] = L[x]$, that is, the value of p is equal to the address of x. As a shorthand, we write $p \hookrightarrow z$ to mean that p points to some memory cell such that $*p = z$. Observe also that the meaning of pointer arithmetic, for example, $p + i$, does not depend on the type of object that p points to.[6]

$\boxed{p \hookrightarrow z}$

[6] In contrast, the semantics of ANSI-C requires that an integer that is added to a pointer p is multiplied by the size of the type that p points to.

Example 8.7. Consider the following expression, where a is an array identifier:

$$*\,((\&a)+1) \;=\; a[1]\;. \tag{8.1}$$

The semantic definition of (8.1) expands as follows:

$$\begin{aligned}
[\![*((\&a)+1) \;=\; a[1]]\!] &\iff [\![*((\&a)+1)]\!] \;=\; [\![a[1]]\!] && (8.2)\\
&\iff M[[\![(\&a)+1]\!]] \;=\; M[L[a]+[\![1]\!]] && (8.3)\\
&\iff M[[\![\&a]\!]+[\![1]\!]] \;=\; M[L[a]+1] && (8.4)\\
&\iff M[L[a]+1] \;=\; M[L[a]+1] && (8.5)
\end{aligned}$$

Equation (8.5) is obviously valid, and thus, so is (8.1). Note that the translated formula must evaluate to TRUE for any L and M and, thus, the following formula is not valid:

$$*\,p = 1 \implies x = 1\;. \tag{8.6}$$

For $p \neq \&x$, this formula evaluates to FALSE. ▟

8.2.3 Axiomatization of the Memory Model

Formulas in pointer logic may exploit assumptions made about the memory model. The set of these assumptions depends highly on the architecture. Here, we formalize properties that most architectures comply with, and thus that many programs rely on.

On most architectures, the following two formulas are valid, and hence can be safely assumed by programmers:

$$\&x \neq \text{NULL}\;, \tag{8.7}$$

$$\&x \neq \&y\;. \tag{8.8}$$

Equation (8.7) translates into $L[x] \neq 0$ and relies on the fact that no object has address 0. Equation (8.8) relies on the fact that the memory layout assigns nonoverlapping addresses to the objects. We define a series of *memory model axioms* in order to formalize these properties.

Memory Model Axiom 1 ("No object has address 0") *The fact "no object has address 0" is easily formalized:*[7]

$$\forall v \in V.\; L[v] \neq 0\;. \tag{8.9}$$

[7] Note that the ANSI-C standard does not actually guarantee that the symbolic constant NULL is represented by a bit vector consisting of zeros; however, it guarantees that the NULL pointer compares to the integer zero and can be obtained by converting the integer zero to a pointer type.

The easiest way to ensure that (8.8) is valid is to assume that $\forall v_1, v_2 \in V.\ v_1 \neq v_2 \implies L[v_1] \neq L[v_2]$. However, this assumption is often not strong enough, as objects with size greater or equal to two may still overlap. We therefore assume the following two conditions, which together are stronger:

Memory Model Axiom 2 ("Objects have size at least one") *The fact "an object has size at least one" is easily captured by*

$$\forall v \in V.\ \sigma(v) \geq 1\ . \tag{8.10}$$

Memory Model Axiom 3 ("Objects do not overlap") *Different objects do not share any addresses:*

$$\begin{aligned} \forall v_1, v_2 \in V.\ v_1 \neq v_2 \implies\ & \{L[v_1], \ldots, L[v_1] + \sigma(v_1) - 1\} \cap \\ & \{L[v_2], \ldots, L[v_2] + \sigma(v_2) - 1\} = \emptyset\ . \end{aligned} \tag{8.11}$$

Program analysis tools that are applied to code that relies on additional, architecture-specific guarantees may require a larger set of memory model axioms. Examples are *byte ordering* and *endianness*, and specific assumptions about *alignment*.

8.2.4 Adding Structure Types

Structure types are a convenient way to implement data structures. Structure types can be added to our pointer logic as a purely syntactic extension, as we shall soon see. We assume that the fields of the structure types are named, and write $s.f$ to denote the value of the field f in the structure s.

Formally, we can view structure types as "syntactic sugar" for array types, and record the following shorthands. Each field of the structure is assigned a unique **offset**. Let $o(f)$ denote the offset of field f. We then define the meaning of $s.f$ as follows:

$\boxed{o(f)}$

$\boxed{s.f}$

$$s.f \quad \doteq \quad *((\&s) + o(f))\ . \tag{8.12}$$

For convenience, we introduce two additional shorthands. Following the PASCAL and ANSI-C syntax, we write $p\text{->}f$ for $(*p).f$ (this shorthand is not to be confused with logical implication or with $p \hookrightarrow a$). Adopting some notation from separation logic (see the aside on separation logic on p. 184), we also extend the $p \hookrightarrow a$ notation by introducing $p \hookrightarrow a, b, c, \ldots$ as a shorthand for

$\boxed{p\text{->}f}$

$$\begin{aligned} &*(p + 0) = a\ \wedge \\ &*(p + 1) = b\ \wedge \\ &*(p + 2) = c\ \ldots\ . \end{aligned} \tag{8.13}$$

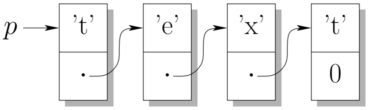
8.3 Modeling Heap-Allocated Data Structures

8.3.1 Lists

Heap-allocated data structures play an important role in programs, and are prone to pointer-related errors. We now illustrate how to model a number of commonly used data structures using pointer logic.

After the array, the simplest dynamically allocated data structure is the *linked list*. It is typically realized by means of a structure type that contains fields for a *next pointer* and the data that are to be stored in the list.

As an example, consider the following list: The first field is named a and is an ASCII character, serving as the "payload", and the second field is named n, and is the pointer to the next element of the list. Following ANSI-C syntax, we use 'x' to denote the integer that represents the ASCII character "x":



The list is terminated by a NULL pointer, which is denoted by "0" in the diagram above. A way of modeling this list is to use the following formula:

$$\begin{aligned} & p \hookrightarrow \text{'t'}, p_1 \\ \wedge \; & p_1 \hookrightarrow \text{'e'}, p_2 \\ \wedge \; & p_2 \hookrightarrow \text{'x'}, p_3 \\ \wedge \; & p_3 \hookrightarrow \text{'t'}, \text{NULL} \; . \end{aligned} \tag{8.14}$$

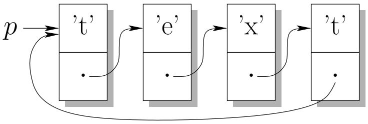
This way of specifying lists is cumbersome, however. Therefore, disregarding the payload field, we first introduce a recursive shorthand for the i-th member of a list:[8]

$$\begin{aligned} \textsf{list-elem}(p,0) &\doteq p \; , \\ \textsf{list-elem}(p,i) &\doteq \textsf{list-elem}(p,i-1)\text{->}n \quad \text{for } i \geq 1 \; . \end{aligned} \tag{8.15}$$

$\boxed{\textsf{list}}$ We now define the shorthand $\textsf{list}(p,l)$ to denote a predicate that is true if p points to a NULL-terminated acyclic list of length l:

$$\textsf{list}(p,l) \doteq \textsf{list-elem}(p,l) = \text{NULL} \; . \tag{8.16}$$

A linked list is *cyclic* if the pointer of the last element points to the first one:

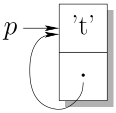


Consider the following variant $\textsf{my-list}(p,l)$, intended to capture the fact that p points to such a cyclic list of length $l \geq 1$:

[8] Note that recursive definitions of this form are, in general, only embeddable into our pointer logic if the second argument is a constant.

$$\mathsf{my\text{-}list}(p,l) \doteq \mathsf{list\text{-}elem}(p,l) = p\ . \tag{8.17}$$

Does this definition capture the concept properly? The list in the diagram above satisfies $\mathsf{my\text{-}list}(p,4)$. Unfortunately, the following list satisfies $\mathsf{my\text{-}list}(p,4)$ just as well:



This is due to the fact that our definition does not preclude *sharing* of elements of the list, despite the fact that we had certainly intended to specify that there are l disjoint list elements. Properties of this kind are often referred to as *separation properties*. A way to assert that the list elements are disjoint is to define a shorthand $\mathsf{overlap}$ as follows:

$$\mathsf{overlap}(p,q) \doteq\ p = q \lor p+1 = q \lor p = q+1\ . \tag{8.18}$$

This shorthand is then used to state that all list elements are pairwise disjoint:

$$\begin{aligned} \mathsf{list\text{-}disjoint}(p,0) &\doteq\ \text{TRUE}\ ,\\ \mathsf{list\text{-}disjoint}(p,l) &\doteq\ \mathsf{list\text{-}disjoint}(p,l-1) \land \\ \forall 0 \le i < l-1.\ \neg&\mathsf{overlap}(\mathsf{list\text{-}elem}(p,i), \mathsf{list\text{-}elem}(p,l-1))\ . \end{aligned} \tag{8.19}$$

Note that the size of this formula grows quadratically in l. As separation properties are frequently needed, more concise notations have been developed for this concept, for example, *separation logic* (see the aside on that topic). Separation logic can express such properties with formulas of linear size.

8.3.2 Trees

We can implement a *binary tree* by adding another pointer field to each element of the data structure (see Fig. 8.3). We denote the pointer to the left-hand child node by l, and the pointer to the right-hand child by r.

In order to illustrate a pointer logic formula for trees, consider the tree in Fig. 8.3, which has one integer x as payload. Observe that the integers are arranged in a particular fashion: the integer of the left-hand child of any node n is always smaller than the integer of the node n itself, whereas the integer of the right-hand child of node n is always larger than the integer of the node n. This property permits lookup of elements with a given integer value in time $O(h)$, where h is the height of the tree. The property can be formalized as follows:

$$\begin{aligned} &(n.l \neq \text{NULL} \implies n.l\text{->}x < n.x) \\ \land\ &(n.r \neq \text{NULL} \implies n.r\text{->}x > n.x)\ . \end{aligned} \tag{8.22}$$

Unfortunately, (8.22) is not strong enough to imply lookup in time $O(h)$. For this, we need to establish the ordering over the integers of an *entire subtree*.

Aside: Separation Logic

Theories for dynamic data structures are frequently used for proving that memory cells *do not alias*. While it is possible to model the statement that a given object does not alias with other objects with pairwise comparison, reasoning about such formulation scales poorly. It requires enumeration of all heap-allocated objects, which makes it difficult to reason about a program in a local manner.

John Reynolds' *separation logic* [242] addresses both problems by introducing a new binary operator "$*$", as in "$P * Q$", which is called a *separating conjunction*. The meaning of $*$ is similar to the standard Boolean conjunction, i.e., $P \wedge Q$, but it also asserts that P and Q reason about separate, nonoverlapping portions of the heap. As an example, consider the following variant of the list predicate:

$$\begin{aligned} \mathsf{list}(p,0) &\doteq p = \text{NULL} \\ \mathsf{list}(p,l) &\doteq \exists q.\ p \hookrightarrow z, q \wedge \mathsf{list}(q, l-1) \quad \text{for } l \geq 1\ . \end{aligned} \tag{8.20}$$

Like our previous definition, the definition above suffers from the fact that some memory cells of the elements of the list might overlap. This can be mended by replacing the standard conjunction in the definition above by a separating conjunction:

$$\mathsf{list}(p,l) \doteq \exists q.\ p \hookrightarrow z, q * \mathsf{list}(q, l-1)\ . \tag{8.21}$$

This new list predicate also asserts that the memory cells of all list elements are pairwise disjoint. Separation logic, in its generic form, is not decidable, but a variety of decidable fragments have been identified.

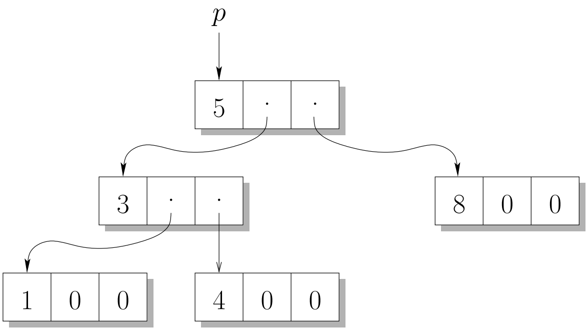


Fig. 8.3. A binary tree that represents a set of integers

We define a predicate $\mathsf{tree\text{-}reach}(p, q)$, which holds if q is reachable from p in one step:

$$\begin{aligned}\mathsf{tree\text{-}reach}(p, q) \doteq\ & p \neq \mathrm{NULL} \wedge q \neq \mathrm{NULL} \wedge \\ & (p = q \vee p{\text{->}}l = q \vee p{\text{->}}r = q)\ .\end{aligned} \tag{8.23}$$

In order to obtain a predicate that holds if and only if q is reachable from p in any number of steps, we define the *transitive closure* of a given binary relation R.

Definition 8.8 (transitive closure). *Given a binary relation R, the* transitive closure TC_R *relates x and y if there are $z_1, z_2, \ldots, z_n$ such that*

$$xRz_1 \ \wedge\ z_1Rz_2 \ \wedge\ \ldots\ \wedge\ z_nRy\ .$$

Formally, transitive closure can be defined inductively as follows:

$$\begin{aligned}\mathsf{TC}^1_R(p, q) &\doteq R(p, q)\ , \\ \mathsf{TC}^i_R(p, q) &\doteq \exists p'.\ \mathsf{TC}^{i-1}_R(p, p') \wedge R(p', q)\ , \\ \mathsf{TC}(p, q) &\doteq \exists i.\ \mathsf{TC}^i_R(p, q)\ .\end{aligned} \tag{8.24}$$

Using the transitive closure of our $\mathsf{tree\text{-}reach}$ relation, we obtain a new relation $\mathsf{tree\text{-}reach^*}(p, q)$ that holds if and only if q is reachable from p in any number of steps:

$$\mathsf{tree\text{-}reach^*}(p, q) \iff \mathsf{TC}_{\mathsf{tree\text{-}reach}}(p, q)\ . \tag{8.25}$$

Using $\mathsf{tree\text{-}reach^*}$, it is easy to strengthen (8.22) appropriately:

$$\begin{aligned}&(\forall p.\ \mathsf{tree\text{-}reach^*}(n.l, p) \implies p{\text{->}}x < n.x) \\ \wedge\ &(\forall p.\ \mathsf{tree\text{-}reach^*}(n.r, p) \implies p{\text{->}}x > n.x)\ .\end{aligned} \tag{8.26}$$

Unfortunately, the addition of the transitive closure operator can make even simple logics undecidable, and thus, while convenient for modeling, it is a burden for automated reasoning. We restrict the presentation below to decidable cases by considering only special cases.

8.4 A Decision Procedure

8.4.1 Applying the Semantic Translation

The semantic translation introduced in Sect. 8.2.2 not only assigns meaning to the pointer formulas, but also gives rise to a simple decision procedure. The formulas generated by this semantic translation contain array read operators and linear arithmetic over the type that is used for the indices. This may be the set of integers (Chap. 5) or the set of bit vectors (Chap. 6). It also contains at least equalities over the type that is used to model the contents of the memory cells. We assume that this is the same type as the index type. As we have seen in Chap. 7, such a logic is decidable. Care has to be taken when extending the pointer logic with quantification, as array logic with arbitrary quantification is undecidable.

A straightforward decision procedure for pointer logic therefore first applies the semantic translation to a pointer formula φ to obtain a formula φ' in the combined logic of linear arithmetic over integers and arrays of integers. The formula φ' is then passed to the decision procedure for the combined logic. As the formulas φ and φ' are equisatisfiable (by definition), the result returned for φ' is also the correct result for φ.

Example 8.9. Consider the following pointer logic formula, where x is a variable, and p identifies a pointer:

$$p = \&x \wedge x = 1 \implies *p = 1 \ . \tag{8.27}$$

The semantic definition of this formula expands as follows:

$$\begin{aligned}
&\llbracket p = \&x \wedge x = 1 \implies *p = 1 \rrbracket \\
&\quad \iff \llbracket p = \&x \rrbracket \wedge \llbracket x = 1 \rrbracket \implies \llbracket *p = 1 \rrbracket \\
&\quad \iff \llbracket p \rrbracket = \llbracket \&x \rrbracket \wedge \llbracket x \rrbracket = 1 \implies \llbracket *p \rrbracket = 1 \\
&\quad \iff M[L[p]] = L[x] \wedge M[L[x]] = 1 \implies M[M[L[p]]] = 1 \ .
\end{aligned} \tag{8.28}$$

A decision procedure for array logic and equality logic easily concludes that the formula above is valid, and thus, so is (8.27).

As an example of an invalid formula, consider

$$p \hookrightarrow x \implies p = \&x \ . \tag{8.29}$$

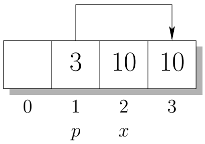
The semantic definition of this formula expands as follows:

$$\begin{aligned}
&\llbracket p \hookrightarrow x \implies p = \&x \rrbracket \\
&\quad \iff \llbracket p \hookrightarrow x \rrbracket \implies \llbracket p = \&x \rrbracket \\
&\quad \iff \llbracket *p = x \rrbracket \implies \llbracket p \rrbracket = \llbracket \&x \rrbracket \\
&\quad \iff \llbracket *p \rrbracket = \llbracket x \rrbracket \implies M[L[p]] = L[x] \\
&\quad \iff M[M[L[p]]] = M[L[x]] \implies M[L[p]] = L[x]
\end{aligned} \tag{8.30}$$

A counterexample to this formula is the following:

$$L[p] = 1,\ L[x] = 2,\ M[1] = 3,\ M[2] = 10,\ M[3] = 10 \ . \tag{8.31}$$

The values of M and L in the counterexample are best illustrated with a picture:



Applying the Memory Model Axioms

A formula may rely on one of the memory model axioms defined in Sect. 8.2.3. As an example, consider the following formula:

$$\sigma(x) = 2 \implies \&y \neq \&x + 1\ . \tag{8.32}$$

The semantic translation yields

$$\sigma(x) = 2 \implies L[y] \neq L[x] + 1\ . \tag{8.33}$$

This formula can be shown to be valid by instantiating Memory Model Axiom 3. After instantiating v_1 with x and v_2 with y, we obtain

$$\{L[x], \ldots, L[x] + \sigma(x) - 1\} \cap \{L[y], \ldots, L[y] + \sigma(y) - 1\} = \emptyset\ . \tag{8.34}$$

We can transform the set expressions in (8.34) into linear arithmetic over the integers as follows:

$$(L[x] + \sigma(x) - 1 < L[y]) \vee (L[x] > L[y] + \sigma(y) - 1)\ . \tag{8.35}$$

Using $\sigma(x) = 2$ and $\sigma(y) \geq 1$ (Memory Model Axiom 2), we can conclude, furthermore, that

$$(L[x] + 1 < L[y]) \vee (L[x] > L[y])\ . \tag{8.36}$$

Equation (8.36) is strong enough to imply $L[y] \neq L[x] + 1$, which proves that Eq. (8.32) is valid.

8.4.2 Pure Variables

The semantic translation of a pointer formula results in a formula that we can decide using the procedures described in this book. However, semantic translation down to memory valuations places an undue burden on the underlying decision procedure, as illustrated by the following example (symmetry of equality):

$$\begin{aligned} & [\![x = y \implies y = x]\!] & (8.37)\\ \iff & [\![x = y]\!] \implies [\![y = x]\!] & (8.38)\\ \iff & M[L[x]] = M[L[y]] \implies M[L[y]] = M[L[x]]\ . & (8.39) \end{aligned}$$

A decision procedure for array logic and equality logic is certainly able to deduce that (8.39) is valid. Nevertheless, the steps required for solving (8.39) obviously exceed the effort required to decide

$$x = y \implies y = x\ . \tag{8.40}$$

In particular, the semantic translation does not exploit the fact that x and y do not actually interact with any pointers. A straightforward optimization is therefore the following: if the address of a variable x is not referred to, we translate it to a new variable Υ_x instead of $M[L[x]]$. A formalization of this idea requires the following definition:

Definition 8.10 (pure variables). *Given a formula φ with a set of variables V, let $\mathcal{P}(\varphi) \subseteq V$ denote the subset of φ's variables that are not used within an argument of the "&" operator within φ. These variables are called* pure.

As an example, $\mathcal{P}(\&x = y)$ is $\{y\}$. We now define a new translation function $[\![\cdot]\!]^{\mathcal{P}}$. The definition of $[\![e]\!]^{\mathcal{P}}$ is identical to the definition of $[\![e]\!]$ unless e denotes a variable in $\mathcal{P}(\varphi)$. The new definition is:

$$\begin{aligned} [\![v]\!]^{\mathcal{P}} &\doteq \Upsilon_v && \text{for } v \in \mathcal{P}(\varphi) \\ [\![v]\!]^{\mathcal{P}} &\doteq M[L[v]] && \text{for } v \in V \setminus \mathcal{P}(\varphi) \end{aligned}$$

Theorem 8.11. *The translation using pure variables is equisatisfiable with the semantic translation:*

$$[\![\varphi]\!]^{\mathcal{P}} \iff [\![\varphi]\!] .$$

Example 8.12. Equation (8.38) is now translated as follows without referring to a memory valuation, and thus no longer burdens the decision procedure for array logic:

$$[\![x = y \implies y = x]\!]^{\mathcal{P}} \tag{8.41}$$

$$\iff [\![x = y \implies y = x]\!]^{\mathcal{P}} \tag{8.42}$$

$$\iff [\![x = y]\!]^{\mathcal{P}} \implies [\![y = x]\!]^{\mathcal{P}} \tag{8.43}$$

$$\iff \Upsilon_x = \Upsilon_y \implies \Upsilon_y = \Upsilon_x . \tag{8.44}$$

▟

8.4.3 Partitioning the Memory

The translation procedure can be optimized further using the following observation: the run time of a decision procedure for array logic depends on the number of different expressions that are used to index a particular array (see Chap. 7). As an example, consider the pointer logic formula

$$*\, p = 1 \wedge *q = 1 , \tag{8.45}$$

which—using our optimized translation—is reduced to

$$M[\Upsilon_p] = 1 \wedge M[\Upsilon_q] = 1 . \tag{8.46}$$

The pointers p and q might alias, but there is no reason why they have to. Without loss of generality, we can therefore safely assume that they do not alias and, thus, we partition M into M_1 and M_2:

$$M_1[\Upsilon_p] = 1 \wedge M_2[\Upsilon_q] = 1 . \tag{8.47}$$

While this has increased the number of array variables, the number of different indices *per array* has decreased. Typically, this improves the performance of a decision procedure for array logic.

This transformation cannot always be applied, illustrated by the following example:

$$p = q \implies *p = *q \; . \tag{8.48}$$

This formula is obviously valid, but if we partition as before, the translated formula is no longer valid:

$$\Upsilon_p = \Upsilon_q \implies M_1[\Upsilon_p] = M_2[\Upsilon_q] \; . \tag{8.49}$$

Unfortunately, deciding if the optimization is applicable is in general as hard as deciding φ itself. We therefore settle for an approximation based on a syntactic test. This approximation is conservative, i.e., sound, while it may not result in the best partitioning that is possible in theory.

Definition 8.13. *We say that two pointer expressions p and q are* related *directly by a formula φ if both p and q are used inside the same relational expression in φ and that the expressions are related transitively if there is a pointer expression p' that relates to p and relates to q. We write $p \approx q$ if p and q are related directly or transitively.* $\boxed{p \approx q}$

The relation $\approx$ induces a partitioning of the pointer expressions in φ. We number these partitions $1, \ldots, n$. Let $I(p) \in \{1, \ldots, n\}$ denote the index of the partition that p is in. We now define a new translation $\llbracket \cdot \rrbracket^{\approx}$, in which we use a separate memory valuation $M_{I(p)}$ when p is dereferenced. The definition of $\llbracket e \rrbracket^{\approx}$ is identical to the definition of $\llbracket e \rrbracket^{\mathcal{P}}$ unless e is a dereferencing expression. In this case, we use the following definition:

$$\llbracket *p \rrbracket^{\approx} \doteq M_{I(p)}(\llbracket p \rrbracket^{\approx}) \; .$$

Theorem 8.14. *Translation using memory partitioning results in a formula that is equisatisfiable with the result of the semantic translation:*

$$\exists \alpha_1 .\, \alpha_1 \models \llbracket \varphi \rrbracket^{\approx} \quad \iff \quad \exists \alpha_2 .\, \alpha_2 \models \llbracket \varphi \rrbracket \; .$$

Note that the theorem relies on the fact that our grammar does not permit explicit restrictions on the memory layout L. The theorem no longer holds as soon as this restriction is lifted (see Problem 8.5).

8.5 Rule-Based Decision Procedures

With pointer logics expressive enough to model interesting data structures, one often settles for incomplete, rule-based procedures. The basic idea of such procedures is to define a fragment of pointer logic enriched with predicates for specific types of data structures (e.g., lists or trees) together with a set of proof rules that are sufficient to prove a wide range of verification conditions that arise in practice. The soundness of these proof rules is usually shown with respect to the definitions of the predicates, which implies soundness of the decision procedure. There are only a few known proof systems that are provably complete.

8.5.1 A Reachability Predicate for Linked Structures

As a simple example of this approach, we present a variant of a calculus for reachability predicates introduced by Greg Nelson [204]. Further rule-based reasoning systems are discussed in the bibliographic notes at the end of this chapter.

We first generalize the $\mathsf{list\text{-}elem}$ shorthand used before for specifying linked lists by parameterizing it with the name of the field that holds the pointer to the "next" element. Suppose that f is a field of a structure and holds a pointer. The shorthand $\mathsf{follow}^f_n(q)$ stands for the pointer that is obtained by starting from q and following the field f, n times:

$$\begin{aligned} \mathsf{follow}^f_0(p) &\doteq p \\ \mathsf{follow}^f_n(p) &\doteq \mathsf{follow}^f_{n-1}(p)\text{->}f \ . \end{aligned} \tag{8.50}$$

If $\mathsf{follow}^f_n(p) = q$ holds, then q is reachable in n steps from p by following f. We say that q is reachable from p by following f if there exists such n. Using this shorthand, we enrich the logic with just a single predicate for list-like data structures, denoted by

$$p \xrightarrow[x]{f} q \ , \tag{8.51}$$

which is called a **reachability predicate**. It is read as "q is reachable from p following f, while avoiding x". It holds if two conditions are fulfilled:

1. There exists some n such that q is reachable from p by following f n times.
2. x is not reachable in fewer than n steps from p following f.

This can be formalized using $\mathsf{follow}()$ as follows:

$$p \xrightarrow[x]{f} q \iff \exists n.(\mathsf{follow}^f_n(p) = q \wedge \forall m < n.\mathsf{follow}^f_m(p) \neq x) \ . \tag{8.52}$$

We say that a formula is a **reachability predicate formula** if it contains the reachability predicate.

Example 8.15. Consider the following software verification problem. The following program fragment iterates over an acyclic list and searches for a list entry with payload a:

```
struct S { struct S *nxt; int payload; } *list;

...
bool find(int a) {
  for(struct S *p=list; p!=0; p=p->nxt)
    if(p->payload==a) return true;
  return false;
}
```

We can specify the correctness of the result returned by this procedure using the following formula:

$$\texttt{find}(a) \iff \exists p'.(\mathit{list} \overset{nxt}{\underset{0}{\to}} p' \wedge p'\text{->}\mathit{payload} = a)\ . \tag{8.53}$$

Thus, $\texttt{find}(a)$ is true if the following conditions hold:

1. There is a list element that is reachable from *list* by following *nxt* without passing through a NULL pointer.
2. The payload of this list element is equal to a.

We annotate the beginning of the loop body in the program above with the following loop invariant, denoted by INV:

$$\mathsf{INV} := \ \mathit{list} \overset{nxt}{\underset{0}{\to}} p \wedge (\forall q \neq p.\ \mathit{list} \overset{nxt}{\underset{p}{\to}} q \implies q\text{->}\mathit{payload} \neq a)\ . \tag{8.54}$$

Informally, we make the following argument: first, we show that the program maintains the loop invariant INV; then, we show that INV implies our property.

Formally, this is shown by means of four **verification conditions**. The validity of all of these verification conditions implies the property. We use the notation $e[x/y]$ to denote the expression e in which x is replaced by y.

$$\mathsf{IND\text{-}BASE} := p = \mathit{list} \implies \mathsf{INV} \tag{8.55}$$

$$\mathsf{IND\text{-}STEP} := (\mathsf{INV} \wedge p\text{->}\mathit{payload} \neq a) \implies \mathsf{INV}[p/p\text{->}\mathit{nxt}] \tag{8.56}$$

$$\begin{aligned}\mathsf{VC\text{-}P1} := {} & (\mathsf{INV} \wedge p\text{->}\mathit{payload} = a) \\ & \implies \exists p'.(\mathit{list} \overset{nxt}{\underset{0}{\to}} p' \wedge p'\text{->}\mathit{payload} = a)\end{aligned} \tag{8.57}$$

$$\mathsf{VC\text{-}P2} := (\mathsf{INV} \wedge p = 0) \implies \neg\exists p'.(\mathit{list} \overset{nxt}{\underset{0}{\to}} p' \wedge p'\text{->}\mathit{payload} = a) \tag{8.58}$$

The first verification condition, $\mathsf{IND\text{-}BASE}$, corresponds to the induction base of the inductive proof. It states that INV holds upon entering the loop, because at that point $p = \mathit{list}$. The formula $\mathsf{IND\text{-}STEP}$ corresponds to the induction step: it states that the loop invariant is maintained if another loop iteration is executed (i.e., $p\text{->}\mathit{payload} \neq a$).

The formulas $\mathsf{VC\text{-}P1}$ and $\mathsf{VC\text{-}P2}$ correspond to the two cases of leaving the `find` function: $\mathsf{VC\text{-}P1}$ establishes the property if TRUE is returned, and $\mathsf{VC\text{-}P2}$ establishes the property if FALSE is returned. Proving these verification conditions therefore shows that the program satisfies the required property. ▟

8.5.2 Deciding Reachability Predicate Formulas

As before, we can simply expand the definition above and obtain a semantic reduction. As an example, consider the verification condition labeled $\mathsf{IND\text{-}BASE}$ in Sect. 8.5.1:

$$p = \mathit{list} \implies \mathsf{INV} \tag{8.59}$$

$$\iff p = \mathit{list} \implies \mathit{list} \overset{nxt}{\underset{0}{\to}} p \wedge \forall q \neq p.\ \mathit{list} \overset{nxt}{\underset{p}{\to}} q \implies q\text{->}\mathit{payload} \neq a \tag{8.60}$$

$$\iff \mathit{list} \overset{nxt}{\underset{0}{\to}} \mathit{list} \wedge \forall q \neq \mathit{list}.\ (\mathit{list} \overset{nxt}{\underset{\mathit{list}}{\to}} q \implies q\text{->}\mathit{payload} \neq a) \tag{8.61}$$

$$\begin{aligned} \iff & (\exists n.\ \mathsf{follow}_n^{nxt}(\mathit{list}) = \mathit{list} \wedge \forall m < n.\ \mathsf{follow}_m^{nxt}(\mathit{list}) \neq \mathit{list}) \wedge \\ & (\forall q \neq \mathit{list}.\ ((\exists n.\ \mathsf{follow}_n^{nxt}(\mathit{list}) = q \wedge \forall m < n.\ \mathsf{follow}_m^{nxt}(\mathit{list}) \neq \mathit{list}) \\ & \quad \implies q\text{->}\mathit{payload} \neq a))\ . \end{aligned} \tag{8.62}$$

Equation (8.62) is argued to be valid as follows: In the first conjunction, instantiate n with 0. In the second conjunct, observe that $q \neq \mathit{list}$, and thus any n satisfying $\exists n.\ \mathsf{follow}_n^{nxt}(\mathit{list}) = q$ must be greater than 0. Finally, observe that $\mathsf{follow}_m^{nxt}(\mathit{list}) \neq \mathit{list}$ is invalid for $m = 0$, and thus the left-hand side of the implication is FALSE.

However, note that the formulas above contain many existential and universal quantifiers over natural numbers and pointers. Applying the semantic reduction therefore does not result in a formula that is in the array property fragment defined in Chap. 7. Thus, the decidability result shown in that chapter does not apply here. How can such complex reachability predicate formulas be solved?

Using Rules

In such situations, the following technique is frequently applied: *rules* are derived from the semantic definition of the predicate, and then they are applied to simplify the formula.

$$p \overset{f}{\underset{x}{\to}} q \iff (p = q \vee (p \neq x \wedge p\text{->}f \overset{f}{\underset{x}{\to}} q)) \tag{A1}$$

$$(p \overset{f}{\underset{x}{\to}} q \wedge q \overset{f}{\underset{x}{\to}} r) \implies p \overset{f}{\underset{x}{\to}} r \tag{A2}$$

$$p \overset{f}{\underset{x}{\to}} q \implies p \overset{f}{\underset{q}{\to}} q \tag{A3}$$

$$(p \overset{f}{\underset{y}{\to}} x \wedge p \overset{f}{\underset{z}{\to}} y) \implies p \overset{f}{\underset{z}{\to}} x \tag{A4}$$

$$(p \overset{f}{\underset{x}{\to}} x \vee p \overset{f}{\underset{y}{\to}} y) \implies (p \overset{f}{\underset{y}{\to}} x \vee p \overset{f}{\underset{x}{\to}} y) \tag{A5}$$

$$(p \overset{f}{\underset{y}{\to}} x \wedge p \overset{f}{\underset{z}{\to}} y) \implies x \overset{f}{\underset{z}{\to}} y \tag{A6}$$

$$p\text{->}f \overset{f}{\underset{q}{\to}} q \iff p\text{->}f \overset{f}{\underset{p}{\to}} q \tag{A7}$$

Fig. 8.4. Rules for the reachability predicate

The rules provided in [204] for our reachability predicate are given in Fig. 8.4. The first rule ($A1$) corresponds to a program fragment that follows field f once. If q is reachable from p, avoiding x, then either $p = q$ (we are already there) or $p \neq x$, and we can follow f from p to get to a node from which q is reachable, avoiding x. We now prove the correctness of this rule.

Proof. We first expand the definition of our reachability predicate:

$$p \overset{f}{\underset{x}{\to}} q \iff \exists n.\ (\mathsf{follow}^f_n(p) = q \wedge \forall m < n.\ \mathsf{follow}^f_m(p) \neq x)\ . \quad (8.63)$$

Observe that for any natural n, $n = 0 \vee n > 0$ holds, which we can therefore add as a conjunct:

$$\begin{aligned}\iff \exists n.\ ((n = 0 \vee n > 0) \wedge \\ \mathsf{follow}^f_n(p) = q \wedge \forall m < n.\ \mathsf{follow}^f_m(p) \neq x)\ .\end{aligned} \quad (8.64)$$

This simplifies as follows:

$$\iff \exists n.\ p = q \vee (n > 0 \wedge \mathsf{follow}^f_n(p) = q \wedge \forall m < n.\ \mathsf{follow}^f_m(p) \neq x) \quad (8.65)$$

$$\iff p = q \vee \exists n > 0.\ (\mathsf{follow}^f_n(p) = q \wedge \forall m < n.\ \mathsf{follow}^f_m(p) \neq x)\ . \quad (8.66)$$

We replace n by $n' + 1$ for natural n':

$$\iff p = q \vee \exists n'.\ (\mathsf{follow}^f_{n'+1}(p) = q \wedge \forall m < n' + 1.\ \mathsf{follow}^f_m(p) \neq x)\ . \quad (8.67)$$

As $\mathsf{follow}^f_{n'+1}(p) = \mathsf{follow}^f_{n'}(p\text{->}f)$, this simplifies to

$$\iff p = q \vee \exists n'.\ (\mathsf{follow}^f_{n'}(p\text{->}f) = q \wedge \forall m < n' + 1.\ \mathsf{follow}^f_m(p) \neq x)\ . (8.68)$$

By splitting the universal quantification into the two parts $m = 0$ and $m \geq 1$, we obtain

$$\begin{aligned}\iff p = q \vee \exists n'.\ (\mathsf{follow}^f_{n'}(p\text{->}f) = q \wedge \\ p \neq x \wedge \forall 1 \leq m < n' + 1.\ \mathsf{follow}^f_m(p) \neq x)\ .\end{aligned} \quad (8.69)$$

The universal quantification is rewritten:

$$\begin{aligned}\iff p = q \vee \exists n'.\ (\mathsf{follow}^f_{n'}(p\text{->}f) = q \wedge \\ p \neq x \wedge \forall m < n'.\ \mathsf{follow}^f_m(p\text{->}f) \neq x)\ .\end{aligned} \quad (8.70)$$

As the first and the third conjunct are equivalent to the definition of $p\text{->}f \overset{f}{\underset{x}{\to}} q$, the claim is shown. ∎

There are two simple consequences of rule ($A1$):

$$p \overset{f}{\underset{x}{\to}} p \quad \text{and} \quad p \overset{f}{\underset{p}{\to}} q \iff p = q\ . \quad (8.71)$$

In the following example we use these consequences to prove (8.61), the reachability predicate formula for our first verification condition.

Example 8.16. Recall (8.61):

$$\mathit{list} \overset{nxt}{\underset{0}{\to}} \mathit{list} \wedge \forall q \neq \mathit{list}.\ (\mathit{list} \overset{nxt}{\underset{list}{\to}} q \implies q\text{->}\mathit{payload} \neq a)\ . \tag{8.72}$$

The first conjunct is a trivial instance of the first consequence. To show the second conjunct, we introduce a **Skolem variable**[9] q' for the universal quantifier:

$$(q' \neq \mathit{list} \wedge \mathit{list} \overset{nxt}{\underset{list}{\to}} q') \implies q'\text{->}\mathit{payload} \neq a\ . \tag{8.73}$$

By the second consequence, the left-hand side of the implication is FALSE. ▟

Even when the axioms are used, however, reasoning about a reachability predicate remains tedious. The goal is therefore to devise an automatic decision procedure for a logic that includes a reachability predicate. We mention several decision procedures for logics with reachability predicates in the bibliographical notes.

8.6 Problems

8.6.1 Pointer Formulas

Problem 8.1 (semantics of pointer formulas). Determine if the following pointer logic formulas are valid using the semantic translation:

1. $x = y \implies \&x = \&y$.
2. $\&x \neq x$.
3. $\&x \neq \&y + i$.
4. $p \hookrightarrow x \implies *p = x$.
5. $p \hookrightarrow x \implies p\text{->}f = x$.
6. $(p_1 \hookrightarrow p_2, x_1 \wedge p_2 \hookrightarrow \text{NULL}, x_2) \implies p_1 \neq p_2$.

Problem 8.2 (modeling dynamically allocated data structures).

1. tt data structure is modeled by $\mathsf{my\text{-}ds}(q, l)$ in the following? Draw an example.

$$\begin{aligned} \mathsf{c}(q,0) &\doteq (*q).p = \text{NULL} \\ \mathsf{c}(q,i) &\doteq (*\mathsf{list\text{-}elem}(q,i)).p = \mathsf{list\text{-}elem}(q, i-1) \quad \text{for } i \geq 1 \\ \mathsf{my\text{-}ds}(q,l) &\doteq \mathsf{list\text{-}elem}(q,l) = \text{NULL} \wedge \forall 0 \leq i < l.\ \mathsf{c}(q,i) \end{aligned}$$

2. Write a recursive shorthand $\mathsf{DAG}(p)$ to denote that p points to the root of a directed acyclic graph.

[9] A Skolem variable is a ground variable introduced to eliminate a quantifier, i.e., $\forall x.P(x)$ is valid iff $P(x')$ is valid for a new variable x'. This is a special case of Skolemization, which is named after Thoralf Skolem.

3. Write a recursive shorthand $\mathsf{tree}(p)$ to denote that p points to the root of a tree.
4. Write a shorthand $\mathsf{hashtbl}(p)$ to denote that p points to an array of lists.

Problem 8.3 (extensions of the pointer logic). Consider a pointer logic that only permits a conjunction of predicates of the following form, where p is a pointer, and f_i, g_i are field identifiers:

$$\forall p.\ p\text{->}f_1\text{->}f_2\text{->}f_3 \ldots = p\text{->}g_1\text{->}g_2\text{->}g_3 \ldots$$

Show that this logic is Turing complete.

Problem 8.4 (axiomatization of the memory model). Define a set of memory model axioms for an architecture that uses 32-bit integers and little-endian byte ordering (this means that the least-significant byte has the lowest address in the word).

Problem 8.5 (partitioning the memory). Suppose that a pointer logic permits restrictions on L, the memory layout. Give a counterexample to Theorem 8.14.

8.6.2 Reachability Predicates

Problem 8.6 (semantics of reachability predicates). Determine the satisfiability of the following reachability predicate formulas:

1. $p \xrightarrow[p]{f} q \wedge p \neq q$.
2. $p \xrightarrow[x]{f} q \wedge p \xrightarrow[q]{f} x$.
3. $p \xrightarrow[q]{f} q \wedge q \xrightarrow[p]{f} p$.
4. $\neg(p \xrightarrow[q]{f} q) \wedge \neg(q \xrightarrow[p]{f} p)$.

Problem 8.7 (modeling). Try to write reachability predicate formulas for the following scenarios:

1. p points to a cyclic list where the next field is nxt.
2. p points to a NULL-terminated, doubly linked list.
3. p points to the root of a binary tree. The names of the fields for the left and right subtrees are l and r, respectively.
4. p points to the root of a binary tree as above, and the leaves are connected to a cyclic list.
5. p and q point to NULL-terminated singly linked lists that do not share cells.

Problem 8.8 (decision procedures). Build a decision procedure for a conjunction of atoms that have the form $p \xrightarrow[q]{f} q$ (or its negation).

Problem 8.9 (program verification). Write a code fragment that removes an element from a singly linked list, and provide the verification conditions using reachability predicate formulas.

8.7 Bibliographic Notes

The view of pointers as indices into a global array is commonplace, and similarly so is the identification of structure components with arrays. Leino's thesis is an instance of recent work applying this approach [181], and resembles our Sect. 8.3. An alternative point of view was proposed by Burstall: each component introduces an array, where the array indices are the addresses of the structures [60].

Transitive closure is frequently used to model recursive data structures. Immerman et al. explored the impact of adding transitive closure to a given logic. They showed that already very weak logics became undecidable as soon as transitive closure was added [153].

The PALE (Pointer Assertion Logic Engine) toolkit, implemented by Anders Møller, uses a graph representation for various dynamically allocated data structures. The graphs are translated into monadic second-order logic and passed to MONA, a decision procedure for this logic [198]. Michael Rabin proved in 1969 that the monadic second-order theory of trees was decidable [236].

The reachability predicate discussed in Sect. 8.5 was introduced by Greg Nelson [204]. This 1983 paper stated that the question of whether the set of (eight) axioms provided was complete remained open. A technical report gives a decision procedure for a conjunction of reachability predicates, which implies the existence of a complete axiomatization [207]. The procedure has linear time complexity.

Numerous modern logics are based on this idea. For example, Lahiri and Qadeer proposed two logics based on the idea of reachability predicates, and offered effective decision procedures [175, 176]. The decision procedure for [176] was based on a recent SMT solver.

Alain Deutsch [102] introduced an alias analysis algorithm that uses *symbolic access paths*, i.e., expressions that symbolically describe which field to follow for a given number of times. Symbolic access paths are therefore a generalization of the technique we described in Sect. 8.5. Symbolic access paths are very expressive when combined with an expressive logic for the *basis* of the access path, but this combination often results in undecidability.

Benedikt et al. [24] defined a logic for linked data structures. This logic uses constraints on paths (called *routing expressions*) in order to define memory regions, and permits one to reason about sharing and reachability within such regions. These authors showed the logic to be decidable using a small-model property argument, but did not provide an efficient decision procedure.

A major technique for analyzing dynamically allocated data structures is *parametric shape analysis*, introduced by Sagiv, Reps, and Wilhelm [240, 251, 282]. An important concept in the shape analysis of Sagiv et al. is the use of Kleene's three-valued logic for distinguishing predicates that are true, false, or *unknown* in a particular abstract state. The resulting concretizations are more precise than an abstraction using traditional, two-valued logic.

Separation logic (see the aside on this subject on p. 184) was introduced by John Reynolds as an intuitionistic way of reasoning about dynamically allocated data structures [242]. Calcagno et al. [64] showed that deciding the validity of a formula in separation logic, even if robbed of its characteristic separating conjunction, was not recursively enumerable. On the other hand, they showed that, once quantifiers were prohibited, validity became decidable. Decidable fragments of separation logic have been studied, for example, by Berdine et al. [25, 26, 27]; these are typically restricted to predicates over lists. Parkinson and Bierman address the problem of modular reasoning about programs using separation logic [217].

Kuncak and Rinard introduced *regular graph constraints* as a representation of heaps. They showed that satisfiability of such heap summary graphs was decidable, whereas entailment was not [172].

Alias analysis techniques have also been integrated directly into verification algorithms. Manevich et al. described predicate abstraction techniques for singly linked lists [187]. Beyer et al. described how to combine a predicate abstraction tool that implements lazy abstraction with shape analysis [28]. Podelski and Wies propose *Boolean heaps* as an abstract model for heap-manipulating programs [230]. Here, the abstract domain is spanned by a vector of arbitrary first-order predicates characterizing the heap. Bingham and Rakamarić [36] also proposed to extend predicate abstraction with predicates designated to describe the heap. Distefano et al. [103] defined an abstract domain that is based on predicates drawn from separation logic. Berdine et al. use separation logic predicates in an add-on to Microsoft's SLAM device driver verifier, called TERMINATOR, in order to prove that loops iterating over dynamically allocated data structures terminate. A graph-based decision procedure for heaps that relies on a small-model property is given in [85]. The procedure is able to reason about the length of list segments.

Most frameworks for reasoning about dynamically allocated memory treat the heap as composed of disjoint memory fragments, and do not model accesses beyond these fragments using pointer arithmetic. Calcagno et al. introduced a variant of separation logic that permits reasoning about low-level programs including pointer arithmetic [63]. This logic permits the analysis of infrastructure usually assumed to exist at higher abstraction layers, e.g., the code that implements the `malloc` function.

8.8 Glossary

The following symbols were used in this chapter:

Symbol	**Refers to** ...	**First used on page** ...
A	Set of addresses	174
D	Set of data words	174
M	Map from addresses to data words	174
L	Memory layout	174
$\sigma(v)$	The size of v	174
V	Set of variables	174
$[\![\cdot]\!]$	Semantics of pointer expressions	179
$p \hookrightarrow z$	p points to a variable with value z	179
p->f	Shorthand for $(*p).f$	181
$\mathsf{list}(p, l)$	p points to a list of length l	182

9

Quantified Formulas

9.1 Introduction

Quantification allows us to specify the extent of validity of a predicate, or in other words the **domain** (range of values) in which the predicate should hold. The syntactic element used in the logic for specifying quantification is called a **quantifier**. The most commonly used quantifiers are the *universal quantifier*, denoted by "$\forall$", and the *existential quantifier*, denoted by "$\exists$". These two quantifiers are interchangeable using the following equivalence:

$\boxed{\forall}$ $\boxed{\exists}$

$$\forall x.\ \varphi \iff \neg \exists x.\ \neg \varphi\ . \tag{9.1}$$

Some examples of quantified statements are:

- For any integer x, there is an integer y smaller than x:

$$\forall x \in \mathbb{Z}.\ \exists y \in \mathbb{Z}.\ y < x\ . \tag{9.2}$$

- There exists an integer y such that, for any integer x, x is greater than y:

$$\exists y \in \mathbb{Z}.\ \forall x \in \mathbb{Z}.\ x > y\ . \tag{9.3}$$

- (Bertrand's postulate) For any natural number n greater than 1, there is a prime number p such that $n < p < 2n$:

$$\forall n \in \mathbb{N}.\ \exists p \in \mathbb{N}.\ n > 1 \implies (isprime(p) \wedge n < p < 2n)\ . \tag{9.4}$$

In these three examples, there is **quantifier alternation** between the universal and existential quantifiers. In fact, the satisfiability and validity problems that we considered in earlier chapters can be cast as decision problems for formulas with nonalternating quantifiers. When we ask whether the propositional formula

$$x \vee y \tag{9.5}$$

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_9

is satisfiable, we can equivalently ask whether *there exists* a truth assignment to x, y that satisfies this formula.[1] And when we ask whether

$$x > y \lor x < y \tag{9.6}$$

is valid for $x, y \in \mathbb{N}$, we can equivalently ask whether this formula holds *for all* naturals x and y. The formulations of these two decision problems are, respectively,

$$\exists x \in \mathbb{B}.\ \exists y \in \mathbb{B}.\ (x \lor y) \tag{9.7}$$

and

$$\forall x \in \mathbb{N}.\ \forall y \in \mathbb{N}.\ x > y \lor x < y\ . \tag{9.8}$$

We omit the domain of each quantified variable from now on when it is not essential for the discussion.

An important characteristic of quantifiers is the scope in which they are applied, called the **binding scope**. For example, in the following formula, the existential quantification over x overrides the external universal quantification over x:

$$\forall x.\ \underbrace{((x < 0) \land \exists y.\ \overbrace{(y > x \land (y \geq 0 \lor \exists x.\ \underbrace{(y = x + 1)}_{\text{scope of } \exists x}))}^{\text{scope of } \exists y})}_{\text{scope of } \forall x}\ . \tag{9.9}$$

Within the scope of the second existential quantifier, all *occurrences* of x refer to the variable bound by the existential quantifier. It is impossible within this scope to refer directly to the variable x bound by the universal quantifier. A possible solution is to rename x in the inner scope: clearly, this does not change the validity of the formula. After this renaming, we can assume that every occurrence of a variable is bound at most once.

Definition 9.1 (free variable). *A variable is called* free *in a given formula if at least one of its occurrences is not bound by any quantifier.*

Definition 9.2 (sentence). *A formula Q is called a* sentence *(or* closed*) if none of its variables are free.*

In this chapter we only focus on sentences.

Arbitrary first-order theories with quantifiers are undecidable. We restrict the discussion in this chapter to decidable theories only (other than in Sect. 9.5), and begin with two examples.

[1] As explained in Sect. 1.4.1, the difference between the two formulations, namely with no quantifiers and with nonalternating quantifiers, is that in the former all variables are free (unquantified), and hence a satisfying structure (a *model*) for such formulas includes an assignment to these variables. Since such assignments are necessary in many applications, this book uses the former formulation.

9.1.1 Example: Quantified Boolean Formulas

Quantified propositional logic is propositional logic enhanced with quantifiers. Sentences in quantified propositional logic are better known as **quantified Boolean formulas** (QBFs). The set of sentences permitted by the logic is defined by the following grammar:

$$\begin{array}{c} \mathit{formula} : \mathit{formula} \wedge \mathit{formula} \mid \neg \mathit{formula} \mid (\mathit{formula}) \mid \\ \mathit{identifier} \mid \exists\, \mathit{identifier}.\ \mathit{formula} \end{array}$$

Other symbols such as "$\vee$", "$\forall$", and "$\Longleftrightarrow$" can be constructed using elements of the formal grammar. Examples of quantified Boolean formulas are

- $\forall x.\ (x \vee \exists y.\ (y \vee \neg x))$,
- $\forall x.\ (\exists y.\ ((x \vee \neg y) \wedge (\neg x \vee y)) \wedge \exists y.\ ((\neg y \vee \neg x) \wedge (x \vee y)))$.

Complexity

The validity problem of QBF is PSPACE-complete, which means that it is theoretically harder to solve than SAT, which is "only" NP-complete.[2] Both of these problems (SAT and the QBF problem) are frequently presented as the quintessential problems of their respective complexity classes. The known algorithms for both problems are exponential.

Usage Example: Chess

The following is an example of the use of QBF.

Example 9.3. QBF is a convenient way of modeling many finite two-player games. As an example, consider the problem of determining whether there is a winning strategy for a chess player in k steps, i.e., given a state of a board and assuming white goes first, can white take the black king in k steps, regardless of black's moves? This problem can be modeled as QBF rather naturally, because what we ask is whether there *exists* a move of white such that *for all* possible moves of black that follow there *exists* a move of white such that *for all* possible moves of black... and so forth, k times, such that the goal of eliminating the black king is achieved. The number of steps k has to be an odd natural, as white plays both the first and last move.

[2] The difference between these two classes is that problems in NP are known to have nondeterministic algorithms that solve them in polynomial time. It has not been proven that these two classes are indeed different, but it is widely suspected that this is the case.

This is a classical **planning problem**. Planning is a popular field of study in artificial intelligence. To formulate the chess problem in QBF,[3] we use the notation in Fig. 9.1. Every piece of each player has a unique index. Each location on the board has a unique index as well, and the location 0 of a piece indicates that it is outside the board. The size of the board is s (normally $s = 8$), and hence there are $s^2 + 1$ locations and $4s$ pieces.

Symbol	Meaning
$x_{\{m,n,i\}}$	Piece m is at location n in step i, for $1 \le m \le 4s$, $0 \le n \le s^2$, and $0 \le i \le k$
I_0	A set of clauses over the $x_{\{m,n,0\}}$ variables that represent the initial state of the board
T_i^w	A set of clauses over the $x_{\{m,n,i\}}, x_{\{m,n,i+1\}}$ variables that represent the valid moves by white at step i
T_i^b	A set of clauses over the $x_{\{m,n,i\}}, x_{\{m,n,i+1\}}$ variables that represent the valid moves by black at step i
G_k	A set of clauses over the $x_{\{m,n,k\}}$ variables that represent the goal, i.e., in step k the black king is off the board and the white king is on the board

Fig. 9.1. Notation used in Example 9.3

We use the following convention: we write $\{x_{\{m,n,i\}}\}$ to represent the set of variables $\{x_{\{m,n,i\}} \mid m, n, i$ in their respective ranges$\}$. Let us begin with the following attempt to formulate the problem:

$$\exists\{x_{\{m,n,0\}}\}\exists\{x_{\{m,n,1\}}\}\forall\{x_{\{m,n,2\}}\}\exists\{x_{\{m,n,3\}}\}\cdots\forall\{x_{\{m,n,k-1\}}\}\exists\{x_{\{m,n,k\}}\}.$$
$$I_0 \wedge (T_0^w \wedge T_2^w \wedge \cdots \wedge T_{k-1}^w) \wedge (T_1^b \wedge T_3^b \wedge \cdots \wedge T_{k-2}^b) \wedge G_k \; . \tag{9.10}$$

This formulation includes the necessary restrictions on the initial and goal states, as well as on the allowed transitions. The problem is that this formula is not valid even when there is a winning strategy for white, because black can make an illegal move—such as moving two pieces at once—which falsifies the formula (it contradicts the subformula T_i for some odd i).

The formula needs to be weakened, as it is sufficient to find a white move for the *legal* moves of black:

[3] Classical formulations of planning problems distinguish between actions (moves in this case) and states. Here we choose to present a formulation based on states only.

$$\exists\{x_{\{m,n,0\}}\}\exists\{x_{\{m,n,1\}}\}\forall\{x_{\{m,n,2\}}\}\exists\{x_{\{m,n,3\}}\}\cdots\forall\{x_{\{m,n,k-1\}}\}\exists\{x_{\{m,n,k\}}\}.$$
$$I_0 \wedge ((T_1^b \wedge T_3^b \wedge \cdots \wedge T_{k-2}^b) \implies (T_0^w \wedge T_2^w \wedge \cdots \wedge T_{k-1}^w \wedge G_k))\ . \tag{9.11}$$

Is this formula a faithful representation of the chess problem? Unfortunately not yet, because of the possibility of a stalemate: there could be a situation in which black is not able to make a valid move, which results in a draw. In such a case (9.11) is valid although it should not be. A possible solution for this problem is to ban white from making moves that result in such a state by modifying T^w appropriately. ■

9.1.2 Example: Quantified Disjunctive Linear Arithmetic

The syntax of **quantified disjunctive linear arithmetic** (QDLA) is defined by the following grammar:

$$\begin{aligned} \mathit{formula} &: \mathit{formula} \wedge \mathit{formula} \mid \neg \mathit{formula} \mid (\mathit{formula}) \mid \\ &\quad \mathit{predicate} \mid \forall\, \mathit{identifier}.\ \mathit{formula} \\ \mathit{predicate} &: \Sigma_i a_i x_i \leq c \end{aligned}$$

where c and a_i are constants, and x_i are variables of type real, for all i. As before, other symbols such as "$\vee$", "$\exists$", and "$=$" can be defined using the formal grammar.

Aside: Presburger Arithmetic
Presburger arithmetic has the same grammar as quantified disjunctive linear arithmetic, but is defined over the natural numbers rather than over the reals. Presburger arithmetic is decidable and, as proven by Fischer and Rabin [114], there is a lower bound of $2^{2^{c \cdot n}}$ on the worst-case run-time complexity of a decision procedure for this theory, where n is the length of the input formula and c is a positive constant. This theory is named after Mojzesz Presburger, who introduced it in 1929 and proved its decidability. Replacing the Fourier–Motzkin procedure with the Omega test (see Sect. 5.5) in the procedure described in this section gives a decision procedure for this theory. Other decision procedures for Presburger arithmetic are mentioned in the bibliographic notes at the end of this chapter.

As an example, the following is a QDLA formula:

$$\forall x.\ \exists y.\ \exists z.\ ((y + 1 \leq x \quad \vee \quad z + 1 \leq y) \quad \wedge \quad 2x + 1 \leq z)\ . \tag{9.12}$$

9.2 Quantifier Elimination

9.2.1 Prenex Normal Form

We begin by defining a normal form for quantified formulas.

Definition 9.4 (prenex normal form). *A formula is said to be in* prenex normal form *(PNF) if it is in the form*

$$Q[n]V[n]\dots Q[1]V[1].\ \langle \textit{quantifier-free formula}\rangle\ , \tag{9.13}$$

where for all $i \in \{1,\dots,n\}$, $Q[i] \in \{\forall, \exists\}$ *and* $V[i]$ *is a variable.*

We call the quantification string on the left of the formula the **quantification prefix**, and call the quantifier-free formula to the right of the quantification prefix the **quantification suffix** (also called the *matrix*).

Lemma 9.5. *For every quantified formula* $\mathcal{Q}$ *there exists a formula* $\mathcal{Q}'$ *in prenex normal form such that* $\mathcal{Q}$ *is valid if and only if* $\mathcal{Q}'$ *is valid.*

Algorithm 9.2.1 transforms an input formula into prenex normal form.

Algorithm 9.2.1: Prenex

Input: A quantified formula
Output: A formula in prenex normal form

1. Eliminate Boolean connectives other than $\vee$, $\wedge$, and $\neg$.
2. Push negations to the right across all quantifiers, using De Morgan's rules (see Sect. 1.3) and (9.1).
3. If there are name conflicts across scopes, solve by renaming: give each variable in each scope a unique name.
4. Move quantifiers out by using equivalences such as

$$\begin{aligned}
\phi_1 \wedge Qx.\ \phi_2(x) &\iff Qx.\ (\phi_1 \wedge \phi_2(x))\ ,\\
\phi_1 \vee Qx.\ \phi_2(x) &\iff Qx.\ (\phi_1 \vee \phi_2(x))\ ,\\
Q_1y.\ \phi_1(y) \wedge Q_2x.\ \phi_2(x) &\iff Q_1y.\ Q_2x.\ (\phi_1(y) \wedge \phi_2(x))\ ,\\
Q_1y.\ \phi_1(y) \vee Q_2x.\ \phi_2(x) &\iff Q_1y.\ Q_2x.\ (\phi_1(y) \vee \phi_2(x))\ ,
\end{aligned}$$

where $Q, Q_1, Q_2 \in \{\forall, \exists\}$ are quantifiers, $x \not\in \mathit{var}(\phi_1)$, and $y \not\in \mathit{var}(\phi_2)$.

Example 9.6. We demonstrate Algorithm 9.2.1 with the following formula:

$$\mathcal{Q} := \neg\exists x.\ \neg(\exists y.\ ((y \implies x) \wedge (\neg x \vee y)) \wedge \neg\forall y.\ ((y \wedge x) \vee (\neg x \wedge \neg y)))\ . \tag{9.14}$$

In steps 1 and 2, eliminate " $\implies$ " and push negations inside:

$$\forall x.\ (\exists y.\ ((\neg y \vee x) \wedge (\neg x \vee y)) \wedge \exists y.\ ((\neg y \vee \neg x) \wedge (x \vee y)))\ . \tag{9.15}$$

In step 3, rename y as there are two quantifications over this variable:

$$\forall x.\ (\exists y_1.\ ((\neg y_1 \vee x) \wedge (\neg x \vee y_1)) \wedge \exists y_2.\ ((\neg y_2 \vee \neg x) \wedge (x \vee y_2)))\ . \tag{9.16}$$

Finally, in step 4, move quantifiers to the left of the formula:

$$\forall x.\ \exists y_1.\ \exists y_2.\ (\neg y_1 \vee x) \wedge (\neg x \vee y_1) \wedge (\neg y_2 \vee \neg x) \wedge (x \vee y_2)\ . \tag{9.17}$$

▟

We assume from here on that the input formula is given in prenex normal form.[4]

9.2.2 Quantifier Elimination Algorithms

A *quantifier elimination* algorithm transforms a quantified formula into an equivalent formula without quantifiers. Not every theory has a quantifier elimination algorithm. In fact, the existence of a quantifier elimination algorithm typically implies the decidability of the logic, and not all theories are decidable.

It is sufficient to show that there exists a procedure for eliminating an existential quantifier. Universal quantifiers can be eliminated by making use of (9.1). For this purpose we define a general notion of *projection*, which has to be concretized for each individual theory.

Definition 9.7 (projection). *A* projection *of a variable x from a quantified formula in prenex normal form with n quantifiers,* $\boxed{n}$

$$\mathcal{Q}_1 = Q[n]V[n] \ldots Q[2]V[2].\ \exists x.\ \phi\ , \tag{9.18}$$

is a formula

$$\mathcal{Q}_2 = Q[n]V[n] \ldots Q[2]V[2].\ \phi' \tag{9.19}$$

(where both ϕ and ϕ' are quantifier-free), such that $x \notin var(\phi')$, and $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are logically equivalent.

Given a projection algorithm Project, Algorithm 9.2.2 eliminates all quantifiers. Assuming that we begin with a sentence (see Definition 9.2), the remaining formula is over constants and easily solvable.

[4] Whereas the method in the following section relies on this form, there are alternatives that do not. The process of pushing quantifiers *inwards* is called **miniscoping**. A good example of miniscoping for solving QBF can be found in [9].

Algorithm 9.2.2: Quantifier-Elimination

Input: A sentence $Q[n]V[n]\ldots Q[1]V[1].\ \phi$, where ϕ is quantifier-free
Output: A (quantifier-free) formula over constants ϕ', which is valid if and only if ϕ is valid

```
$\phi' := \phi$;
for $i := 1, \ldots, n$ do
    if $Q[i] = \exists$ then
        $\phi' :=$ Project( $\phi', V[i]$);
    else
        $\phi' := \neg$Project($\neg\phi', V[i]$);
Return $\phi'$;
```

We now give two examples of projection procedures and their use in quantifier elimination.

9.2.3 Quantifier Elimination for Quantified Boolean Formulas

Eliminating an existential quantifier over a conjunction of Boolean literals is trivial: if x appears with both phases in the conjunction, then the formula is unsatisfiable; otherwise, x can be removed. For example,

$$\begin{array}{ll} \exists y.\ \exists x.\ x \land \neg x \land y & = \text{FALSE}\ , \\ \exists y.\ \exists x.\ x \land y = \exists y.\ y & = \text{TRUE}\ . \end{array} \tag{9.20}$$

This observation can be used if we first convert the quantification suffix to DNF and then apply projection to each term separately. This is justified by the following equivalence:

$$\exists x.\ \bigvee_i \bigwedge_j l_{ij} \iff \bigvee_i \exists x.\ \bigwedge_j l_{ij}\ , \tag{9.21}$$

where l_{ij} are literals. But since converting formulas to DNF can result in an exponential growth in the formula size (see Sect. 1.16), it is preferable to have a projection that works directly on the CNF, or better yet, on a general Boolean formula. We consider two techniques: *binary resolution* (see Definition 2.11), which works directly on CNF formulas, and *expansion.*

Projection with Binary Resolution

Resolution gives us a method to eliminate a variable x from a pair of clauses in which x appears with opposite phases. To eliminate x from a CNF formula by projection (Definition 9.7), we need to apply resolution to *all* pairs of clauses where x appears with opposite phases. This eliminates x together with its quantifier. For example, given the formula

$$\exists y.\ \exists z.\ \exists x.\ (y \lor x) \land (z \lor \neg x) \land (y \lor \neg z \lor \neg x) \land (\neg y \lor z)\ , \tag{9.22}$$

we can eliminate x together with $\exists x$ by applying resolution on x to the first and second clause, and to the first and third clause, resulting in

$$\exists y.\ \exists z.\ (y \lor z) \land (y \lor \neg z) \land (\neg y \lor z)\ . \tag{9.23}$$

What about universal quantifiers? Relying on (9.1), in the case of CNF formulas, results in a surprisingly easy shortcut to eliminating universal quantifiers: simply erase them from the formula. For example, eliminating x and $\forall x$ from

$$\exists y.\ \exists z.\ \forall x.\ (y \lor x) \land (z \lor \neg x) \land (y \lor \neg z \lor \neg x) \land (\neg y \lor z) \tag{9.24}$$

results in

$$\exists y.\ \exists z.\ (y) \land (z) \land (y \lor \neg z) \land (\neg y \lor z)\ . \tag{9.25}$$

This step is called **forall reduction**. It should be applied only after removing tautology clauses (clauses in which a literal appears with both phases). We leave the proof of correctness of this step to Problem 9.4. Intuitively, however, it is easy to see why this is correct: if the formula is evaluated to TRUE for all values of x, this means that we cannot satisfy a clause while relying on a specific value of x.

Example 9.8. In this example, we show how to use resolution on both universal and existential quantifiers. Consider the following formula:

$$\begin{array}{l}\forall u_1.\ \forall u_2.\ \exists e_1.\ \forall u_3.\ \exists e_3.\ \exists e_2.\\ (u_1 \lor \neg e_1) \land (\neg u_1 \lor \neg e_2 \lor e_3) \land (u_2 \lor \neg u_3 \lor \neg e_1) \land (e_1 \lor e_2) \land (e_1 \lor \neg e_3)\ .\end{array} \tag{9.26}$$

By resolving the second and fourth clause on e_2, we obtain

$$\begin{array}{l}\forall u_1.\ \forall u_2.\ \exists e_1.\ \forall u_3.\ \exists e_3.\\ (u_1 \lor \neg e_1) \land (\neg u_1 \lor e_1 \lor e_3) \land (u_2 \lor \neg u_3 \lor \neg e_1) \land (e_1 \lor \neg e_3)\ .\end{array} \tag{9.27}$$

By resolving the second and fourth clause on e_3, we obtain

$$\forall u_1.\ \forall u_2.\ \exists e_1.\ \forall u_3.\ (u_1 \lor \neg e_1) \land (\neg u_1 \lor e_1) \land (u_2 \lor \neg u_3 \lor \neg e_1)\ . \tag{9.28}$$

By eliminating u_3, we obtain

$$\forall u_1.\ \forall u_2.\ \exists e_1.\ (u_1 \lor \neg e_1) \land (\neg u_1 \lor e_1) \land (u_2 \lor \neg e_1)\ . \tag{9.29}$$

By resolving the first and second clause on e_1, and the second and third clause on e_1, we obtain

$$\forall u_1.\ \forall u_2.\ (u_1 \lor \neg u_1) \land (\neg u_1 \lor u_2)\ . \tag{9.30}$$

The first clause is a tautology and hence is removed. Next, u_1 and u_2 are removed, which leaves us with the empty clause. The formula, therefore, is not valid. ▟

What is the complexity of this procedure? Consider the elimination of a quantifier $\exists x$, and let n, m denote the number of quantifiers and clauses, respectively. In the worst case, half of the clauses contain x and half $\neg x$. Since we create a new clause from each pair of the two types of clauses, this results in $O(m^2)$ new clauses, while we erase the m old clauses that contain x. Repeating this process n times, once for each quantifier, results in $O(m^{2^n})$ clauses.

$\boxed{N}$

This seems to imply that the complexity of projection with binary resolution is doubly exponential. This, in fact, is only true if we do not prevent duplicate clauses. Observe that there cannot be more than 3^N distinct clauses, where N is the total number of variables. The reason is that each variable can appear positively, negatively, or not at all in a clause. This implies that, if we add each clause at most once, the number of clauses is only singly exponential in n (assuming N is not exponentially larger than n).

Expansion-Based Quantifier Elimination

The following quantifier elimination technique is based on *Shannon expansion*, which is summarized by the following equivalences:

$$\exists x.\ \varphi = \varphi|_{x=0} \vee \varphi|_{x=1}\ , \tag{9.31}$$

$$\forall x.\ \varphi = \varphi|_{x=0} \wedge \varphi|_{x=1}\ . \tag{9.32}$$

The notation $\varphi|_{x=0}$ simply means that x is replaced with 0 (FALSE) in φ. Note that (9.32) can be derived from (9.31) by using (9.1).

Projections using expansion result in formulas that grow to $O(m \cdot 2^n)$ clauses in the worst case, where, as before, m is the number of clauses in the original formula. This technique can be applied directly to non-CNF formulas, in contrast to resolution, as the following example shows:

Example 9.9. Consider the following formula:

$$\exists y.\ \forall z.\ \exists x.\ (y \vee (x \wedge z))\ . \tag{9.33}$$

Applying (9.31) to $\exists x$ results in

$$\exists y.\ \forall z.\ (y \vee (x \wedge z))|_{x=0} \vee (y \vee (x \wedge z))|_{x=1}\ , \tag{9.34}$$

which simplifies to

$$\exists y.\ \forall z.\ (y \vee z)\ . \tag{9.35}$$

Applying (9.32) yields

$$\exists y.\ (y \vee z)|_{z=0} \wedge (y \vee z)|_{z=1}\ , \tag{9.36}$$

which simplifies to

$$\exists y.\ (y)\ , \tag{9.37}$$

which is obviously valid. Hence, (9.33) is valid. ⌟

Competitive QBF solvers apply expansion to all but the last level of quantifiers. Since most QBF formulas that are found in practice have only one quantifier alternation (a fragment called **2-QBF**), this means that only one level has to be expanded. If the last (most external) level consists of the "$\exists$" quantifier, they run a SAT solver on the remaining formula. Otherwise, they negate the formula and do the same thing. Naïve expansion as described above works only if the formulas are relatively small. To improve scalability one may apply simplification after each expansion step, which removes FALSE literals, clauses with TRUE literals, and clauses that are subsumed by other clauses.

Since we can reduce QBF to SAT via expansion, what does this tell us about the expressive power of QBF? That it is not more expressive than propositional logic. It only offers a more succinct representation, in fact exponentially more succinct, which explains why QBF is not in NP.

9.2.4 Quantifier Elimination for Quantified Disjunctive Linear Arithmetic

Once again we need a projection method. We use the Fourier–Motzkin elimination, which was described in Sect. 5.4. This technique resembles the resolution method introduced in Sect. 9.2.3, and has a worst-case complexity of $O(m^{2^n})$. It can be applied directly to a conjunction of linear atoms and, consequently, if the input formula has an arbitrary structure, it has to be converted first to DNF.

Let us briefly recall the Fourier–Motzkin elimination method. In order to eliminate a variable x_n from a formula with variables $x_1, \ldots, x_n$, for every two conjoined constraints of the form

$$\sum_{i=1}^{n-1} a'_i \cdot x_i < x_n < \sum_{i=1}^{n-1} a_i \cdot x_i \ , \tag{9.38}$$

where for $i \in \{1, \ldots, n-1\}$, a_i and a'_i are constants, we generate a new constraint

$$\sum_{i=1}^{n-1} a'_i \cdot x_i < \sum_{i=1}^{n-1} a_i \cdot x_i \ . \tag{9.39}$$

After generating all such constraints for x_n, we remove all constraints that involve x_n from the formula.

Example 9.10. Consider the following formula:

$$\forall x.\ \exists y.\ \exists z.\ (y + 1 \leq x \quad \wedge \quad z + 1 \leq y \quad \wedge \quad 2x + 1 \leq z) \ . \tag{9.40}$$

By eliminating z, we obtain

$$\forall x.\ \exists y.\ (y + 1 \leq x \quad \wedge \quad 2x + 1 \leq y - 1) \ . \tag{9.41}$$

By eliminating y, we obtain

$$\forall x.\ (2x + 2 \leq x - 1)\ . \tag{9.42}$$

Using (9.1), we obtain

$$\neg\exists x.\ \neg(2x + 2 \leq x - 1)\ , \tag{9.43}$$

or, equivalently,

$$\neg\exists x.\ x > -3\ , \tag{9.44}$$

which is obviously not valid. ▟

Several alternative quantifier elimination procedures are cited in the bibliographic notes at the end of this chapter.

9.3 Search-Based Algorithms for Quantified Boolean Formulas

Most competitive QBF solvers are based on an adaptation of CDCL solvers. The adaptation that we consider here is naive, in that it resembles the basic CDCL algorithm without the more advanced features such as learning and nonchronological backtracking (see Chap. 2 for details on the CDCL algorithmic framework).

The key difference between SAT and the QBF problem is that the latter requires handling of quantifier alternation. The binary search tree now has to distinguish between **universal nodes** and **existential nodes**. Universal nodes are labeled with a symbol "∀", as can be seen in the right-hand drawing in Fig. 9.2.

Fig. 9.2. An existential node (*left*) and a universal node (*right*) in a QBF search tree

A QBF binary search tree corresponding to a QBF $\mathcal{Q}$, is defined as follows:

Definition 9.11 (QBF search tree corresponding to a quantified Boolean formula). *Given a QBF $\mathcal{Q}$ in prenex normal form and an ordering of its variables (say, $x_1, \ldots, x_n$), a* QBF search tree *corresponding to $\mathcal{Q}$ is a binary labeled tree of height $n + 1$ with two types of internal nodes, universal and existential, in which:*

- *The root node is labeled with $\mathcal{Q}$ and associated with depth 0.*
- *One of the children of each node at level i, $0 \le i < n$, is marked with x_{i+1}, and the other with $\neg x_{i+1}$.*
- *A node in level i, $0 \le i < n$, is* universal *if the variable in level $i+1$ is universally quantified.*
- *A node in level i, $0 \le i < n$, is* existential *if the variable in level $i+1$ is existentially quantified.*

The validity of a QBF tree is defined recursively, as follows:

Definition 9.12 (validity of a QBF tree). *A QBF tree is valid if its root is satisfied. This is determined recursively according to the following rules:*

- *A leaf in a QBF binary tree corresponding to a QBF $\mathcal{Q}$ is satisfied if the assignment corresponding to the path to this leaf satisfies the quantification suffix of $\mathcal{Q}$.*
- *A universal node is satisfied if both of its children are satisfied.*
- *An existential node is satisfied if at least one of its children is satisfied.*

Example 9.13. Consider the formula

$$\mathcal{Q} := \exists e.\ \forall u.\ (e \vee u) \wedge (\neg e \vee \neg u)\ . \tag{9.45}$$

The corresponding QBF tree appears in Fig. 9.3.

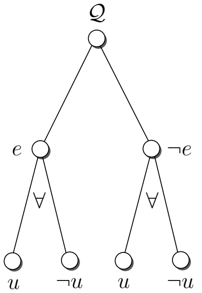


Fig. 9.3. A QBF search tree for the formula $\mathcal{Q}$ of (9.45)

The second and third u nodes are the only nodes that are satisfied (since $(e, \neg u)$ and $(\neg e, u)$ are the only assignments that satisfy the suffix). Their parent nodes, e and $\neg e$, are not satisfied, because they are universal nodes and only one of their child nodes is satisfied. In particular, the root node, representing $\mathcal{Q}$, is not satisfied and hence $\mathcal{Q}$ is not valid. ▟

A naive implementation based on these ideas is described in Algorithm 9.3.1. More sophisticated algorithms exist [296, 297], in which techniques such as

```
Algorithm 9.3.1: SEARCH-BASED-DECISION-OF-QBF

Input:   A QBF $\mathcal{Q}$ in PNF $Q[n]V[n] \ldots\ Q[1]V[1].\ \phi$, where $\phi$ is in
         CNF
Output:  "Valid" if $\mathcal{Q}$ is valid, and "Not valid" otherwise

 1. function MAIN(QBF formula $\mathcal{Q}$)
 2.     if QBF($\mathcal{Q}, \emptyset, n$) then return "Valid";
 3.     else return "Not valid";
 4.
 5. function QBF($\mathcal{Q}$, assignment set $\hat{v}$, $level \in \mathbb{N}_0$)
 6.     if ($\phi|_{\hat{v}}$ simplifies to FALSE) then return FALSE;
 7.     if ($level = 0$) then return TRUE;
 8.     if ($Q[level] = \forall$) then
 9.         return $\left(\begin{array}{l} \mathrm{QBF}(\mathcal{Q}, \hat{v} \cup \neg V[level], level - 1) \wedge \\ \mathrm{QBF}(\mathcal{Q}, \hat{v} \cup \ \ V[level], level - 1) \end{array}\right)$;
10.     else
11.         return $\left(\begin{array}{l} \mathrm{QBF}(\mathcal{Q}, \hat{v} \cup \neg V[level], level - 1) \vee \\ \mathrm{QBF}(\mathcal{Q}, \hat{v} \cup \ \ V[level], level - 1) \end{array}\right)$;
```

nonchronological backtracking and learning are applied: as in SAT, in the QBF problem we are not interested in searching the whole search space defined by the graph above, but rather in pruning it as much as possible.

$\phi|_{\hat{v}}$

The notation $\phi|_{\hat{v}}$ in line 6 refers to the simplification of ϕ resulting from the assignments in the assignment set $\hat{v}$. For example, let $\hat{v} := \{x \mapsto 0, y \mapsto 1\}$. Then

$$(x \vee (y \wedge z))|_{\hat{v}} = (z)\ . \tag{9.46}$$

Example 9.14. Consider (9.45) once again:

$$\mathcal{Q} :=\ \exists e.\ \forall u.\ (e \vee u) \wedge (\neg e \vee \neg u)\ .$$

The progress of Algorithm 9.3.1 when applied to this formula, with the variable ordering u, e, is shown in Fig. 9.4.

▟

9.4 Effectively Propositional Logic

Many problems can be modeled with a fragment of first-order logic known by the name **Effectively Propositional** (EPR) (the 'R' is sometimes attributed to "Reasoning" and sometimes to the second letter in "Propositional"). EPR formulas are of the form

$$\exists \mathbf{x} \forall \mathbf{y}.\ \varphi(\mathbf{x}, \mathbf{y})\ , \tag{9.47}$$

Recursion level	Line	Comment
0	2	QBF($\mathcal{Q}, \emptyset, 2$) is called.
0	6,7	The conditions in these lines do not hold.
0	8	$Q[2] = \exists$.
0	11	QBF($\mathcal{Q}, \{e = 0\}, 1$) is called first.
1	6	$\phi\|_{e=0} = (u)$.
1	8	$Q[1] = \forall$.
1	9	QBF($\mathcal{Q}, \{e = 0, u = 0\}, 0$) is called first.
2	6	$\phi\|_{e=0,u=0} =$ FALSE. return FALSE.
1	9	return FALSE.
0	11	QBF($\mathcal{Q}, \{e = 1\}, 1$) is called second.
1	6	$\phi\|_{e=1} = (\neg u)$.
1	8	$Q[1] = \forall$.
1	9	QBF($\mathcal{Q}, \{e = 1, u = 0\}, 0$) is called first.
2	6	$\phi\|_{e=1,u=0} =$ TRUE.
2	7	return TRUE.
1	9	QBF($\mathcal{Q}, \{e = 1, u = 1\}, 0$) is called second.
2	6	$\phi\|_{e=1,u=1} =$ FALSE; return FALSE.
1	9	return FALSE.
0	11	return FALSE.
0	3	return “Not valid”.

Fig. 9.4. A trace of Algorithm 9.3.1 when applied to (9.45)

where $\mathbf{x}, \mathbf{y}$ are sets of variables, and φ is a quantifier-free formula without function symbols (uninterpreted *predicate* symbols are allowed). For simplicity we assume there are no free variables.

It turns out that satisfiability of such formulas can be reduced to propositional SAT—hence the name—although it may lead to an exponential growth in the size of the formula (EPR is a NEXPTIME problem). After the transformation, we can apply a propositional SAT solver, as discussed in Chap. 2, in order to decide the satisfiability of the original formula.

A basic decision procedure for such formulas has several steps. We will use the following formula to demonstrate them:

$$\exists e_1 \exists e_2 \forall a_1 \forall a_2 .\ p(e_1, a_1) \vee q(e_2, a_2) . \tag{9.48}$$

1. Remove the existential quantifiers.[5] This results in

$$\forall a_1 \forall a_2 .\ p(e_1, a_1) \vee q(e_2, a_2) . \tag{9.49}$$

[5] This step is typically described in the literature as Skolemization, which we will only cover later in Definition 9.15. Since in EPR formulas the existential quantifiers are not in the scope of any universal quantifier, Skolemization results in functions without parameters, which are called *symbolic constants*. We avoid here this additional jargon by working directly with the existentially quantified variables as if they were such constants.

2. Grounding: eliminate each universal quantifier by forming a conjunction of instantiations of the suffix φ with every possible variable that was originally existentially quantified. In our example, this results in

$$\begin{array}{l}(p(e_1,e_1) \vee q(e_2,e_1)) \wedge \\ (p(e_1,e_1) \vee q(e_2,e_2)) \wedge \\ (p(e_1,e_2) \vee q(e_2,e_1)) \wedge \\ (p(e_1,e_2) \vee q(e_2,e_2))\ .\end{array} \tag{9.50}$$

   Given n existentially and m universally quantified variables, this step results in n^m conjuncts.
3. Encoding: at this point we are left with a Boolean combination of uninterpreted predicates. It is left to check whether we can assign a Boolean value to each such predicate under the constraint that two instances of the same predicate that are invoked with an identical vector of parameters are assigned the same value. The simplest way to do this is to encode each predicate with a propositional variable that corresponds to the signature of its parameters. For example, the predicate $p(e_1,e_2)$ is encoded with a propositional variable p_{12}. This leaves us with the propositional formula

$$\begin{array}{l}(p_{11} \vee q_{21}) \wedge \\ (p_{11} \vee q_{22}) \wedge \\ (p_{12} \vee q_{21}) \wedge \\ (p_{12} \vee q_{22})\ .\end{array} \tag{9.51}$$

4. SAT: invoke a propositional SAT solver. The formula obtained by the transformation above is equisatisfiable with the original formula. Our example is clearly satisfiable.

EPR is a natural formalism for modeling decision problems over relations. In particular, decision problems in set theory can be decided this way, since sets can be modeled with unary predicates. For example, suppose we want to check whether the following is satisfiable for arbitrary sets A, B, and C:

$$(A \cap B \neq \emptyset) \wedge (B \cap C \neq \emptyset) \wedge (A \cap C \neq \emptyset) \wedge (A \cap B \cap C = \emptyset)\ . \tag{9.52}$$

A unary predicate modeling a set variable is true if its argument belongs to the original set. For sets $A, B, C, \ldots$ we will denote by P_A, P_B, P_C unary predicates that represent them respectively, e.g., $P_A(x) = \text{TRUE} \iff x \in A$. Intersection is modeled with conjunction. Hence, the above formula can be reformulated in EPR as follows:

$$\begin{array}{l}(\exists x_1.\ P_A(x_1) \wedge P_B(x_1)) \wedge (\exists x_2.\ P_B(x_2) \wedge P_C(x_2)) \wedge (\exists x_3.\ P_A(x_3) \wedge P_C(x_3)) \\ \quad \wedge\ (\forall x_4.\ \neg P_A(x_4) \vee \neg P_B(x_4) \vee \neg P_C(x_4))\ .\end{array} \tag{9.53}$$

Transforming (9.53) to prenex normal form (see Sect. 9.2.1) reveals that it is an EPR formula, and following the procedure described above shows that it is satisfiable.

It is not hard to see that other set operations can also be modeled with Boolean operations between their respective predicates, e.g., union is modeled with disjunction, complementation with negation, and subtraction with a combination of intersection and complementation based on the equivalence $A \setminus B = A \cap B^c$. A relation such as $A \subseteq B$ can be modeled with a universal quantifier: $\forall x.\ P_A(x) \implies P_B(x)$. For a list of other uses of EPR the reader is referred to [224].

9.5 General Quantification

So far we have considered three special cases of quantified formulas for which the validity problem can be decided. In the case of QBF the decision procedure is based on splitting or on quantifier elimination, in the case of linear arithmetic it is based on quantifier elimination, and in the case of EPR it is based on a reduction to propositional logic. More generally, as long as the domain of the quantified variable is finite, or otherwise if there is a quantifier elimination algorithm, then the problem is decidable.

In this section we consider the general case, where enumeration and quantifier elimination are either impossible or just too computationally expensive. This is the only section in this book in which we consider a problem which is in general undecidable. Undecidability implies that there is no general algorithm that solves all cases. The best we can hope for is an algorithm that solves many useful cases. In fact general quantification is a key component in first-order **theorem proving** and as such receives a massive amount of attention in the literature. Automated first-order theorem provers have existed from the 1970s and continue to be developed and supported today—see the bibliographic notes in Sect. 9.7.

Since all the theories discussed in this book are first-order axiomatizable (i.e., the predicates in their signatures such as equality and inequality can be defined via axioms such as (1.35)), it follows that a general quantifier elimination algorithm, if it exists, can decide all of them. In other words, if there was a general and computationally competitive way to solve such formulas there would not be a need for the specialized decision procedures described thus far. For example, given an equality formula such as $x = y \wedge y = z \wedge x \neq z$, rather than invoking the congruence closure algorithm of Sect. 4.3, we would instead solve the same formula in conjunction with the equality axioms (1.35). Similarly, if the formula also contains uninterpreted functions we would add the congruence axiom (4.2). Such a universal mechanism does not exist, but there are heuristics that are effective in many cases.

The first step in attempting to solve general quantified formulas is transformation to the following normal form:

Definition 9.15 (Skolem normal form). *A formula is in* **Skolem normal form** *if it is in prenex normal form (see Sect. 9.2.1) and has only universal quantifiers.*

Every first-order formula φ can be converted into this form in linear time while not changing its satisfiability. This requires a procedure called **Skolemization**. Let $\psi = \exists x.\ P$ be a subformula in φ. Let $y_1, \ldots, y_n$ be universally quantified variables such that ψ is in their scope. To Skolemize, i.e., eliminate the existential quantification over x:

1. Remove the quantifier $\exists x$ from ψ.
2. Replace occurrences of x in P with $f_x(y_1, \ldots, y_n)$, where f_x is a new function symbol. It is sufficient to include those y variables that are actually used in P.

A special case occurs when x is not in the scope of any universally quantified variable, and hence $n = 0$. In this case replace x with a new "symbolic" constant c_x.

Example 9.16. Consider the formula

$$\forall y_1.\ \forall y_2.\ f(y_1, y_2) \wedge \exists x.\ (f(x, y_2) \wedge x < 0)\ . \tag{9.54}$$

After Skolemization, we have

$$\forall y_1.\ \forall y_2.\ f(y_1, y_2) \wedge (f(f_x(y_1, y_2), y_2) \wedge f_x(y_1, y_2) < 0)\ . \tag{9.55}$$

▟

Assuming we begin with a formula in prenex normal form, Skolemization is repeated for each existentially quantified subformula, which leaves us with a formula in Skolem normal form that includes new uninterpreted function symbols.

Instantiation

$\boxed{G}$ A typical scenario is one in which we attempt to prove the validity of a **ground formula** (i.e., unquantified formula) G based on sentences that represent axioms. Assuming the axioms are in Skolem normal form (in fact, typically axioms only use universal quantifiers to begin with), to prove G we can try to *instantiate* the universally quantified variables in order to reach a contradiction with $\neg G$. The instantiation, namely the choice of atoms to add, is based on heuristics. Consider for example the problem of proving that

$$\overbrace{f(h(a), b) = f(b, h(a))}^{G} \tag{9.56}$$

is implied by

$$\forall x.\ \forall y.\ f(x, y) = f(y, x)\ . \tag{9.57}$$

Presenting this as a satisfiability problem, we need to show that the following formula is unsatisfiable:

$$(\forall x.\ \forall y.\ f(x,y) = f(y,x)) \land f(h(a),b) \neq f(b,h(a))\ . \tag{9.58}$$

It is obvious that we need to instantiate x with $h(a)$ and y with b in order to reach a contradiction. This gives us the unsatisfiable ground formula

$$f(h(a),b) = f(b,h(a)) \land f(h(a),b) \neq f(b,h(a))\ . \tag{9.59}$$

In this case the terms with which we instantiated x and y already appeared in G. Furthermore, the single quantified atom was strong enough to give us exactly the predicate that we needed for the proof. Verification conditions that arise in practice can contain hundreds of sentences to choose from, and the ground formula G can be several megabytes in size. Hence we need to choose which quantified formulas to instantiate and what to instantiate them with, while keeping in mind that the goal is to derive ground terms that are most likely to contradict G. After each instantiation, we still need to check the ground formula. For this we can use the DPLL(T) framework that was studied in Chap. 3. Hence, the instantiation heuristic can be thought of as a wrapper around DPLL(T), invoking it with a different formula each time.

Let $(\forall \bar{x}.\ \psi)\ \land\ G$ be the formula that we attempt to prove to be unsatisfiable, where as before G denotes a ground formula. A naive approach is to instantiate $\bar{x}$ with all the ground terms in G of the same type. Such an approach leads to an exponential number in $|\bar{x}|$ of added ground terms. For example, if we consider once again (9.56) and (9.57), instantiating each of $\{x, y\}$ with a, b, $h(a)$, $f(h(a), b)$, and $f(b, h(a))$ yields 25 new predicates. Other than for the simplest of formulas this exponential growth makes it impractical. In the following we will describe a better heuristic that was implemented in a tool called Simplify [101], which was considered the state-of-the-art solver for many years. Modern solvers such as Z3 [92, 96] and CVC [121] use various optimizations over the basic techniques described here.

A typical instantiation heuristic attempts to generate predicates that refer to existing terms in G (a syntactic test), or to terms that are known to be equal to one of them (a semantic test). Most provers, Simplify included, use the congruence closure algorithm (see Sect. 4.3) for reasoning about equalities, and accordingly maintain a union-find data structure which represents the currently known equalities. This is called the **E-graph** in Simplify. A ground term is considered as *relevant* if it is represented as a node in this graph.

We will attempt to instantiate each variable in $\bar{x}$ with a ground term s such that at least one of the terms in ψ becomes relevant. More formally, we search for a sequence of ground terms $\bar{s}$ such that $|\bar{x}| = |\bar{s}|$ and $\psi[\bar{x} \leftarrow \bar{s}]$ contains at least one term in the E-graph. A more sophisticated algorithm may prioritize the substitutions by the number of such terms. Note that matching is not always possible. For example, no substitution $\bar{s}$ exists that can bring us from a subformula such as $g(x, y)$ or $f(x, 2)$ to the ground term $f(h(a), b)$.

Let us begin with a simple strategy:

- For each quantified formula of the form $\forall \bar{x}.\psi$, identify those subterms in ψ that contain references to all the variables in $\bar{x}$. These are called the

triggers. For example, in (9.57), both $f(x, y)$ and $f(y, x)$ can be triggers, as they contain all the quantified variables. If no single term contains all variables, then choose a set of terms that cover all quantified variables. Such terms are known as **multitriggers**.

$\boxed{tr}$ $\boxed{gr}$

- Try to **match** each trigger tr to an existing ground term gr in G. A trigger can be thought of as a pattern that has to be instantiated in order to reach an existing ground term. In the example above, matching $f(x, y)$ to $f(h(a), b)$ yields the substitution $\bar{s} = \{x \mapsto h(a), y \mapsto b\}$. We will later present an algorithm for performing such matches.
- Given a substitution $\bar{s}$, assign $G := G \wedge \psi[\bar{x} \leftarrow \bar{s}]$ and check the satisfiability of G.

Example 9.17. Consider

$$G := (b = c \implies f(h(a), g(c)) = f(g(b), h(a))) \,, \tag{9.60}$$

which we need to validate under the assumption that f is commutative, i.e.,

$$\forall x.\ \forall y.\ f(x, y) = f(y, x) \,. \tag{9.61}$$

Cast in terms of satisfiability, we need to prove the unsatisfiability of

$$(\forall x.\ \forall y.\ f(x, y) = f(y, x)) \wedge b = c \wedge f(h(a), g(c)) \neq f(g(b), h(a)) \,. \tag{9.62}$$

In the first step we identify the triggers: $f(x, y), f(y, x)$. Since in this case the triggers are symmetric we will focus only on the first one. We can match $f(x, y)$ to $f(h(a), g(c))$ with the substitution $\{x \mapsto h(a), y \mapsto g(c)\}$ or to $f(g(b), h(a))$ with $\{x \mapsto g(b), y \mapsto h(a)\}$. Now we check the satisfiability of

$$\begin{aligned} &(b = c \wedge f(h(a), g(c)) \neq f(g(b), h(a))) \\ \wedge\ & f(h(a), g(c)) = f(g(c), h(a)) \\ \wedge\ & f(g(b), h(a)) = f(h(a), g(b)) \,. \end{aligned} \tag{9.63}$$

This formula is unsatisfiable, which means that the instantiation was successful. In fact the first substitution is sufficient in this case, which raises the question of whether we should add all terms together or do it gradually. Different solvers may use different strategies in this respect. ▪

Frequently, however, the predicates necessary for proving unsatisfiability are not based on terms in the existing formula. Simplify has a more flexible matching algorithm, which exploits its current knowledge on equivalences among various terms, which is called **E-matching**. This technique is described next.

The E-Matching Algorithm

A simple matching algorithm of formulas is based on the syntactic similarity of a trigger (the pattern) tr and a ground term gr. More specifically, tr and *gr* have to be equivalent except in the locations of the quantified variables. For example, it can match a trigger $f(x, g(y))$ to a ground term $f(a, g(h(b)))$. E-matching is more flexible than this, as at each point in the matching process it considers not only a given subterm of *gr*, but also any ground term that is known to be equivalent to it and matches the pattern.

The E-matching algorithm appears in Algorithm 9.5.1. Its input is a 3-tuple $\langle tr, gr, sub\rangle$, where *tr* is a trigger, *gr* is a term to which we try to match *tr*, and *sub* is the current set of possible substitutions (initially empty). The output of this algorithm is a set of substitutions, each of which brings us from *tr* to *gr*, possibly by using congruence closure. More formally, if E denotes $\boxed{E}$ the equalities, then for each possible substitution $\alpha \in sub$, it holds that $E \models \alpha(tr) = gr$, where $\alpha(tr)$ denotes the substitution α applied to the trigger *tr*. For example, for $tr \doteq f(x)$ and $gr \doteq f(a)$, if $E \doteq \{a = b\}$, the value of *sub* at the end of the algorithm will be $\{x \mapsto a, x \mapsto b\}$.

When reading the algorithm, recall that *sub* is a set of substitutions, where a substitution is a mapping between some of the quantified variables and terms. The algorithm progresses by either adding new substitutions or augmenting existing ones with mappings of additional variables. The algorithm calls three auxiliary functions:

- $dom(\alpha)$ is the domain of the substitution α. For example, $dom(\{x \mapsto a, y \mapsto b\}) = \{x, y\}$.

The other two functions, *find* and *class*, are part of the standard interface of a congruence closure algorithm that is implemented with union-find:

- *find*(gr) returns the representative element of the class of *gr*. If two terms gr_1, gr_2 are such that $find(gr_1) = find(gr_2)$ then it means that they are equivalent.
- *class*(gr) returns the equivalence class of *gr*.

We now explain the algorithm line by line. In line 3 we handle the case in which *tr* is a variable x, by examining separately each element in the substitution set *sub*. The set returned is a union of two sets, depending on the element $\alpha \in sub$ that we examine. To understand these sets, consider these two illustrations:

- Let $\alpha \doteq \{y \mapsto c, z \mapsto d\}$ be a substitution in *sub*. We add the substitution $\{y \mapsto c, z \mapsto d, x \mapsto c\}$ to the returned set. In other words, we augment α with a mapping of x. We perform this augmentation only if $x \not\in dom(\alpha)$, where in this case $dom(\alpha) = \{y, z\}$.
- Let $\alpha \doteq \{y \mapsto c, x \mapsto d\}$ be a substitution in *sub* and let *gr* be a term. The returned set is the same substitution α, but only if $E \models d = gr$, i.e., d is an expression that is logically equivalent to *gr*, according to E. Otherwise α is discarded, since we do not want to have x mapped to two terms that are not known to be equal.

In line 4 we handle the case in which *tr* is a constant c. This case is simple: either the term to which we try to match is equivalent to c, or we return the empty set in the case where there is no match.

In line 7 we handle the case in which *tr* is a function with n terms $f(p_1, \ldots, p_n)$. First consider a simple case in which *gr* is a term with the same pattern, e.g., $f(gr_1, \ldots, gr_n)$. In this case the returned set is constructed recursively, by applying MATCH to each pair of arguments p_i, gr_i for $i \in [1..n]$. Since we are doing E-matching and not just matching, we need to consider terms that are known to be logically equivalent to *gr* that also have this pattern, regardless of whether *gr* itself has this pattern. This is the reason for the union in the last line of the algorithm. Although not explicitly stated in the algorithm, if matching one or more of the arguments *fails* (MATCH returns the empty set for this argument), then whatever MATCH has found for the other arguments is discarded from the result.

Algorithm 9.5.1: E-MATCHING

Input: Trigger *tr*, term *gr*, current substitution set *sub*
Output: Substitution set *sub* such that for each $\alpha \in sub$, $E \models \alpha(tr) = gr$.

```
1. function MATCH($tr, gr, sub$)
2.     if $tr$ is a variable $x$ then
3.         return
               $\{\alpha \cup \{x \mapsto gr\} \mid \alpha \in sub, x \notin dom(\alpha)\} \cup$
               $\{\alpha \mid \alpha \in sub, find(\alpha(x)) = find(gr)\}$
4.     if $tr$ is a constant $c$ then
5.         if $c \in class(gr)$ then return $sub$
6.         else return $\emptyset$
7.     if $tr$ is of the form $f(p_1, \ldots, p_n)$ then return
```

$$\bigcup_{f(gr_1,\ldots,gr_n)\in class(gr)} \begin{array}{l} \text{MATCH}(p_n, gr_n, \\ \quad \text{MATCH}(p_{n-1}, gr_{n-1}, \\ \qquad \ddots \\ \qquad \text{MATCH}(p_1, gr_1, sub)\ldots)) \end{array}$$

Example 9.18. Consider the formula

$$(\forall x.\ f(x) = x) \wedge (\forall y_1.\ \forall y_2.\ g(g(y_1, y_2), y_2) = y_2) \wedge g(f(g(a, b)), b) \neq b\ . \quad (9.64)$$

The triggers are $f(x)$ and $g(g(y_1, y_2), y_2)$. To match the first of those we run

$$\text{MATCH}(f(x), g(f(g(a, b)), b) \neq b, \emptyset)\ .$$

Since $f(x)$ is a function application we invoke line 7. The only relevant subterm in *gr* is $f(g(a, b))$, and the computation is

$$
\begin{array}{ll}
\quad \text{MATCH}(x, g(a,b), \emptyset) & \text{line 7} \\
= \{x \mapsto g(a,b)\} & \text{line 3}
\end{array}
$$

At this point the equivalence $f(g(a,b)) = g(a,b)$ is added to E. We now invoke E-matching on the second trigger $g(g(y_1, y_2), y_2)$. In line 7 we find that the candidate ground terms for the matching are $g(a,b)$ and $g(f(g(a,b)), b)$, as both are applications of the g function and are terms in gr. We leave the reader to run the algorithm in the case of the first candidate and conclude that it fails. As for the second term, the computation is:

$$
\begin{array}{ll}
= \text{MATCH}(y_2, b, \text{MATCH}(g(y_1, y_2), f(g(a,b)), \emptyset)) & \text{line 7} \\
= \text{MATCH}(y_2, b, \text{MATCH}(g(y_1, y_2), g(a,b), \emptyset)) & \text{line 7} \\
= \text{MATCH}(y_2, b, \text{MATCH}(y_2, b, \text{MATCH}(y_1, a, \emptyset))) & \text{line 7} \\
= \text{MATCH}(y_2, b, \text{MATCH}(y_2, b, \{y_1 \mapsto a\})) & \text{line 3} \\
= \text{MATCH}(y_2, b, \{y_1 \mapsto a, y_2 \mapsto b\}) & \text{line 3} \\
= \{y_1 \mapsto a, y_2 \mapsto b\} & \text{line 3} .
\end{array}
$$

Note the switch between $f(g(a,b))$ and $g(a,b)$: it happens in line 7, because these two terms are in the same equivalence class according to the E-graph.

Example 9.19. Consider a trigger $tr \doteq f(x, y, x)$. Let $gr \doteq g(a)$ be a term that according to E is equivalent to $f(a, b, a)$ and to $f(c, d, e)$. We call MATCH with $(tr, gr, \emptyset)$. The recursive computation of MATCH appears below.

$$
\begin{array}{l}
\quad \text{MATCH}(tr, gr, \emptyset) \\
= \text{MATCH}(x, a, \text{MATCH}(y, b, \text{MATCH}(x, a, \emptyset))) \cup \\
\quad \text{MATCH}(x, e, \text{MATCH}(y, d, \text{MATCH}(x, c, \emptyset))) \\
= \text{MATCH}(x, a, \text{MATCH}(y, b, \{x \mapsto a\})) \ \cup \ \text{MATCH}(x, e, \text{MATCH}(y, d, \{x \mapsto c\})) \\
= \text{MATCH}(x, a, \{x \mapsto a, y \mapsto b\}) \ \cup \ \text{MATCH}(x, e, \{x \mapsto c, y \mapsto d\}) \\
= \{x \mapsto a, y \mapsto b\} \ \cup \ \emptyset \\
= \{x \mapsto a, y \mapsto b\} \ .
\end{array}
$$

As expected, the term $f(c, d, e)$ does not contribute a substitution, because c and e, the terms to which x is supposed to be mapped, are not known to be equivalent in E.

Beyond the Congruence Closure

The E-matching algorithm searches for terms in the congruence closure of the terms, but clearly this is not always sufficient for a proof. Consider, for example, the following formula:

$$(\forall x.\ f(2x - x) < x) \wedge (f(a) \geq a) \ . \tag{9.65}$$

Even if we refer to the minus sign as any other function symbol when running the matching algorithm, without knowing that $2x - x = x$ it will get stuck, not finding a ground term to which it can match. Owing to complexity considerations, most theorem provers that handle quantifiers refrain from considering anything beyond the congruence closure when matching. The latter is relatively easy to do, because most provers use the congruence closure algorithm (see Sect. 4.3) and accordingly maintain a union-find data structure which represents the currently known equalities. Hence they can solve (9.64), but not (9.65). The solver Z3 mentioned earlier can, in fact, solve the above formula because it applies basic simplifications to expressions. Hence $2x - x$ will be simplified to x before matching begins. More complicated cases are still beyond its reach.

The story does not end here, although the chapter does. There are other complications that we refrain from describing here in detail. For example, note that instantiation may run into an infinite loop: matching will add predicates, and the terms in these new predicates create more matching opportunities, and there is no guarantee of convergence. Problem 9.12 shows an example of a formula that triggers this phenomenon. Theorem provers such as Simplify employ a cycle detection technique to prevent such cases, as well as various heuristics that order the triggers in a way that will make such cycles less frequent. Another technique worth mentioning is that of user assistance. Many provers can now read markings on terms inserted by the user, which guides the prover in choosing the right triggers. This can be helpful when without it the solver diverges and cannot find the right triggers to activate.

9.6 Problems

9.6.1 Warm-up Exercises

Problem 9.1 (example of forall reduction). Show that the equivalence

$$\exists e.\ \exists f.\ \forall u.(e \vee f \vee u) \equiv \exists e.\ \exists f.\ (e \vee f) \tag{9.66}$$

holds.

Problem 9.2 (expansion-based quantifier elimination). Is the following formula valid? Check by eliminating all quantifiers with expansion. Perform simplifications when possible.

$$\begin{aligned} \mathcal{Q} := {} & \forall x_1.\ \forall x_2.\ \forall x_3.\ \exists x_4. \\ & (x_1 \implies (x_2 \implies x_3)) \implies ((x_1 \wedge x_2 \implies x_3) \wedge (x_4 \vee x_1))\ . \end{aligned} \tag{9.67}$$

Problem 9.3 (EPR). Based on the procedure in Sect. 9.4,

1. Prove that $\exists x \forall y.\ p(x) \not\Leftrightarrow p(y)$ is unsatisfiable.
2. Solve to completion Eq. (9.52).
3. Prove the validity of $A \cup B \subseteq C \wedge B \setminus A \neq \emptyset \implies A \subset C$ for arbitrary sets A, B, and C.

9.6.2 QBF

Problem 9.4 (eliminating universal quantifiers from CNF). Let

$$\mathcal{Q} := \; Q[n]V[n] \ldots Q[2]V[2].\ \forall x.\ \phi\ , \tag{9.68}$$

where ϕ is a CNF formula. Let

$$\mathcal{Q}' := \; Q[n]V[n] \ldots Q[2]V[2].\ \phi'\ , \tag{9.69}$$

where ϕ' is the same as ϕ except that x and $\neg x$ are erased from all clauses.

1. Prove that $\mathcal{Q}$ and $\mathcal{Q}'$ are logically equivalent if ϕ does not contain tautology clauses.
2. Show an example where $\mathcal{Q}$ and $\mathcal{Q}'$ are not logically equivalent if ϕ contains tautology clauses.

Problem 9.5 (modeling: the diameter problem). QBFs can be used for finding the longest shortest path of any state from an initial state in a finite state machine. More formally, what we would like to find is defined as follows:

Definition 9.20 (initialized diameter of a finite state machine). *The* initialized diameter *of a state machine is the smallest $k \in \mathbb{N}$ for which every node reachable in $k+1$ steps can also be reached in k steps or fewer.*

Our assumption is that the finite state machine is too large to represent or explore explicitly: instead, it is given to us implicitly in the form of a transition system, in a similar fashion to the chess problem that was described in Sect. 9.1.1.

For the purpose of this problem, a finite transition system is a tuple $\langle S, I, T \rangle$, where S is a finite set of states, each of which is a valuation of a finite set of variables $(V \cup V' \cup In)$. V is the set of state variables and V' is the corresponding set of next-state variables. In is the set of input variables. I is a predicate over V defining the initial states, and T is a transition function that maps each variable $v \in V'$ to a predicate over $V \cup I$.

An example of a class of state machines that are typically represented in this manner is digital circuits. The initialized diameter of a circuit is important in the context of formal verification: it represents the largest depth to which one needs to search for an error state.

Given a transition system M and a natural k, formulate with QBF the problem of whether k is the diameter of the graph represented by M. Introduce proper notation in the style of the chess problem that was described in Sect. 9.1.1.

Problem 9.6 (search-based QBFs). Apply Algorithm 9.3.1 to the formula

$$\mathcal{Q} := \; \forall u.\ \exists e.\ (e \vee u)(\neg e \vee \neg u)\ . \tag{9.70}$$

Show a trace of the algorithm as in Fig. 9.4.

Problem 9.7 (QBFs and resolution). Using resolution, check whether the formula

$$\mathcal{Q} := \ \forall u.\ \exists e.\ (e \vee u)(\neg e \vee \neg u) \tag{9.71}$$

is valid.

Problem 9.8 (projection by resolution). Show that the pairwise resolution suggested in Sect. 9.2.3 results in a projection as defined in Definition 9.7.

Problem 9.9 (QBF refutations). Let

$$\mathcal{Q} = Q[n]V[n] \ldots Q[1]V[1].\ \phi\ , \tag{9.72}$$

where ϕ is in CNF and $\mathcal{Q}$ is FALSE, i.e., $\mathcal{Q}$ is not valid. Propose a *proof format* for such QBFs that is generally applicable, i.e., allows us to give a proof for any QBF that is not valid (similarly to the way that binary-resolution proofs provide a proof format for propositional logic).

Problem 9.10 (QBF models). Let

$$\mathcal{Q} = Q[n]V[n] \ldots Q[1]V[1].\ \phi\ , \tag{9.73}$$

where ϕ is in CNF and $\mathcal{Q}$ is TRUE, i.e., $\mathcal{Q}$ is valid. In contrast to the quantifier-free SAT problem, we cannot provide a satisfying assignment to all variables that convinces us of the validity of $\mathcal{Q}$.

(a) Propose a *proof format* for valid QBFs.
(b) Provide a proof for the formula in Problem 9.7 using your proof format.
(c) Provide a proof for the following formula:

$$\forall u.\ \exists e.\ (u \vee \neg e)(\neg u \vee e)\ .$$

9.6.3 EPR

Problem 9.11 (direct reduction to propositional logic). We saw how set logic can be decided with EPR. Show a fragment of this logic that can be decided by a straight mapping of the set variables $(A, B, \ldots)$ to propositional variables, i.e., without instantiation.

9.6.4 General Quantification

Problem 9.12 (quantification loop). Consider the formula

$$(\forall x.\ f(x) = f(g(x))) \wedge f(g(a)) = a\ . \tag{9.74}$$

Assume that we choose $f(x)$ as a trigger. Show how this may lead to a loop in which new terms are added indefinitely.

Problem 9.13 (instantiating the congruence axiom). While assuming the congruence axiom (4.2), prove the validity of (11.3) with quantifier instantiation. Note that (4.2) introduces an implication, which means that either case splitting or other standard inference rules for propositional logic may be necessary.

Problem 9.14 (simple matching). Prove that (9.64) is unsatisfiable with simple matching (not e-matching).

9.7 Bibliographic Notes

The quantifier elimination procedure based on Fourier–Motzkin that was presented in this chapter is not the most efficient known, although it is probably the simplest. We refer the reader to works by Ferrante and Rackoff [112], Loos and Weispfenning [183], and Bjørner [39] for alternatives and a survey.

Stockmeyer and his PhD advisor at MIT, Meyer, identified the QBF problem as PSPACE-complete as part of their work on the polynomial hierarchy [264, 265]. The idea of solving QBF by alternating between resolution and eliminating universally quantified variables from CNF clauses was proposed by Büning, Karpinski, and Flögel [59]. The resolution part was termed **Q-resolution** (recall that the original SAT-solving technique developed by Davis and Putnam was based on resolution [88]).

There are many similarities in the research directions of SAT and QBF, and in fact there are researchers who are active in both areas. The positive impact that annual competitions and benchmark repositories have had on the development of SAT solvers has led to similar initiatives for the QBF problem (e.g., see QBFLIB [128], which includes thousands of benchmarks and a collection of more than 50 QBF solvers). Further, similarly to the evidence provided by propositional SAT solvers (namely a satisfying assignment or a resolution proof), many QBF solvers now provide a **certificate** of the validity or invalidity of a QBF instance [160] (also see Problems 9.9 and 9.10). Not surprisingly, there is a huge difference between the size of problems that can be solved in a reasonable amount of time by the best QBF solvers (thousands or a few tens of thousands of variables) and the size of problems that can be solved by the best SAT solvers (several hundreds of thousands or even a few millions of variables). It turns out that the exact encoding of a given problem can have a very significant impact on the ability to solve it—see, for example, the work by Sabharwal et al. [250]. The formulation of the chess problem in this chapter is inspired by that paper.

The research in the direction of applying propositional SAT techniques to QBFs, such as adding conflict and blocking clauses and the search-based method, is mostly based on work by Zhang and Malik [296, 297]. Quantifier expansion is folk knowledge, and was used for efficient QBF solving by, for example, Biere [29]. A similar type of expansion, called Shannon expansion, was used for one-alternation QBFs in the context of symbolic *model checking* with BDDs—see, for example, the work of McMillan [192]. Variants of BDDs were used for QBF solving in [125].

Presburger arithmetic was defined by Mojzesz Presburger, who published his work, in German, in 1929 [232]. At that time, Hilbert considered Presburger's decidability result as a major step towards full mechanization of mathematics (full mechanization of mathematics was the ultimate goal of many mathematicians, such as Leibniz and Peano, much earlier than that), which later proved to be an impossibility owing to Gödel's incompleteness theorem. Gödel's result refers to **Peano arithmetic**, which is the same as Presburger arithmetic with the addition of multiplication. One of the first mechanical deduction systems was an implementation of Presburger's algorithm on the Johnniac, a vacuum tube computer, in 1954. At the time, it was considered a major achievement that the program was able to show that the sum of two even numbers is an even number.

Two well-known approaches for solving Presburger formulas, in addition to that based on the Omega test that was mentioned in this chapter, are due to Cooper [78] and the family of methods based on finite automata and model checking: see the article by Wolper and Boigelot [287] and the publications regarding the LASH system, as well as Ganesh, Berezin, and Dill's survey and empirical comparison [117] of such methods when applied to unquantified Presburger formulas.

The problem of deciding quantified formulas over nonlinear real arithmetic is decidable, although a description of a decision procedure for this problem is not within the scope of this book. A well-known decision procedure for this theory is cylindrical algebraic decomposition (CAD). A comparison of CAD with other techniques can be found in [104]. Several tutorials on CAD can be found on the Web.

General quantification is a key element in first-order theorem proving, which is a topic of multiple books (e.g., the two-volume handbook of automated reasoning [244]), conferences, university courses, etc. There are many tools, *theorem provers*, which, like SAT and SMT tools, compete annually in a competition called CASC [271]. Currently the tool VAMPIRE [169], most of which was developed by Andrei Voronkov, is the best overall first-order theorem prover. A tool called IPROVER, developed by Konstantin Korovin, is currently the best EPR solver. E-matching was introduced in the SIMPLIFY theorem prover [101]. A discussion of modern incarnations of this technique, various optimizations and implementation details can be found in, e.g., [96] and [121].

9.8 Glossary

The following symbols were used in this chapter:

Symbol	Refers to ...	First used on page ...
$\forall, \exists$	The universal and existential quantification symbols	199
n	The number of quantifiers	205
N	The total number of variables (not only those existentially quantified)	208
$\phi\|_{\hat{v}}$	A simplification of ϕ based on the assignments in $\hat{v}$	212
G	A ground formula whose validity we attempt to prove	216
tr	A trigger: a subformula with universally quantified variables	218
gr	A ground term that we attempt to match with a trigger	218
E	A set of equivalences assumed by the solver	219

10

Deciding a Combination of Theories

10.1 Introduction

The decision procedures that we have studied so far focus on one specific theory. Verification conditions that arise in practice, however, frequently mix expressions from several theories. Consider the following examples:

- A combination of linear arithmetic and uninterpreted functions:

$$(x_2 \geq x_1) \wedge (x_1 - x_3 \geq x_2) \wedge (x_3 \geq 0) \wedge f(f(x_1) - f(x_2)) \neq f(x_3) \quad (10.1)$$

- A combination of bit vectors and uninterpreted functions:

$$f(a[32], b[1]) = f(b[32], a[1]) \wedge a[32] = b[32] \quad (10.2)$$

- A combination of arrays and linear arithmetic:

$$x = v\{i \longleftarrow e\}[j] \wedge y = v[j] \wedge x > e \wedge x > y \quad (10.3)$$

In this chapter, we cover the popular **Nelson–Oppen** combination method. This method assumes that we have a decision procedure for each of the theories involved. The Nelson–Oppen combination method permits the decision procedures to communicate information with one another in a way that guarantees a sound and complete decision procedure for the combined theory.

10.2 Preliminaries

Let us recall several basic definitions and conventions that should be covered in any basic course on mathematical logic (see also Sect. 1.4). We assume a basic familiarity with first-order logic here.

First-order logic is a baseline for defining various restrictions thereof, which are called **theories**. It includes:

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_10

- Variables
- **Logical symbols** that are shared by all theories, such as the Boolean operators ($\wedge$, $\vee$, ...), quantifiers ($\forall$, $\exists$), and parentheses
- **Nonlogical symbols**, namely function and predicate symbols, that are uniquely specified for each theory
- Syntax

It is common to consider the equality sign as a logical symbol rather than a predicate that is specific to a theory, since first-order theories without this symbol are rarely considered. We follow this convention in this chapter.

A first-order theory is defined by a set of sentences (first-order formulas in which all variables are quantified). It is common to represent such sets by a set of axioms, with the implicit meaning that the theory is the set of sentences that are derivable from these axioms. In such a case, we can talk about the "axioms of the theory". Axioms that define a theory are called the **nonlogical axioms**, and they come in addition to the axioms that define the logical symbols, which, correspondingly, are called the **logical axioms**.

$\boxed{\Sigma}$

A theory is defined over a signature Σ, which is a set of nonlogical symbols (i.e., function and predicate symbols). If T is such a theory, we say it is a Σ-theory. Let T be a Σ-theory. A Σ-formula φ is **T-satisfiable** if there exists an interpretation that satisfies both φ and T. A Σ-formula φ is **T-valid**, denoted $T \models \varphi$, if all interpretations that satisfy T also satisfy φ. In other words, such a formula is T-valid if it can be derived from the T axioms and the logical axioms.

$\boxed{T \models \varphi}$

$\boxed{\oplus}$

Definition 10.1 (theory combination). *Given two theories T_1 and T_2 with signatures Σ_1 and Σ_2, respectively, the* theory combination *$T_1 \oplus T_2$ is a $(\Sigma_1 \cup \Sigma_2)$-theory defined by the axiom set $T_1 \cup T_2$.*

The generalization of this definition to n theories rather than two theories is straightforward.

Definition 10.2 (the theory combination problem). *Let φ be a $\Sigma_1 \cup \Sigma_2$ formula. The* theory combination *problem is to decide whether φ is $T_1 \oplus T_2$-valid. Equivalently, the problem is to decide whether the following holds:*

$$T_1 \oplus T_2 \models \varphi \,. \tag{10.4}$$

The theory combination problem is undecidable for arbitrary theories T_1 and T_2, even if T_1 and T_2 themselves are decidable. Under certain restrictions on the combined theories, however, the problem becomes decidable. We discuss these restrictions later on.

An important notion required in this chapter is that of a convex theory.

Definition 10.3 (convex theory). *A Σ-theory T is* convex *if for every conjunctive Σ-formula φ*

$$\begin{array}{l}(\varphi \implies \bigvee_{i=1}^{n} x_i = y_i) \text{ is } T\text{-valid for some finite } n > 1 \implies \\ (\varphi \implies x_i = y_i) \text{ is } T\text{-valid for some } i \in \{1, \ldots, n\}\ ,\end{array} \qquad (10.5)$$

where x_i, y_i, for $i \in \{1, \ldots, n\}$, are some variables.

In other words, in a convex theory T, if a formula T-implies a disjunction of equalities, it also T-implies at least one of these equalities separately.

Example 10.4. Examples of convex and nonconvex theories include:

- Linear arithmetic over $\mathbb{R}$ is convex. A conjunction of linear arithmetic predicates defines a set of values which can be empty, a singleton, as in

$$x \le 3 \land x \ge 3 \implies x = 3\ , \qquad (10.6)$$

  or infinitely large, and hence it implies an infinite disjunction. In all three cases, it fits the definition of convexity.
- Linear arithmetic over $\mathbb{Z}$ is not convex. For example, while

$$x_1 = 1 \land x_2 = 2 \land 1 \le x_3 \land x_3 \le 2 \implies (x_3 = x_1 \lor x_3 = x_2) \qquad (10.7)$$

  holds, neither

$$x_1 = 1 \land x_2 = 2 \land 1 \le x_3 \land x_3 \le 2 \implies x_3 = x_1 \qquad (10.8)$$

  nor

$$x_1 = 1 \land x_2 = 2 \land 1 \le x_3 \land x_3 \le 2 \implies x_3 = x_2 \qquad (10.9)$$

  holds.
- The conjunctive fragment of equality logic is convex. A conjunction of equalities and disequalities defines sets of variables that are equal (equality sets) and sets of variables that are different. Hence, it implies any equality between variables in the same equality set separately. Convexity follows.

▟

Many theories used in practice are in fact nonconvex, which, as we shall soon see, makes them computationally harder to combine with other theories.

10.3 The Nelson–Oppen Combination Procedure

10.3.1 Combining Convex Theories

The Nelson–Oppen combination procedure solves the theory combination problem (see Definition 10.2) for theories that comply with several restrictions.

Definition 10.5 (Nelson–Oppen restrictions). *In order for the Nelson–Oppen procedure to be applicable, the theories $T_1, \ldots, T_n$ should comply with the following restrictions:*

1. *$T_1, \ldots, T_n$ are quantifier-free first-order theories with equality.*
2. *There is a decision procedure for each of the theories $T_1, \ldots, T_n$.*
3. *The signatures are disjoint, i.e., for all $1 \leq i < j \leq n$, $\Sigma_i \cap \Sigma_j = \emptyset$.*
4. *$T_1, \ldots, T_n$ are theories that are interpreted over an infinite domain (e.g., linear arithmetic over $\mathbb{R}$, but not the theory of finite-width bit vectors).*

There are extensions to the basic Nelson–Oppen procedure that overcome each of these restrictions, some of which are covered in the bibliographic notes at the end of this chapter.

Algorithm 10.3.1 is the Nelson–Oppen procedure for combinations of convex theories. It accepts a formula φ, which must be a conjunction of literals, as input. In general, adding disjunction to a convex theory makes it nonconvex. Extensions of convex theories with disjunctions can be supported with the extension to nonconvex theories that we present later on or, alternatively, with the methods described in Chap. 3, which are based on combining a decision procedure for the theory with a SAT solver.

The first step of Algorithm 10.3.1 relies on the idea of **purification**. Purification is a satisfiability-preserving transformation of the formula, after which each atom is from a specific theory. In this case, we say that all the atoms are **pure**. More specifically, given a formula φ, purification generates an equisatisfiable formula φ' as follows:

1. Let $\varphi' := \varphi$.
2. For each "alien" subexpression ϕ in φ':
   (a) Replace ϕ with a new auxiliary variable a_ϕ
   (b) Constrain φ' with $a_\phi = \phi$

Example 10.6. Given the formula

$$\varphi := \; x_1 \leq f(x_1)\;, \tag{10.10}$$

which mixes arithmetic and uninterpreted functions, purification results in

$$\varphi' := \; x_1 \leq a \wedge a = f(x_1)\;. \tag{10.11}$$

In φ', all atoms are pure: $x_1 \leq a$ is an arithmetic formula, and $a = f(x_1)$ belongs to the theory of equalities with uninterpreted functions. ▟

$\boxed{F_i}$

After purification, we are left with a set of pure expressions $F_1, \ldots, F_n$ such that:

1. For all i, F_i belongs to theory T_i and is a conjunction of T_i-literals.
2. Shared variables are allowed, i.e., it is possible that for some i, j, $1 \leq i < j \leq n$, $var(F_i) \cap var(F_j) \neq \emptyset$.
3. The formula φ is satisfiable in the combined theory if and only if $\bigwedge_{i=1}^{n} F_i$ is satisfiable in the combined theory.

Algorithm 10.3.1: NELSON–OPPEN-FOR-CONVEX-THEORIES

Input: A convex formula φ that mixes convex theories, with restrictions as specified in Definition 10.5

Output: "Satisfiable" if φ is satisfiable, and "Unsatisfiable" otherwise

1. *Purification:* Purify φ into $F_1, \ldots, F_n$.
2. Apply the decision procedure for T_i to F_i. If there exists i such that F_i is unsatisfiable in T_i, return "Unsatisfiable".
3. *Equality propagation:* If there exist i, j such that F_i T_i-implies an equality between variables of φ that is not T_j-implied by F_j, add this equality to F_j and go to step 2.
4. Return "Satisfiable".

Example 10.7. Consider the formula

$$\begin{array}{c}(f(x_1, 0) \geq x_3) \wedge (f(x_2, 0) \leq x_3) \wedge \\ (x_1 \geq x_2) \wedge (x_2 \geq x_1) \wedge \\ (x_3 - f(x_1, 0) \geq 1)\ ,\end{array} \tag{10.12}$$

which mixes linear arithmetic and uninterpreted functions. Purification results in

$$\begin{array}{l}(a_1 \geq x_3) \wedge (a_2 \leq x_3) \wedge (x_1 \geq x_2) \wedge (x_2 \geq x_1) \wedge (x_3 - a_1 \geq 1) \wedge \\ (a_0 = 0) \wedge \\ (a_1 = f(x_1, a_0)) \wedge \\ (a_2 = f(x_2, a_0))\ .\end{array} \tag{10.13}$$

In fact, we applied a small optimization here, assigning both instances of the constant "0" to the same auxiliary variable a_0. Similarly, both instances of the term $f(x_1, 0)$ have been mapped to a_1 (purification, as described earlier, assigns them to separate auxiliary variables).

The top part of Table 10.1 shows the formula (10.13) divided into the two pure formulas F_1 and F_2. The first is a linear arithmetic formula, whereas the second is a formula in the theory of equalities with uninterpreted functions (EUF). Neither F_1 nor F_2 is independently contradictory, and hence we proceed to step 3. With a decision procedure for linear arithmetic over the reals, we infer $x_1 = x_2$ from F_1, and propagate this fact to the other theory (i.e., we add this equality to F_2). We can now deduce $a_1 = a_2$ in T_2, and propagate this equality to F_1. From this equality, we conclude $a_1 = x_3$ in T_1, which is a contradiction to $x_3 - a_1 \geq 1$ in T_1. ▟

Example 10.8. Consider the following formula, which mixes linear arithmetic and uninterpreted functions:

F_1 (arithmetic over $\mathbb{R}$)	F_2 (EUF)
$a_1 \geq x_3$	$a_1 = f(x_1, a_0)$
$a_2 \leq x_3$	$a_2 = f(x_2, a_0)$
$x_1 \geq x_2$	
$x_2 \geq x_1$	
$x_3 - a_1 \geq 1$	
$a_0 = 0$	
$\star\ x_1 = x_2$	$x_1 = x_2$
$a_1 = a_2$	$\star\ a_1 = a_2$
$\star\ a_1 = x_3$	
$\star$ FALSE	

Table 10.1. Progress of the Nelson–Oppen combination procedure starting from the purified formula (10.13). The equalities beneath the middle horizontal line result from step 3 of Algorithm 10.3.1. An equality is marked with a "$\star$" if it was inferred within the respective theory

$$(x_2 \geq x_1) \wedge (x_1 - x_3 \geq x_2) \wedge (x_3 \geq 0) \wedge (f(f(x_1) - f(x_2)) \neq f(x_3)) \ . \tag{10.14}$$

Purification results in

$$\begin{array}{l} (x_2 \geq x_1) \ \wedge \ (x_1 - x_3 \geq x_2) \ \wedge \ (x_3 \geq 0) \ \wedge \ (f(a_1) \neq f(x_3)) \wedge \\ (a_1 = a_2 - a_3) \wedge \\ (a_2 = f(x_1)) \wedge \\ (a_3 = f(x_2)) \ . \end{array} \tag{10.15}$$

The progress of the equality propagation step, until the detection of a contradiction, is shown in Table 10.2.

▟

10.3.2 Combining Nonconvex Theories

Next, we consider the combination of nonconvex theories (or of convex theories together with theories that are nonconvex). First, consider the following example, which illustrates that Algorithm 10.3.1 may fail if one of the theories is not convex:

$$(1 \leq x) \ \wedge \ (x \leq 2) \ \wedge \ p(x) \wedge \neg p(1) \wedge \neg p(2) \ , \tag{10.16}$$

where $x \in \mathbb{Z}$.

Equation (10.16) mixes linear arithmetic over the integers and equalities with uninterpreted predicates. Linear arithmetic over the integers, as demonstrated in Example 10.4, is not convex. Purification results in

$$\begin{array}{l} 1 \leq x \ \wedge \ x \leq 2 \ \wedge \ p(x) \ \wedge \ \neg p(a_1) \ \wedge \ \neg p(a_2) \wedge \\ a_1 = 1 \wedge \\ a_2 = 2 \end{array} \tag{10.17}$$

F_1 (arithmetic over $\mathbb{R}$)	F_2 (EUF)
$x_2 \geq x_1$	$f(a_1) \neq f(x_3)$
$x_1 - x_3 \geq x_2$	$a_2 = f(x_1)$
$x_3 \geq 0$	$a_3 = f(x_2)$
$a_1 = a_2 - a_3$	
$\star\ x_3 = 0$	
$\star\ x_1 = x_2$	$x_1 = x_2$
$a_2 = a_3$	$\star\ a_2 = a_3$
$\star\ a_1 = 0$	
$\star\ a_1 = x_3$	$a_1 = x_3$
	FALSE

Table 10.2. Progress of the Nelson–Oppen combination procedure starting from the purified formula (10.15)

F_1 (arithmetic over $\mathbb{Z}$)	F_2 (EUF)
$1 \leq x$	$p(x)$
$x \leq 2$	$\neg p(a_1)$
$a_1 = 1$	$\neg p(a_2)$
$a_2 = 2$	

Table 10.3. The two pure formulas corresponding to (10.16) are independently satisfiable and do not imply any equalities. Hence, Algorithm 10.3.1 returns "Satisfiable"

Table 10.3 shows the partitioning of the predicates in the formula (10.17) into the two pure formulas F_1 and F_2. Note that both F_1 and F_2 are individually satisfiable, and neither implies any equalities in its respective theory. Hence, Algorithm 10.3.1 returns "Satisfiable" even though the original formula is unsatisfiable in the combined theory.

The remedy to this problem is to consider not only implied equalities, but also implied *disjunctions* of equalities. Recall that there is a finite number of variables, and hence of equalities and disjunctions of equalities, which means that computing these implications is feasible. Given such a disjunction, the problem is split into as many parts as there are disjuncts, and the procedure is called recursively. For example, in the case of the formula (10.16), F_1 implies $x = 1 \vee x = 2$. We can therefore split the problem into two, considering separately the case in which $x = 1$ and the case in which $x = 2$. Algorithm 10.3.2 merely adds one step (step 4) to Algorithm 10.3.1: the step that performs this split.

Algorithm 10.3.2: NELSON–OPPEN

Input: A formula φ that mixes theories, with restrictions as specified in Definition 10.5

Output: "Satisfiable" if φ is satisfiable, and "Unsatisfiable" otherwise

1. *Purification:* Purify φ into $\varphi' := F_1, \ldots, F_n$.
2. Apply the decision procedure for T_i to F_i. If there exists i such that F_i is unsatisfiable, return "Unsatisfiable".
3. *Equality propagation:* If there exist i, j such that F_i T_i-implies an equality between variables of φ that is not T_j-implied by F_j, add this equality to F_j and go to step 2.
4. *Splitting:* If there exists i such that
   - $F_i \implies (x_1 = y_1 \vee \cdots \vee x_k = y_k)$ and
   - $\forall j \in \{1, \ldots, k\}.\ F_i \;\not\!\!\!\implies x_j = y_j$,

   then apply NELSON–OPPEN recursively to

   $$\varphi' \wedge x_1 = y_1, \ldots, \varphi' \wedge x_k = y_k\ .$$

   If any of these subproblems is satisfiable, return "Satisfiable". Otherwise, return "Unsatisfiable".
5. Return "Satisfiable".

F_1 (arithmetic over $\mathbb{Z}$)	F_2 (EUF)
$1 \leq x$	$p(x)$
$x \leq 2$	$\neg p(a_1)$
$a_1 = 1$	$\neg p(a_2)$
$a_2 = 2$	
$\star\ x = 1 \vee x = 2$	

Table 10.4. The disjunction of equalities $x = a_1 \vee x = a_2$ is implied by F_1. Algorithm 10.3.2 splits the problem into the subproblems described in Tables 10.5 and 10.6, both of which return "Unsatisfiable"

Example 10.9. Consider the formula (10.16) again. Algorithm 10.3.2 infers $(x = 1 \vee x = 2)$ from F_1, and splits the problem into two subproblems, as illustrated in Tables 10.4–10.6.

F_1 (arithmetic over $\mathbb{Z}$)	F_2 (EUF)
$1 \leq x$	$p(x)$
$x \leq 2$	$\neg p(a_1)$
$a_1 = 1$	$\neg p(a_2)$
$a_2 = 2$	
$x = 1$	
$\star\ x = a_1$	$x = a_1$
	FALSE

Table 10.5. The case $x = a_1$ after the splitting of the problem in Table 10.4

F_1 (arithmetic over $\mathbb{Z}$)	F_2 (EUF)
$1 \leq x$	$p(x)$
$x \leq 2$	$\neg p(a_1)$
$a_1 = 1$	$\neg p(a_2)$
$a_2 = 2$	
$x = 2$	
$\star\ x = a_2$	$x = a_2$
	FALSE

Table 10.6. The case $x = a_2$ after the splitting of the problem in Table 10.4

10.3.3 Proof of Correctness of the Nelson–Oppen Procedure

We now prove the correctness of Algorithm 10.3.1 for convex theories and for conjunctions of theory literals. The generalization to Algorithm 10.3.2 is not hard. Without proof, we rely on the fact that $\bigwedge_i F_i$ is equisatisfiable with φ.

Theorem 10.10. *Algorithm 10.3.1 returns "Unsatisfiable" if and only if its input formula φ is unsatisfiable in the combined theory.*

Proof. Without loss of generality, we can restrict the proof to the combination of two theories T_1 and T_2.

($\Rightarrow$, Soundness) Assume that φ is satisfiable in the combined theory. We are going to show that this contradicts the possibility that Algorithm 10.3.2 returns "Unsatisfiable". Let α be a satisfying assignment of φ. Let A be the set of auxiliary variables added as a result of the purification step (step 1). As $\bigwedge_i F_i$ and φ are equisatisfiable in the combined theory, we can extend α to an assignment α' that includes also the variables A.

Lemma 10.11. *Let φ be satisfiable. After each loop iteration, $\bigwedge_i F_i$ is satisfiable in the combined theory.*

Proof. The proof is by induction on the number of loop iterations. Denote by F_i^j the formula F_i after iteration j.

Base case. For $j = 0$, we have $F_i^j = F_i$, and, thus, a satisfying assignment can be constructed as described above.

Induction step. Assume that the claim holds up to iteration j. We shall show the correctness of the claim for iteration $j + 1$. For any equality $x = y$ that is added in step 3, there exists an i such that $F_i^j \implies x = y$ in T_i. Since $\alpha' \models F_i^j$ in T_i by the hypothesis, clearly, $\alpha' \models x = y$ in T_i. Since for all i it holds that $\alpha' \models F_i^j$ in T_i, then for all i it holds that $\alpha' \models F_i \wedge x = y$ in T_i. Hence, in step 2, the algorithm will *not* return "Unsatisfiable". ▟

($\Leftarrow$, Completeness) First, observe that Algorithm 10.3.1 always terminates, as there are only finitely many equalities over the variables in the formula. It is left to show that the algorithm gives the answer "Unsatisfiable". We now record a few observations about Algorithm 10.3.1. The following observation is simple to see:

$\boxed{F_i'}$

Lemma 10.12. *Let F_i' denote the formula F_i upon termination of Algorithm 10.3.1. Upon termination with the answer "Satisfiable", any equality between φ's variables that is implied by any of the F_i' is also implied by all F_j' for any j.*

We need to show that, if φ is unsatisfiable, Algorithm 10.3.1 returns "Unsatisfiable". Assume falsely that it returns "Satisfiable".

Let $E_1, \ldots, E_m$ be a set of equivalence classes of the variables in φ such that x and y are in the same class if and only if F_1' implies $x = y$ in T_1. Owing to Lemma 10.12, $x, y \in E_i$ for some i if and only if $x = y$ is T_2-implied by F_2'.

For $i \in \{1, \ldots, m\}$, let r_i be an element of E_i (a *representative* of that set).

$\boxed{\Delta}$

We now define a constraint Δ that forces all variables that are not implied to be equal to be different:

$$\Delta \doteq \bigwedge_{i \neq j} r_i \neq r_j \ . \tag{10.18}$$

Lemma 10.13. *Given that both T_1 and T_2 have an infinite domain and are convex, Δ is T_1-consistent with F_1' and T_2-consistent with F_2'.*

Informally, this lemma can be shown to be correct as follows: Let x and y be two variables that are not implied to be equal. Owing to convexity, they do not have to be equal to satisfy F_i'. As the domain is infinite, there are always values left in the domain that we can choose in order to make x and y different.

Using Lemma 10.13, we argue that there are satisfying assignments α_1 and α_2 for $F_1' \wedge \Delta$ and $F_2' \wedge \Delta$ in T_1 and T_2, respectively. These assignments are **maximally diverse**, i.e., any two variables that are assigned equal values by either α_1 or α_2 *must* be equal.

Given this property, it is easy to build a mapping M (an isomorphism) from domain elements to domain elements such that $\alpha_2(x)$ is mapped to $\alpha_1(x)$ for any variable x (this is not necessarily possible unless the assignments are maximally diverse).

As an example, let F_1 be $x = y$ and F_2 be $F(x) = G(y)$. The only equality implied is $x = y$, by F_1. This equality is propagated to T_2, and thus both F_1' and F_2' imply this equality. Possible variable assignments for $F_1' \wedge \varDelta$ and $F_2' \wedge \varDelta$ are

$$\begin{aligned} \alpha_1 &= \{x \mapsto \mathcal{D}_1, y \mapsto \mathcal{D}_1\} \ , \\ \alpha_2 &= \{x \mapsto \mathcal{D}_2, y \mapsto \mathcal{D}_2\} \ , \end{aligned} \tag{10.19}$$

where $\mathcal{D}_1$ and $\mathcal{D}_2$ are some elements from the domain. This results in an isomorphism M such that $M(\mathcal{D}_1) = \mathcal{D}_2$.

Using the mapping M, we can obtain a model α' for $F_1' \wedge F_2'$ in the combined theory by adjusting the interpretation of the symbols in F_2' appropriately. This is always possible, as T_1 and T_2 do not share any nonlogical symbols.

Continuing our example, we construct the following interpretation for the nonlogical symbols F and G:

$$F(\mathcal{D}_1) = \mathcal{D}_3 \ , \quad G(\mathcal{D}_1) = \mathcal{D}_3 \ . \tag{10.20}$$

As F_i' implies F_i in T_i, α' is also a model for $F_1 \wedge F_2$ in the combined theory, which contradicts our assumption that φ is unsatisfiable. ▟

Note that, without the restriction to infinite domains, Algorithm 10.3.1 may fail. The original description of the algorithm lacked such a restriction. The algorithm was later amended by adding the requirement that the theories are *stably infinite*, which is a generalization of the requirement in our presentation. The following example, given by Tinelli and Zarba in [276], demonstrates why this restriction is important.

Example 10.14. Let T_1 be a theory over signature $\varSigma_1 = \{f\}$, where f is a function symbol, and axioms that enforce solutions with no more than two distinct values. Let T_2 be a theory over signature $\varSigma_2 = \{g\}$, where g is a function symbol.

Recall that the combined theory $T_1 \oplus T_2$ contains the union of the axioms. Hence, the solution to any formula $\varphi \in T_1 \oplus T_2$ cannot have more than two distinct values.

Now, consider the following formula:

$$f(x_1) \neq f(x_2) \wedge g(x_1) \neq g(x_3) \wedge g(x_2) \neq g(x_3) \ . \tag{10.21}$$

This formula is unsatisfiable in $T_1 \oplus T_2$ because any assignment satisfying it must use three different values for x_1, x_2, and x_3. However, this fact is not revealed by Algorithm 10.3.2, as illustrated in Table 10.7. ▟

F_1 (a Σ_1-formula)	F_2 (a Σ_2-formula)
$f(x_1) \neq f(x_2)$	$g(x_1) \neq g(x_3)$ $g(x_2) \neq g(x_3)$

Table 10.7. No equalities are propagated by Algorithm 10.3.2 when checking the formula (10.21). This results in an error: although $F_1 \wedge F_2$ is unsatisfiable, both F_1 and F_2 are satisfiable in their respective theories

An extension to the Nelson–Oppen combination procedure for nonstably infinite theories was given in [276], although the details of the procedure are beyond the scope of this book. The main idea is to compute, for each nonstably infinite theory T_i, a lower bound N_i on the size of the domain in which satisfiable formulas in this theory must be satisfied (it is not always possible to compute this bound). Then, the algorithm propagates this information between the theories along with the equalities. When it checks for consistency of an individual theory, it does so under the restrictions on the domain defined by the other theories. F_j is declared unsatisfiable if it does not have a solution within the bound N_i for all i.

10.4 Problems

Problem 10.1 (using the Nelson–Oppen procedure). Prove that the following formula is unsatisfiable using the Nelson–Oppen procedure, where the variables are interpreted over the integers:

$$g(f(x_1 - 2)) = x_1 + 2 \wedge g(f(x_2)) = x_2 - 2 \wedge (x_2 + 1 = x_1 - 1) \ .$$

Problem 10.2 (an improvement to the Nelson–Oppen procedure). A simple improvement to Algorithm 10.3.1 is to restrict the propagation of equalities in step 3 as follows. We call a variable *local* if it appears only in a single theory. Then, if an equality $v_i = v_j$ is implied by F_i and not by F_j, we propagate it to F_j only if v_i, v_j are not local to F_i. Prove the correctness of this improvement.

Problem 10.3 (proof of correctness of Algorithm 10.3.2 for the Nelson–Oppen procedure). Prove the correctness of Algorithm 10.3.2 by generalizing the proof of Algorithm 10.3.1 given in Sect. 10.3.3.

10.5 Bibliographic Notes

The theory combination problem (Definition 10.2) was shown to be undecidable in [40], hence combination methods must impose restrictions on the

Aside: An Abstract Version of the Nelson–Oppen Procedure

Let V be the set of variables used in $F_1, \ldots, F_n$. A partition P of V induces equivalence classes, in which variables are in the same class if and only if they are in the same partition as defined by P. (Every assignment to V's variables induces such a partition.) Denote by R the equivalence relation corresponding to these classes. The **arrangement** corresponding to P is defined by

$$ar(P) \doteq \bigwedge_{v_i\, R\, v_j, i<j} v_i = v_j \wedge \bigwedge_{\neg(v_i\, R\, v_j), i<j} v_i \neq v_j \; . \tag{10.22}$$

In words, the arrangement $ar(P)$ is a conjunction of all equalities and disequalities corresponding to P, modulo reflexivity and symmetry. For example, if $V := \{x_1, x_2, x_3\}$ and $P := \{\{x_1, x_2\}, \{x_3\}\}$, then

$$ar(P) := \; x_1 = x_2 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3 \; . \tag{10.23}$$

Now, consider the following abstract version of the Nelson–Oppen procedure:

1. Choose nondeterministically a partition P of V's variables.
2. If one of $F_i \wedge ar(P)$ with $i \in \{1, \ldots, n\}$ is unsatisfiable, return "Unsatisfiable". Otherwise, return "Satisfiable".

We have:

- *Termination.* The procedure terminates, since there is a finite number of partitions.
- *Soundness and completeness.* If the procedure returns "Unsatisfiable", then the input formula is unsatisfiable. Indeed, if there is a satisfying assignment to the combined theory, this assignment corresponds to some arrangement; testing this arrangement leads to a termination with the result "Satisfiable". Proving the other direction is harder, but also possible. See [275] for more details.

The nondeterministic step can be replaced with a deterministic one, by trying all such partitions possible. Hence, now it is clear that the requirement in the Nelson–Oppen procedure for sharing implied equalities can be understood as an optimization over an exhaustive search, rather than a necessity for correctness.

More generally, **abstract decision procedures** such as the one presented here are quite common in the literature. They are convenient for theoretical reasons, and can even help in designing concrete procedures in a more modular way. Abstracting some implementation details—typically by using nondeterminism—can be helpful for various reasons, such as clarity and generality, simplicity of proving an upper bound on the complexity, and simplicity of the correctness argument, as demonstrated above.

theories. There is a rich literature on combining decision procedures for first-order theories, starting with seminal papers by Nelson and Oppen [206] and by Shostak [259]. The presentation of the algorithm in this chapter is based on the former. The original presentation in [206] was not entirely correct, however, because it referred to general theories, although it is correct only for theories that are stably infinite. One year later, Oppen fixed this problem by adding this restriction, but without presenting a revised proof [215]. A full, model-theoretic proof was provided only in 1996 by Tinelli and Harandi in [275], which also serves as a basis for the (simplified) proof in Sect. 10.3.3. Several publications since then have extended the basic algorithms in order to combine theories with fewer restrictions. In Sect. 10.3.3, we mentioned Tinelli and Zarba's extension to the combination of nonstably infinite theories [276]. In [238] Ranise, Ringeissen, and Zarba identify a class of theories (called **polite theories**) that can be combined with nonstably infinite theories. Several extensions of these ideas were published by Jovanovic and Barrett [158].

Nelson and Oppen's combination procedure in its original form, as described in this chapter, can be optimized. Several optimizations have been suggested, including a method for avoiding the purification step [20]. There is empirical evidence showing that the computation of the implied equalities can become a bottleneck when one is combining, for example, linear arithmetic on the basis of the Simplex method [96].

Shostak's combination procedure [259] was considered to be an alternative to the Nelson–Oppen procedure for many years. However, Ruess and Shankar [246] showed in 2001 that it was in fact flawed in the general case (it was incomplete and not necessarily terminating), but is correct under certain restrictions. At the time several prominent theorem provers were using it. Currently (2015) only the theorem provers PVS and ALT-ERGO use some variation of Shostak's procedure, for cases where it is known to be correct. Here is N. Shankar's description of Shostak's method:

"Shostak's combination method is based on a far-reaching generalization of Gaussian elimination. He showed that many theories actually support a *canonizer* and a *solver*. A *canonizer* is an algorithm that transforms logically equivalent formulas to a syntactically identical representation. Given an equation of the form $a = b$ (where a, b are Σ-terms, Σ being the signature of the theory), a *solver* transforms it into an equivalent form $\mathit{solve}(a = b)$. The operation $\mathit{solve}(a = b)$ returns s, which is equisatisfiable to $a = b$. When the equation is unsolvable $s = \bot$, and otherwise s is a solution of the form $x_1 = e_1, \ldots, x_n = e_n$, where for $1 \leq i \leq n$, x_i is a variable in the equation $(a = b)$ and e_i is a Σ-term.

A canonizer σ for a theory can be used to decide that the equality $c = d$ is valid by applying the σ to c and d, respectively, to see if the canonical forms $\sigma(c)$ and $\sigma(d)$ are identical. A theory with such a solver and canonizer is called a *Shostak theory.*

Shostak's method uses the combination of a solver and canonizer to verify $a_1 = b_1, ..., a_n = b_n \vdash c = d$ by successively placing the antecedent equations into a solution set S, with $S_0 = \emptyset$ (the empty substitution), $S_i = \mathit{solve}(S(a_i = b_i))$ for $1 \leq i \leq n$, and $S = S_n$. The claim can then be checked by verifying that either some S_i is $\perp$ for $1 \leq i \leq n$, or $\sigma(S(c))$ and $\sigma(S(d))$ are identical canonical forms.

Though Shostak's ideas are deep, his original algorithm and proof had a number of flaws. He incorrectly claimed that solvers and canonizers for disjoint theories could be combined into a solver and canonizer for the union of these theories. The *basic* combination of a single Shostak theory with equality over uninterpreted functions was presented and proved correct in [246]. Ganzinger showed in [119] that a basic combination could be constructed solely with the solver and without needing a canonizer—the use of a canonizer can be seen as an optimization. Shankar and Ruess show in [257] a method for extending the basic combination to disjoint unions of Shostak theories without requiring the combination of solvers and canonizers."

The lazy approach, as described in Chap. 3, opens up new opportunities with regard to implementing the Nelson–Oppen combination procedure. A contribution by Bozzano et al. [41] suggests a technique called **delayed theory combination**. Each pair of shared variables is encoded with a new Boolean variable (resulting in a quadratic increase in the number of variables). After all the other encoding variables have been assigned, the SAT solver begins to assign values (arbitrary at first) to the new variables, and continues as usual, i.e., after every such assignment, the current partial assignment is sent to a theory solver. If any one of the theory solvers "objects" to the arrangement implied by this assignment (i.e., it finds a conflict with the current assignment to the other literals), this leads to a conflict and backtracking. Otherwise, the formula is declared satisfiable. This way, each theory can be solved separately, without passing information about equalities. Empirically, this method is very effective, both because the individual theory solvers need not worry about propagating equalities, and because only a small amount of information has to be shared between the theory solvers in practice—far less, on average, than is passed during the normal execution of the Nelson–Oppen procedure. In [159], Jovanovic and Barrett showed how many pairs of shared variables can be ignored, when they have no effect on the individual theories.

A different approach has been proposed by de Moura and Bjørner [91]. These authors also make the equalities part of the model, but instead of letting the SAT solver decide on their values, they attempt to compute a consistent assignment to the theory variables that is as diverse as possible. The equalities are then decided upon by following the assignment to the theory variables.

10.6 Glossary

The following symbols were used in this chapter:

Symbol	**Refers to ...**	**First used on page ...**
Σ	The signature of a theory, i.e., its set of nonlogical predicates and function symbols and their respective arities (i.e., those symbols that are *not* common to all first-order theories)	230
$T \models \varphi$	φ is T-valid	230
$T_1 \oplus T_2$	Denotes the theory obtained from combining the theories T_1 and T_2, i.e., a theory over $\Sigma_1 \cup \Sigma_2$ defined by the set of axioms $T_1 \cup T_2$	230
F_i	The pure (theory-specific) formulas in Algorithm 10.3.1	232
F_i'	The formula F_i upon termination of Algorithm 10.3.1	238
Δ	A constraint that forces all variables that are not implied to be equal to be different	238

11

Propositional Encodings

11.1 Lazy vs. Eager Encodings

The DPLL(T) method that was described in Chap. 3 is based on an interplay between a SAT solver and a theory solver for a conjunction of terms. This method is frequently called "lazy", emphasizing the fact that the theory solver is invoked only as needed, namely to check the consistency of a set of theory literals.

In this chapter we consider a different approach, which does not rely on alternations between SAT and a theory solver. Instead it is based on performing a full reduction of a T-formula to an equisatisfiable propositional formula. This approach is sometimes called "**eager**" because in contrast to the lazy approach, it performs a reduction to propositional logic in one step. A single run of the SAT solver on the propositional formula is then sufficient to decide the original formula.

The eager approach should be tailored for each theory T, although there exists a general strategy [171]. In this chapter we only demonstrate it for the case of equalities and uninterpreted functions, which were presented already in Chap. 4. There is some work in the literature on eager encoding for linear arithmetic [266], but we are not aware of literature on handling other theories this way. Generally the eager approach is less developed in the literature compared with the lazy one, and fewer tools support it.

We begin by considering two methods for eliminating uninterpreted functions, via a reduction to equality logic constraints. We will then show graph-based methods for an eager encoding of equality logic formulas into propositional logic.

11.2 From Uninterpreted Functions to Equality Logic

Luckily, we do not need to examine all possible interpretations of an uninterpreted function in a given EUF formula in order to know whether it is

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_11

valid. Instead, we rely on the strongest property that is common to all functions, namely functional consistency (see p. 80). Relying on this property, we can reduce the decision problem of EUF formulas to that of deciding equality logic. We shall see two possible reductions, **Ackermann's reduction** and **Bryant's reduction**, both of which enforce functional consistency by adding constraints. The size of the resulting formula in both cases may grow quadratically in the number of function instances. Ackermann's reduction is somewhat more intuitive to understand, but also imposes certain restrictions on the decision procedures that can be used to solve it. The implications of the differences between the two methods are explained in Sect. 11.7.

In the discussion that follows, for the sake of simplicity, we make several assumptions regarding the input formula: it has a single uninterpreted function, with a single argument, and no two instances of this function have the same argument. The generalization of the reductions is rather straightforward, as the examples later on demonstrate.

11.2.1 Ackermann's Reduction

Ackermann's reduction (Algorithm 11.2.1) adds explicit constraints to the formula in order to enforce the functional consistency requirement stated above. The algorithm reads an EUF formula φ^{UF} that we wish to validate, and transforms it to an equality logic formula φ^{E} of the form

$$\varphi^{\text{E}} := \; FC^{\text{E}} \implies \mathit{flat}^{\text{E}} \; , \tag{11.1}$$

where FC^{E} is a conjunction of functional-consistency constraints, and flat^{E} is a flattening of φ^{UF}, i.e., a formula in which each unique function instance is replaced with a corresponding new variable.

Example 11.1. Consider the formula

$$(x_1 \neq x_2) \lor (F(x_1) = F(x_2)) \lor (F(x_1) \neq F(x_3)) \; , \tag{11.2}$$

which we wish to reduce to equality logic using Algorithm 11.2.1.

After assigning indices to the instances of F (for this example, we assume that this is done from left to right), we compute flat^{E} and FC^{E} accordingly:

$$\mathit{flat}^{\text{E}} := \; (x_1 \neq x_2) \lor (f_1 = f_2) \lor (f_1 \neq f_3) \; , \tag{11.3}$$

$$\begin{aligned} FC^{\text{E}} := \; & (x_1 = x_2 \implies f_1 = f_2) \land \\ & (x_1 = x_3 \implies f_1 = f_3) \land \\ & (x_2 = x_3 \implies f_2 = f_3) \; . \end{aligned} \tag{11.4}$$

Equation (11.2) is valid if and only if the resulting equality formula φ^{E}, as prescribed in (11.1), is valid. ▟

In the next example, we go back to our running example for this chapter, and transform it to equality logic.

Algorithm 11.2.1: Ackermann's-Reduction

Input: An EUF formula φ^{UF} with m instances of an uninterpreted function F

$\boxed{m}$

Output: An equality logic formula φ^{E} such that φ^{E} is valid if and only if φ^{UF} is valid

1. Assign indices to the uninterpreted-function instances from subexpressions outwards. Denote by F_i the instance of F that is given the index i, and by $arg(F_i)$ its single argument.
2. Let $\mathit{flat}^{\text{E}} \doteq \mathcal{T}(\varphi^{\text{UF}})$, where $\mathcal{T}$ is a function that takes an EUF formula (or term) as input and transforms it to an equality formula (or term, respectively) by replacing each uninterpreted-function instance F_i with a new term-variable f_i (in the case of nested functions, only the variable corresponding to the most external instance remains).
3. Let FC^{E} denote the following conjunction of functional-consistency constraints:

$\boxed{F_i}$ $\boxed{arg(F_i)}$ $\boxed{\mathit{flat}^{\text{E}}}$ $\boxed{\mathcal{T}}$ $\boxed{FC^{\text{E}}}$

$$FC^{\text{E}} := \bigwedge_{i=1}^{m-1} \bigwedge_{j=i+1}^{m} (\mathcal{T}(arg(F_i)) = \mathcal{T}(arg(F_j))) \implies f_i = f_j \;.$$

4. Let

$$\varphi^{\text{E}} := FC^{\text{E}} \implies \mathit{flat}^{\text{E}} \;.$$

Return φ^{E}.

Example 11.2. We now use Ackermann's reduction to solve the problem stated in Sect. 4.2.2, which the reader is advised to revisit before continuing. Fig. 11.1 is simply a replica of Fig. 4.3 of that section.

Our example has four instances of the uninterpreted function G,

$$G(out0\_a, in),\; G(out1\_a, in),\; G(in, in),\; \text{and } G(G(in, in), in)\;,$$

which we number in this order. On the basis of (4.5), we compute flat^{E}, replacing each uninterpreted-function symbol with the corresponding variable:

$$\mathit{flat}^{\text{E}} := \left(\begin{pmatrix} out0\_a = in \;\wedge \\ out1\_a = g_1 \;\wedge \\ out2\_a = g_2 \end{pmatrix} \wedge out0\_b = g_4\right) \implies out2\_a = out0\_b\;. \quad (11.5)$$

The functional-consistency constraints are given by

Aside: Checking the *Satisfiability* of φ^{UF}

Ackermann's reduction was defined above for checking the *validity* of φ^{UF}. It tells us that we need to check for the validity of $\varphi^{\mathrm{E}} := FC^{\mathrm{E}} \implies \mathit{flat}^{\mathrm{E}}$ or, equivalently, check that $\neg\varphi^{\mathrm{E}} := FC^{\mathrm{E}} \wedge \neg\mathit{flat}^{\mathrm{E}}$ is unsatisfiable. This is important in our case, because all the algorithms that we shall see later check for *satisfiability* of formulas, not for their validity. Thus, as a first step we need to negate φ^{E}.

What if we want to check for the satisfiability of φ^{UF}? The short answer is that we need to check for the satisfiability of

$$\varphi^{\mathrm{E}} := \; FC^{\mathrm{E}} \wedge \mathit{flat}^{\mathrm{E}}\ .$$

This is interesting. Normally, if we check for the satisfiability or validity of a formula, this corresponds to checking for the satisfiability of the formula or of its negation, respectively. Thus, we could expect that checking the satisfiability of φ^{UF} is equivalent to checking the satisfiability of $(FC^{\mathrm{E}} \implies \mathit{flat}^{\mathrm{E}})$. However, this is not the same as the above equation. So what has happened here? The reason for the difference is that we check the satisfiability of φ^{UF} *before* the reduction. This means that we can use Ackermann's reduction to check the validity of $\neg\varphi^{\mathrm{UF}}$. The functional-consistency constraints FC^{E} remain unchanged whether we check φ^{UF} or its negation $\neg\varphi^{\mathrm{UF}}$. Thus, we need to check the validity of $FC^{\mathrm{E}} \implies \neg\mathit{flat}^{\mathrm{E}}$, which is the same as checking the satisfiability of $FC^{\mathrm{E}} \wedge \mathit{flat}^{\mathrm{E}}$, as stated above.

$$\begin{array}{lcl} \mathit{out}0\_a = \mathit{in} & \wedge & \\ \mathit{out}1\_a = G(\mathit{out}0\_a, \mathit{in}) & \wedge & \qquad\qquad \mathit{out}0\_b = G(G(\mathit{in}, \mathit{in}), \mathit{in}) \\ \mathit{out}2\_a = G(\mathit{out}1\_a, \mathit{in}) & & \\ & & \\ \quad (\varphi_a^{\mathrm{UF}}) & & \qquad\qquad\qquad\qquad (\varphi_b^{\mathrm{UF}}) \end{array}$$

Fig. 11.1. After replacing "$*$" with the uninterpreted function G

$$\begin{array}{rllll} FC^{\mathrm{E}} := & ((\mathit{out}0\_a = \mathit{out}1\_a & \wedge\ \mathit{in} = \mathit{in}) & \implies g_1 = g_2) & \wedge \\ & ((\mathit{out}0\_a = \mathit{in} & \wedge\ \mathit{in} = \mathit{in}) & \implies g_1 = g_3) & \wedge \\ & ((\mathit{out}0\_a = g_3 & \wedge\ \mathit{in} = \mathit{in}) & \implies g_1 = g_4) & \wedge \\ & ((\mathit{out}1\_a = \mathit{in} & \wedge\ \mathit{in} = \mathit{in}) & \implies g_2 = g_3) & \wedge \\ & ((\mathit{out}1\_a = g_3 & \wedge\ \mathit{in} = \mathit{in}) & \implies g_2 = g_4) & \wedge \\ & ((\mathit{in} = g_3 & \wedge\ \mathit{in} = \mathit{in}) & \implies g_3 = g_4)\ . & \end{array} \tag{11.6}$$

The resulting equality formula is $FC^{\mathrm{E}} \implies \mathit{flat}^{\mathrm{E}}$, which we need to validate.

This example demonstrates how to generalize the reduction to functions with several arguments: only if all arguments of a pair of function instances are the same (pairwise) is the return value of the function forced to be the same.

The reader may observe that most of these constraints are in fact redundant. The validity of the formula depends on $G(\mathit{out}0\_a, \mathit{in})$ being equal to

$G(in, in)$, and $G(out1\_a, in)$ being equal to $G(G(in, in), in)$. Hence, only the second and fifth constraints in (11.6) are necessary. In practice, such observations are important because the quadratic growth in the number of functional-consistency constraints may become a bottleneck. When comparing two systems, as in this case, it is frequently possible to detect in polynomial time large sets of constraints that can be removed without affecting the validity of the formula. More details of this technique can be found in [229]. ▟

Finally, we consider the case in which there is more than one function symbol.

Example 11.3. Consider now the following formula, which we wish to validate:

$$x_1 = x_2 \implies \underbrace{F(\underbrace{F(\overbrace{G(x_1)}^{g_1})}_{f_1})}_{f_2} = \underbrace{F(\underbrace{F(\overbrace{G(x_2)}^{g_2})}_{f_3})}_{f_4} \ . \tag{11.7}$$

We index the function instances from the inside out (from subexpressions outwards) and compute the following:

$$\mathit{flat}^{\mathrm{E}} := \ x_1 = x_2 \implies f_2 = f_4 \ , \tag{11.8}$$

$$\begin{array}{rrcll} FC^{\mathrm{E}} := & x_1 = x_2 & \implies & g_1 = g_2 & \wedge \\ & g_1 = f_1 & \implies & f_1 = f_2 & \wedge \\ & g_1 = g_2 & \implies & f_1 = f_3 & \wedge \\ & g_1 = f_3 & \implies & f_1 = f_4 & \wedge \\ & f_1 = g_2 & \implies & f_2 = f_3 & \wedge \\ & f_1 = f_3 & \implies & f_2 = f_4 & \wedge \\ & g_2 = f_3 & \implies & f_3 = f_4 \ . & \end{array} \tag{11.9}$$

Then, again,

$$\varphi^{\mathrm{E}} := \ FC^{\mathrm{E}} \implies \mathit{flat}^{\mathrm{E}} \ . \tag{11.10}$$

▟

From these examples, it is clear how to generalize Algorithm 11.2.1 to multiple uninterpreted functions. We leave this and other extensions as an exercise (Problem 11.2).

11.2.2 Bryant's Reduction

Bryant's reduction (Algorithm 11.2.2) has the same goal as Ackermann's reduction: to transform EUF formulas to equality logic formulas, such that they are equisatisfiable. To check the *satisfiability* of φ^{UF} rather than the validity, we return $FC^{\mathrm{E}} \wedge \mathit{flat}^{\mathrm{E}}$ in the last step.

The semantics of the *case* expression used in step 3 is such that its value is determined by the first condition that is evaluated to TRUE. Its translation

Algorithm 11.2.2: BRYANT'S-REDUCTION

Input: An EUF formula φ^{UF} with m instances of an uninterpreted function F

Output: An equality logic formula φ^{E} such that φ^{E} is valid if and only if φ^{UF} is valid

1. Assign indices to the uninterpreted-function instances from subexpressions outwards. Denote by F_i the instance of F that is given the index i, and by $arg(F_i)$ its single argument.

$\boxed{\mathit{flat}^{\text{E}}}$ $\boxed{\mathcal{T}^\star}$ $\boxed{F_i^\star}$

2. Let $\mathit{flat}^{\text{E}} = \mathcal{T}^\star(\varphi^{\text{UF}})$, where $\mathcal{T}^\star$ is a function that takes an EUF formula (or term) as input and transforms it to an equality formula (or term, respectively) by replacing each uninterpreted-function instance F_i with a new term-variable $F_i^\star$ (in the case of nested functions, only the variable corresponding to the most external instance remains).
3. For $i \in \{1, \ldots, m\}$, let f_i be a new variable, and let $F_i^\star$ be defined as follows:

$$F_i^\star := \left(\begin{array}{lcl} \text{case } \mathcal{T}^\star(arg(F_1^\star)) & = \mathcal{T}^\star(arg(F_i^\star)) & : f_1 \\ \qquad\vdots & \vdots & \\ \quad\mathcal{T}^\star(arg(F_{i-1}^\star)) & = \mathcal{T}^\star(arg(F_i^\star)) & : f_{i-1} \\ \quad\text{TRUE} & & : f_i \end{array}\right) . \tag{11.11}$$

   Finally, let

$$FC^{\text{E}} := \bigwedge_{i=1}^{m} F_i^\star . \tag{11.12}$$

4. Let

$$\varphi^{\text{E}} := FC^{\text{E}} \implies \mathit{flat}^{\text{E}} . \tag{11.13}$$

   Return φ^{E}.

to an equality logic formula, assuming that the argument of F_i is a variable x_i for all i, is given by

$$\bigvee_{j=1}^{i} \left(F_i^\star = f_j \wedge (x_j = x_i) \wedge \bigwedge_{k=1}^{j-1} (x_k \neq x_i)\right) . \tag{11.14}$$

Example 11.4. Given the case expression

$$F_3^\star = \left(\begin{array}{rl} \text{case } x_1 = x_3 & : f_1 \\ x_2 = x_3 & : f_2 \\ \text{TRUE} & : f_3 \end{array}\right) , \tag{11.15}$$

its equivalent equality logic formula is given by

$$\begin{array}{ll}(F_3^\star = f_1 \wedge x_1 = x_3) & \vee \\ (F_3^\star = f_2 \wedge x_2 = x_3 \wedge x_1 \neq x_3) & \vee \\ (F_3^\star = f_3 \wedge x_1 \neq x_3 \wedge x_2 \neq x_3) & .\end{array} \qquad (11.16)$$

▟

The differences between the two reduction schemes are:

1. Step 1 in Bryant's reduction requires a certain order when indices are assigned to function instances. Such an order is not required in Ackermann's reduction.
2. Step 2 in Bryant's reduction replaces function instances with $F^\star$ variables rather than with f variables. The $F^\star$ variables should be thought of simply as macros, or *placeholders*, which means that they are used only for simplifying the writing of the formula. We can do without them if we remove FC^{E} from the formula altogether and substitute them in $\mathit{flat}^{\mathrm{E}}$ with their definitions. The reason that we retain them is to make the presentation more readable and to retain a structure similar to that of Ackermann's reduction.
3. The definition of FC^{E}, which enforces functional consistency, relies on *case* expressions rather than on a pairwise enforcing of consistency.

The generalization of Algorithm 11.2.2 to functions with multiple arguments is straightforward, as we shall soon see in the examples.

Example 11.5. Let us return to our main example of this chapter, the problem of proving the equivalence of programs (a) and (b) in Fig. 4.1. We continue from Fig. 4.3, where the logical formulas corresponding to these programs are given, with the use of the uninterpreted function G. On the basis of (4.5), we compute $\mathit{flat}^{\mathrm{E}}$, replacing each uninterpreted-function symbol with the corresponding variable:

$$\mathit{flat}^{\mathrm{E}} := \left(\begin{pmatrix} out0\_a = in \ \wedge \\ out1\_a = G_1^\star \wedge \\ out2\_a = G_2^\star \end{pmatrix} \wedge (out0\_b = G_4^\star) \right) \implies out2\_a = out0\_b \ . \qquad (11.17)$$

Not surprisingly, this looks very similar to (11.5). The only difference is that, instead of the g_i variables, we now have the $G_i^\star$ macros, for $1 \leq i \leq 4$. Recall their origin: the function instances are $G(out0\_a, in)$, $G(out1\_a, in)$, $G(in, in)$, and $G(G(in, in), in)$, which we number in this order. The corresponding functional-consistency constraints are

$$FC^{\mathrm{E}} := \begin{array}{ll} G_1^\star = g_1 & \wedge \\ G_2^\star = \left(\begin{array}{ll} \text{case } \mathit{out0\_a} = \mathit{out1\_a} \wedge \mathit{in} = \mathit{in} & : g_1 \\ \quad \text{TRUE} & : g_2 \end{array} \right) & \wedge \\ G_3^\star = \left(\begin{array}{lll} \text{case } \mathit{out0\_a} = \mathit{in} & \wedge\, \mathit{in} = \mathit{in} & : g_1 \\ \quad \mathit{out1\_a} = \mathit{in} & \wedge\, \mathit{in} = \mathit{in} & : g_2 \\ \quad \text{TRUE} & & : g_3 \end{array} \right) & \wedge \\ G_4^\star = \left(\begin{array}{lll} \text{case } \mathit{out0\_a} = G_3^\star & \wedge\, \mathit{in} = \mathit{in} & : g_1 \\ \quad \mathit{out1\_a} = G_3^\star & \wedge\, \mathit{in} = \mathit{in} & : g_2 \\ \quad \mathit{in} = G_3^\star & \wedge\, \mathit{in} = \mathit{in} & : g_3 \\ \quad \text{TRUE} & & : g_4 \end{array} \right) & , \end{array} \tag{11.18}$$

and since we are checking for validity, the formula to be checked is

$$\varphi^{\mathrm{E}} := \; FC^{\mathrm{E}} \implies \mathit{flat}^{\mathrm{E}} \, . \tag{11.19}$$

▟

Example 11.6. If there are multiple uninterpreted-function symbols, the reduction is applied to each of them separately, as demonstrated in the following example, in which we consider the formula of Example 11.3 again:

$$x_1 = x_2 \implies \underbrace{F(\underbrace{F(\overbrace{G(x_1)}^{G_1^\star})}_{F_1^\star})}_{F_2^\star} = \underbrace{F(\underbrace{F(\overbrace{G(x_2)}^{G_2^\star})}_{F_3^\star})}_{F_4^\star} \; . \tag{11.20}$$

As before, we number the function instances of each of the uninterpreted-function symbols F and G from the inside out (this order is required in Bryant's reduction). Applying Bryant's reduction, we obtain

$$\mathit{flat}^{\mathrm{E}} := \; (x_1 = x_2 \implies F_2^\star = F_4^\star) \, , \tag{11.21}$$

$$FC^{\mathrm{E}} := \begin{array}{ll} F_1^\star = f_1 & \wedge \\ F_2^\star = \left(\begin{array}{ll} \text{case } G_1^\star = F_1^\star & : f_1 \\ \qquad \text{TRUE} & : f_2 \end{array} \right) & \wedge \\ F_3^\star = \left(\begin{array}{ll} \text{case } G_1^\star = G_2^\star & : f_1 \\ \qquad F_1^\star = G_2^\star & : f_2 \\ \qquad \text{TRUE} & : f_3 \end{array} \right) & \wedge \\ F_4^\star = \left(\begin{array}{ll} \text{case } G_1^\star = F_3^\star & : f_1 \\ \qquad F_1^\star = F_3^\star & : f_2 \\ \qquad G_2^\star = F_3^\star & : f_3 \\ \qquad \text{TRUE} & : f_4 \end{array} \right) & \wedge \\ & \\ G_1^\star = g_1 & \wedge \\ G_2^\star = \left(\begin{array}{ll} \text{case } x_1 = x_2 & : g_1 \\ \qquad \text{TRUE} & : g_2 \end{array} \right) & , \end{array} \tag{11.22}$$

and

$$\varphi^{\mathrm{E}} := \; FC^{\mathrm{E}} \implies \mathit{flat}^{\mathrm{E}} \; . \tag{11.23}$$

Note that, in any satisfying assignment that satisfies $x_1 = x_2$ (the premise of (11.20)), $F_1^\star$ and $F_3^\star$ are equal to f_1, while $F_2^\star$ and $F_4^\star$ are equal to f_2. ▟

The difference between Ackermann's and Bryant's reductions is not just syntactic, as was hinted earlier. It has implications for the decision procedure that one can use when solving the resulting formula. We discuss this point further in Sect. 11.7.

11.3 The Equality Graph

In this section, we present several basic terms that are used later in the chapter. We assume from here on that uninterpreted functions have already been eliminated with one of the reduction methods described in Sect. 11.2, i.e., that we are solving the satisfiability problem for equality logic without uninterpreted functions. Recall that we are also assuming that the formula is given to us in NNF and without constants. Recall further that an atom in such formulas is an equality predicate, and a literal is either an atom or its negation (see Definition 1.11). Given an equality logic formula φ^{E}, we denote the set of atoms of φ^{E} by $At(\varphi^{\mathrm{E}})$.

$\boxed{At(\varphi^{\mathrm{E}})}$

Definition 11.7 (equality and disequality literals sets). *The* equality literals set $E_=$ *of an equality logic formula* φ^{E} *is the set of positive literals in* φ^{E}*. The* disequality literals set $E_{\neq}$ *of an equality logic formula* φ^{E} *is the set of disequality literals in* φ^{E}*.*

$\boxed{E_=}$

$\boxed{E_{\neq}}$

It is possible, of course, that an equality may appear in the equality literals set and its negation in the disequality literals set.

Example 11.8. Consider the formula

$$\begin{array}{l}(u_1 = F(x_1, y_1) \wedge u_2 = F(x_2, y_2) \wedge z = G(u_1, u_2)) \\ \implies \;\; z = G(F(x_1, y_1), F(x_2, y_2))\end{array} \tag{11.24}$$

(this is the same formula as (4.22)).

Next, we apply Ackermann's reduction (Algorithm 11.2.1):

$$\varphi^{\mathrm{E}} := \begin{array}{c}\left(\begin{array}{l}(x_1 = x_2 \wedge y_1 = y_2 \implies f_1 = f_2) \wedge \\ (u_1 = f_1 \wedge u_2 = f_2 \implies g_1 = g_2)\end{array}\right) \implies \\ ((u_1 = f_1 \;\wedge\; u_2 = f_2 \;\wedge\; z = g_1) \quad \implies \quad z = g_2) \; ,\end{array} \tag{11.25}$$

which we can rewrite as

$$\varphi^{\mathrm{E}} := \left(\begin{array}{c}(x_1 = x_2 \wedge y_1 = y_2 \implies f_1 = f_2) \wedge \\ (u_1 = f_1 \wedge u_2 = f_2 \implies g_1 = g_2) \wedge \\ u_1 = f_1 \;\wedge\; u_2 = f_2 \;\wedge\; z = g_1\end{array}\right) \implies \quad z = g_2 \; . \tag{11.26}$$

The negation normal form of $\neg\varphi^{\mathrm{E}}$ is

$$\neg\varphi^{\mathrm{E}} := \left(\begin{array}{l} (x_1 \neq x_2 \vee y_1 \neq y_2 \vee f_1 = f_2) \wedge \\ (u_1 \neq f_1 \vee u_2 \neq f_2 \vee g_1 = g_2) \wedge \\ (u_1 = f_1 \wedge u_2 = f_2 \wedge z = g_1) \end{array}\right) \wedge\ z \neq g_2 \,. \tag{11.27}$$

We therefore have

$$\begin{array}{ll} E_= := & \{(f_1 = f_2),\ (g_1 = g_2),\ (u_1 = f_1),\ (u_2 = f_2),\ (z = g_1)\} \\ E_{\neq} := & \{(x_1 \neq x_2),\ (y_1 \neq y_2),\ (u_1 \neq f_1),\ (u_2 \neq f_2),\ (z \neq g_2)\} \,. \end{array} \tag{11.28}$$

■

$\boxed{G^{\mathrm{E}}}$

Definition 11.9 (equality graph). *Given an equality logic formula φ^{E} in NNF, the* equality graph *that corresponds to φ^{E}, denoted by $G^{\mathrm{E}}(\varphi^{\mathrm{E}})$, is an undirected graph $(V, E_=, E_{\neq})$ where the nodes in V correspond to the variables in φ^{E}, the edges in $E_=$ correspond to the predicates in the equality literals set of φ^{E}, and the edges in $E_{\neq}$ correspond to the predicates in the disequality literals set of φ^{E}.*

Note that we overload the symbols $E_=$ and $E_{\neq}$ so that each represents both the literals sets and the edges that represent them in the equality graph. Similarly, when we say that an assignment "satisfies an edge", we mean that it satisfies the literal represented by that edge.

We may write simply G^{E} for an equality graph when the formula it corresponds to is clear from the context. Graphically, equality literals are represented as dashed edges and disequality literals as solid edges, as illustrated in Fig. 11.2.

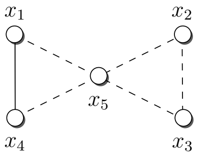


Fig. 11.2. An equality graph. Dashed edges represent $E_=$ literals (equalities), and solid edges represent $E_{\neq}$ literals (disequalities)

It is important to note that the equality graph $G^{\mathrm{E}}(\varphi^{\mathrm{E}})$ represents an *abstraction* of φ^{E}: more specifically, it represents all the equality logic formulas that have the same literal sets as φ^{E}. Since it disregards the Boolean connectives, it can represent both a satisfiable and an unsatisfiable formula. For example, although $x_1 = x_2 \wedge x_1 \neq x_2$ is unsatisfiable and $x_1 = x_2 \vee x_1 \neq x_2$ is satisfiable, both formulas are represented by the same equality graph.

$\boxed{x =^* y}$

Definition 11.10 (equality path). *An* equality path *in an equality graph G^{E} is a path consisting of $E_=$ edges. We denote by $x =^* y$ the fact that there exists an equality path from x to y in G^{E}, where $x, y \in V$.*

Definition 11.11 (disequality path). *A* disequality path *in an equality graph G^{E} is a path consisting of $E_=$ edges and a single $E_{\neq}$ edge. We denote by $x \neq^* y$ the fact that there exists a disequality path from x to y in G^{E}, where $x, y \in V$.*

$\boxed{x \neq^* y}$

Similarly, we use the terms *simple equality path* and *simple disequality path* when the path is required to be loop-free.

Consider Fig. 11.2 and observe, for example, that $x_2 =^* x_4$ owing to the path x_2, x_5, x_4, and $x_2 \neq^* x_4$ owing to the path x_2, x_5, x_1, x_4. In this case, both paths are simple. Intuitively, if $x =^* y$ in $G^{\mathrm{E}}(\varphi^{\mathrm{E}})$, then it might be necessary to assign the two variables equal values in order to satisfy φ^{E}. We say "might" because, once again, the equality graph obscures details about φ^{E}, as it disregards the Boolean structure of φ^{E}. The only fact that we know from $x =^* y$ is that there exist formulas whose equality graph is $G^{\mathrm{E}}(\varphi^{\mathrm{E}})$ and that, in any assignment satisfying them, $x = y$. However, we do not know whether φ^{E} is one of them. A disequality path $x \neq^* y$ in $G^{\mathrm{E}}(\varphi^{\mathrm{E}})$ implies the opposite: it might be necessary to assign different values to x and y in order to satisfy φ^{E}.

The case in which both $x =^* y$ and $x \neq^* y$ hold in $G^{\mathrm{E}}(\varphi^{\mathrm{E}})$ requires special attention. We say that the graph, in this case, contains a *contradictory cycle*.

Definition 11.12 (contradictory cycle). *In an equality graph, a* contradictory cycle *is a cycle with exactly one disequality edge.*

For every pair of nodes x, y in a contradictory cycle, it holds that $x =^* y$ and $x \neq^* y$.

Contradictory cycles are of special interest to us because the conjunction of the literals corresponding to their edges is unsatisfiable. Furthermore, since we have assumed that there are no constants in the formula, these are the only topologies that have this property. Consider, for example, a contradictory cycle with nodes $x_1, \ldots, x_k$ in which (x_1, x_k) is the disequality edge. The conjunction

$$x_1 = x_2 \wedge \ldots \wedge x_{k-1} = x_k \wedge x_k \neq x_1 \tag{11.29}$$

is clearly unsatisfiable.

All the decision procedures that we consider refer explicitly or implicitly to contradictory cycles. For most algorithms we can further simplify this definition by considering only *simple contradictory cycles*. A cycle is simple if it is represented by a path in which none of the vertices is repeated, other than the starting and ending vertices.

11.4 Simplifications of the Formula

Regardless of the algorithm that is used for deciding the satisfiability of a given equality logic formula φ^{E}, it is almost always the case that φ^{E} can be simplified significantly before the algorithm is invoked. Algorithm 11.4.1 presents such a simplification.

```
Algorithm 11.4.1: SIMPLIFY-EQUALITY-FORMULA

Input:  An equality formula $\varphi^{E}$
Output: An equality formula $\varphi^{E'}$ equisatisfiable with $\varphi^{E}$, with
        size less than or equal to the length of $\varphi^{E}$

1. Let $\varphi^{E'} := \varphi^{E}$.
2. Construct the equality graph $G^{E}(\varphi^{E'})$.
3. Replace each pure literal in $\varphi^{E'}$ whose corresponding edge is not part
   of a simple contradictory cycle with TRUE.
4. Simplify $\varphi^{E'}$ with respect to the Boolean constants TRUE and FALSE
   (e.g., replace TRUE $\lor \phi$ with TRUE, and FALSE $\land \phi$ with FALSE).
5. If any rewriting has occurred in the previous two steps, go to step 2.
6. Return $\varphi^{E'}$.
```

The following example illustrates the steps of Algorithm 11.4.1.

Example 11.13. Consider (11.27). Figure 11.3 illustrates $G^{E}(\varphi^{E})$, the equality graph corresponding to φ^{E}.

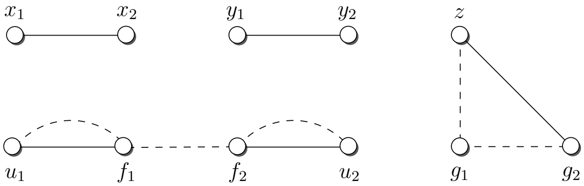


Fig. 11.3. The equality graph corresponding to Example 11.13. The edges $f_1 = f_2$, $x_1 \neq x_2$, and $y_1 \neq y_2$ are not part of any contradictory cycle, and hence their respective predicates in the formula can be replaced with TRUE

In this case, the edges $f_1 = f_2$, $x_1 \neq x_2$ and $y_1 \neq y_2$ are not part of any simple contradictory cycle and can therefore be substituted by TRUE. This results in

$$\varphi^{E'} := \left(\begin{array}{l} (\text{TRUE} \quad \lor \text{TRUE} \quad \lor \text{TRUE}) \quad \land \\ (u_1 \neq f_1 \lor u_2 \neq f_2 \lor g_1 = g_2) \land \\ (u_1 = f_1 \land u_2 = f_2 \land z = g_1 \quad \land z \neq g_2) \end{array} \right) , \tag{11.30}$$

which, after simplification according to step 4, is equal to

$$\varphi^{\mathrm{E}\prime} := \left(\begin{array}{l} (u_1 \neq f_1 \vee u_2 \neq f_2 \vee g_1 = g_2) \wedge \\ (u_1 = f_1 \wedge u_2 = f_2 \wedge z = g_1 \quad \wedge z \neq g_2) \end{array}\right) . \tag{11.31}$$

Reconstructing the equality graph after this simplification does not yield any more simplifications, and the algorithm terminates.

Now, consider a similar formula in which the predicates $x_1 \neq x_2$ and $u_1 \neq f_1$ are swapped. This results in the formula

$$\varphi^{\mathrm{E}} := \left(\begin{array}{l} (u_1 \neq f_1 \vee y_1 \neq y_2 \vee f_1 = f_2) \wedge \\ (x_1 \neq x_2 \vee u_2 \neq f_2 \vee g_1 = g_2) \wedge \\ (u_1 = f_1 \wedge u_2 = f_2 \wedge z = g_1 \quad \wedge z \neq g_2) \end{array}\right) . \tag{11.32}$$

Although we start from exactly the same graph, the simplification algorithm is now much more effective. After the first step we have

$$\varphi^{\mathrm{E}\prime} := \left(\begin{array}{l} (u_1 \neq f_1 \vee \textsc{true} \quad \vee \textsc{true}) \quad \wedge \\ (\textsc{true} \quad \vee u_2 \neq f_2 \vee g_1 = g_2) \wedge \\ (u_1 = f_1 \wedge u_2 = f_2 \wedge z = g_1 \quad \wedge z \neq g_2) \end{array}\right) , \tag{11.33}$$

which, after step 4, simplifies to

$$\varphi^{\mathrm{E}\prime} := \left((u_1 = f_1 \wedge u_2 = f_2 \wedge \ z = g_1 \wedge \ z \neq g_2) \right) . \tag{11.34}$$

The graph corresponding to $\varphi^{\mathrm{E}\prime}$ after this step appears in Fig. 11.4.

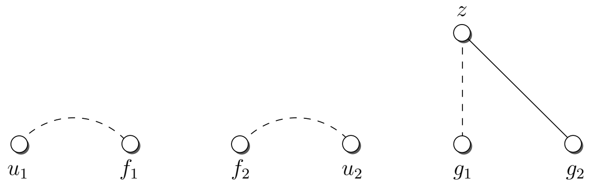


Fig. 11.4. An equality graph corresponding to (11.34), showing the first iteration of step 4

Clearly, no edges in $\varphi^{\mathrm{E}\prime}$ belong to a contradictory cycle after this step, which implies that we can replace all the remaining predicates by TRUE. Hence, in this case, simplification alone proves that the formula is satisfiable, without invoking a decision procedure. ▟

Although we leave the formal proof of the correctness of Algorithm 11.4.1 as an exercise (Problem 11.6), let us now consider what such a proof may look like. Correctness can be shown by proving that steps 3 and 4 maintain satisfiability (as these are the only steps in which the formula is changed). The simplifications in step 4 trivially maintain satisfiability, so the main problem is step 3.

Let φ_1^{E} and φ_2^{E} be the equality formulas before and after step 3, respectively. We need to show that these formulas are equisatisfiable.

($\Rightarrow$) If φ_1^{E} is satisfiable, then so is φ_2^{E}. This is implied by the monotonicity of NNF formulas (see Theorem 1.14) and the fact that only pure literals are replaced by TRUE.

($\Leftarrow$) If φ_2^{E} is satisfiable, then so is φ_1^{E}. Only a proof sketch and an example will be given here. The idea is to construct a satisfying assignment α_1 for φ_1^{E} while relying on the existence of a satisfying assignment α_2 for φ_2^{E}. Specifically, α_1 should satisfy exactly the same predicates as are satisfied by α_2, but also satisfy all those predicates that were replaced by TRUE. The following simple observation can be helpful in this construction: given a satisfying assignment to an equality formula, shifting the values in the assignment uniformly maintains satisfaction (because the values of the equality predicates remain the same). The same observation applies to an assignment of *some* of the variables, as long as none of the predicates that refer to *one* of these variables becomes FALSE owing to the new assignment.

Consider, for example, (11.32) and (11.33), which correspond to φ_1^{E} and φ_2^{E}, respectively, in our argument. An example of a satisfying assignment to the latter is

$$\alpha_2 := \ \{u_1 \mapsto 0,\ f_1 \mapsto 0,\ f_2 \mapsto 1,\ u_2 \mapsto 1, z \mapsto 0,\ g_1 \mapsto 0,\ g_2 \mapsto 1\}\ . \quad (11.35)$$

First, α_1 is set equal to α_2. Second, we need to extend α_1 with an assignment of those variables not assigned by α_2. The variables in this category are x_1, x_2, y_1, and y_2, which can be trivially satisfied because they are not part of any equality predicate. Hence, assigning a unique value to each of them is sufficient. For example, we can now have

$$\alpha_1 := \ \alpha_1 \cup \{x_1 \mapsto 2,\ x_2 \mapsto 3,\ y_1 \mapsto 4,\ y_2 \mapsto 5\}\ . \quad (11.36)$$

Third, we need to consider predicates that are replaced by TRUE in step 3 but are not satisfied by α_1. In our example, $f_1 = f_2$ is such a predicate. To solve this problem, we simply shift the assignment to f_2 and u_2 so that the predicate $f_1 = f_2$ is satisfied (a shift by minus 1 in this case). This clearly maintains the satisfaction of the predicate $u_2 = f_2$. The assignment that satisfies φ_1^{E} is thus

$$\begin{aligned}\alpha_1 := \ &\{u_1 \mapsto 0,\ f_1 \mapsto 0,\ f_2 \mapsto 0,\ u_2 \mapsto 0, z \mapsto 0,\ g_1 \mapsto 0,\ g_2 \mapsto 1,\\ &\ x_1 \mapsto 2,\ x_2 \mapsto 3,\ y_1 \mapsto 4,\ y_2 \mapsto 5\}\ .\end{aligned} \quad (11.37)$$

A formal proof based on this argument should include a precise definition of these shifts, i.e., which vertices they apply to, and an argument as to why no circularity can occur. Circularity can affect the termination of the procedure that constructs α_1.

11.5 A Graph-Based Reduction to Propositional Logic

We now consider a decision procedure for equality logic that is based on a reduction to propositional logic. This procedure was originally presented by Bryant and Velev in [57] (under the name of the **sparse method**). Several definitions and observations are necessary.

Definition 11.14 (nonpolar equality graph). *Given an equality logic formula φ^{E}, the* nonpolar equality graph *corresponding to φ^{E}, denoted by $G^{\mathrm{E}}_{\mathrm{NP}}(\varphi^{\mathrm{E}})$, is an undirected graph (V, E) where the nodes in V correspond to the variables in φ^{E}, and the edges in E correspond to $At(\varphi^{\mathrm{E}})$, i.e., the equality predicates in φ^{E}.*

$\boxed{G^{\mathrm{E}}_{\mathrm{NP}}}$

A nonpolar equality graph represents a degenerate version of an equality graph (Definition 11.9), since it disregards the polarity of the equality predicates.

Given an equality logic formula φ^{E}, the procedure generates two propositional formulas $e(\varphi^{\mathrm{E}})$ and $\mathcal{B}_{trans}$, such that

$\boxed{e(\varphi^{\mathrm{E}})}$

$$\varphi^{\mathrm{E}} \text{ is satisfiable} \iff e(\varphi^{\mathrm{E}}) \wedge \mathcal{B}_{trans} \text{ is satisfiable.} \tag{11.38}$$

$\boxed{\mathcal{B}_{trans}}$

The formulas $e(\varphi^{\mathrm{E}})$ and $\mathcal{B}_{trans}$ are defined as follows:

- The formula $e(\varphi^{\mathrm{E}})$ is the **propositional skeleton** of φ^{E}, which means that every equality predicate of the form $x_i = x_j$ in φ^{E} is replaced with a new Boolean variable $e_{i,j}$.[1] For example, let

  $$\varphi^{\mathrm{E}} := \; x_1 = x_2 \wedge (((x_2 = x_3) \wedge (x_1 \neq x_3)) \vee (x_1 \neq x_2)) \; . \tag{11.39}$$

  Then,

  $$e(\varphi^{\mathrm{E}}) := \; e_{1,2} \wedge ((e_{2,3} \wedge \neg e_{1,3}) \vee \neg e_{1,2}) \; . \tag{11.40}$$

  It is not hard to see that, if φ^{E} is satisfiable, then so is $e(\varphi^{\mathrm{E}})$. The other direction, however, does not hold. For example, while (11.39) is unsatisfiable, its encoding in (11.40) is satisfiable. To maintain an equisatisfiability relation, we need to add constraints that impose the transitivity of equality, which was lost in the encoding. This is the role of $\mathcal{B}_{trans}$.
- The formula $\mathcal{B}_{trans}$ is a conjunction of implications, which are called *transitivity constraints*. Each such implication is associated with a cycle in the nonpolar equality graph. For a cycle with n edges, $\mathcal{B}_{trans}$ forbids an assignment FALSE to one of the edges when all the other edges are assigned TRUE. Imposing this constraint for each of the edges in each one of the cycles is sufficient to satisfy the condition stated in (11.38).

[1] To avoid introducing dual variables such as $e_{i,j}$ and $e_{j,i}$, we can assume that all equality predicates in φ^{E} appear in such a way that the left variable precedes the right one in some predefined order.

Example 11.15. The atoms $x_1 = x_2, x_2 = x_3, x_1 = x_3$ form a cycle of size 3 in the nonpolar equality graph. The following constraint is sufficient for maintaining the condition stated in (11.38):

$$\mathcal{B}_{trans} = \begin{pmatrix} (e_{1,2} \wedge e_{2,3} \implies e_{1,3}) \wedge \\ (e_{1,2} \wedge e_{1,3} \implies e_{2,3}) \wedge \\ (e_{2,3} \wedge e_{1,3} \implies e_{1,2}) \end{pmatrix} . \tag{11.41}$$

▟

Adding n constraints for each cycle is not very practical, however, because there can be an exponential number of cycles in a given undirected graph.

Definition 11.16 (chord). *A* chord *of a cycle is an edge connecting two nonadjacent nodes of the cycle. If a cycle has no chords in a given graph, it is called a* **chord-free cycle***.*

Bryant and Velev proved the following theorem:

Theorem 11.17. *It is sufficient to add transitivity constraints over* simple chord-free *cycles in order to maintain (11.38).*

For a formal proof, see [57]. The following example may be helpful for developing an intuition as to why this theorem is correct.

Example 11.18. Consider the cycle (x_3, x_4, x_8, x_7) in one of the two graphs in Fig. 11.5. It contains the chord (x_3, x_8) and, hence, is not chord-free. Now assume that we wish to assign TRUE to all edges in this cycle other than (x_3, x_4). If (x_3, x_8) is assigned TRUE, then the assignment to the simple chord-free cycle (x_3, x_4, x_8) contradicts transitivity. If (x_3, x_8) is assigned FALSE, then the assignment to the simple chord-free cycle (x_3, x_7, x_8) contradicts transitivity. Thus, the constraints over the chord-free cycles are sufficient for preventing the transitivity-violating assignment to the cycle that includes a chord. ▟

The number of simple chord-free cycles in a graph can still be exponential in the number of vertices. Hence, building $\mathcal{B}_{trans}$ such that it directly constrains every such cycle can make the size of this formula exponential in the number of variables. Luckily, we have:

Definition 11.19 (chordal graphs). *A* chordal graph *is an undirected graph in which no cycle of size 4 or more is chord-free.*

Every graph can be made chordal in a time polynomial in the number of vertices.[2] Since the only chord-free cycles in a chordal graph are triangles, this implies that applying Theorem 11.17 to such a graph results in a formula

[2] We simply remove all vertices from the graph one by one, each time connecting the neighbors of the eliminated vertex if they were not already connected. The original graph plus the edges added in this process is a chordal graph.

of size not more than cubic in the number of variables (three constraints for each triangle in the graph). The newly added chords are represented by new variables that appear in $\mathcal{B}_{trans}$ but not in $e(\varphi^{\mathrm{E}})$. Algorithm 11.5.1 summarizes the steps of this method.

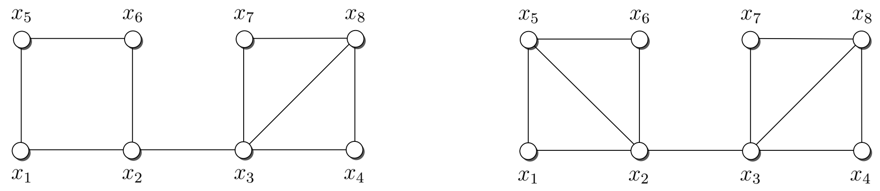


Fig. 11.5. A nonchordal nonpolar equality graph corresponding to φ^{E} (*left*), and a possible chordal version of it (*right*)

```
Algorithm 11.5.1: EQUALITY-LOGIC-TO-PROPOSITIONAL-LOGIC

Input:  An equality formula $\varphi^{E}$
Output: A propositional formula equisatisfiable with $\varphi^{E}$

1. Construct a Boolean formula $e(\varphi^{E})$ by replacing each atom of the form
   $x_i = x_j$ in $\varphi^{E}$ with a Boolean variable $e_{i,j}$.
2. Construct the nonpolar equality graph $G^{E}_{NP}(\varphi^{E})$.
3. Make $G^{E}_{NP}(\varphi^{E})$ chordal.
4. $\mathcal{B}_{trans}$ := TRUE.
5. For each triangle $(e_{i,j}, e_{j,k}, e_{i,k})$ in $G^{E}_{NP}(\varphi^{E})$,
```

$$\begin{aligned} \mathcal{B}_{trans} := \mathcal{B}_{trans} \wedge &\\ (e_{i,j} \wedge e_{j,k} &\implies e_{i,k}) \wedge \\ (e_{i,j} \wedge e_{i,k} &\implies e_{j,k}) \wedge \\ (e_{i,k} \wedge e_{j,k} &\implies e_{i,j}) \ . \end{aligned} \tag{11.42}$$

```
6. Return $e(\varphi^{E}) \wedge \mathcal{B}_{trans}$.
```

Example 11.20. Figure 11.5 depicts a nonpolar equality graph before and after making it chordal. We use solid edges, but note that these *should not be confused with the solid edges in (polar) equality graphs*, where they denote disequalities. After the graph has been made chordal, it contains four triangles and, hence, $\mathcal{B}_{trans}$ conjoins 12 constraints. For example, for the triangle (x_1, x_2, x_5), the constraints are

$$\begin{aligned} e_{1,2} \wedge e_{2,5} &\implies e_{1,5} \ , \\ e_{1,5} \wedge e_{2,5} &\implies e_{1,2} \ , \\ e_{1,2} \wedge e_{1,5} &\implies e_{2,5} \ . \end{aligned} \tag{11.43}$$

The added edge $e_{2,5}$ corresponds to a new auxiliary variable $e_{2,5}$ that appears in $\mathcal{B}_{trans}$ but not in $e(\varphi^{\mathrm{E}})$. ■

There exists a version of this algorithm that is based on the (polar) equality graph, and generates a smaller number of transitivity constraints. See Problem 11.7 for more details.

11.6 Equalities and Small-Domain Instantiations

In this section, we show a method for solving equality logic formulas by relying on the **small-model property** that this logic has. This means that every satisfiable formula in this logic has a model (a satisfying interpretation) of finite size. Furthermore, in equality logic there is a computable bound on the size of such a model. We use the following definitions in the rest of the discussion.

Definition 11.21 (adequacy of a domain for a formula). *A domain is* adequate for a formula *if the formula either is unsatisfiable or has a model within this domain.*

Definition 11.22 (adequacy of a domain for a set of formulas). *A domain is* adequate for a set of formulas *if it is adequate for each formula in the set.*

In the case of equality logic, each set of formulas with the same number of variables has an easily computable adequate finite domain, as we shall soon see. The existence of such a domain immediately suggests a decision procedure: simply enumerate all assignments within this domain and check whether one of them satisfies the formula. Our solution strategy, therefore, for checking whether a given equality formula φ^{E} is satisfiable, can be summarized as follows:

1. Determine, in polynomial time, a **domain allocation**

$$D : var(\varphi^{\mathrm{E}}) \mapsto 2^{\mathbb{N}} \tag{11.44}$$

$\boxed{var}$ $\boxed{D}$ $\boxed{D(x_i)}$

(where $var(\varphi^{\mathrm{E}})$ denotes the set of variables of φ^{E}), by mapping each variable $x_i \in var(\varphi^{\mathrm{E}})$ into a finite set of integers $D(x_i)$, such that φ^{E} is satisfiable if and only if it is satisfiable within D (i.e., there exists a satisfying assignment in which each variable x_i is assigned an integer from $D(x_i)$).

2. Encode each variable x_i as an enumerated type over its finite domain $D(x_i)$. Construct a propositional formula representing φ^{E} under this finite domain, and use SAT to check if this formula is satisfiable.

This strategy is called **small-domain instantiation**, since we instantiate the variables with a finite set of values from the domain computed, each time checking whether it satisfies the formula. The number of instantiations in the worst case is what we call the size of the **state space** spanned by a domain. The size of the state space of a domain D, denoted by $|D|$, is equal to the product of the numbers of elements in the domains of the individual variables. Clearly, the success of this method depends on its ability to find domain allocations with small state spaces.

$\boxed{|D|}$

11.6.1 Some Simple Bounds

We now show several bounds on the number of elements in an adequate domain. Let $\varPhi_n$ be the (infinite) set of all equality logic formulas with n variables and without constants.

$\boxed{\varPhi_n}$

Theorem 11.23 (folk theorem). *The uniform domain allocation $\{1, \ldots, n\}$ for all n variables is adequate for $\varPhi_n$.*

Proof. Let $\varphi^{\mathrm{E}} \in \varPhi_n$ be a satisfiable equality logic formula. Every satisfying assignment α to φ^{E} reflects a partition of its variables into equivalence classes. That is, two variables are in the same equivalence class if and only if they are assigned the same value by α. Since there are only equalities and disequalities in φ^{E}, every assignment which reflects the same equivalence classes satisfies exactly the same predicates as α. Since all partitions into equivalence classes over n variables are possible in the domain $1, \ldots, n$, this domain is adequate for φ^{E}. ⌟

This bound, although not yet tight, implies that we can encode each variable in a $\varPhi_n$ formula with no more than $\lceil \log n \rceil$ bits, and with a total of $n\lceil \log n \rceil$ bits for the entire formula in the worst case. This is very encouraging, because it is already better than the worst-case complexity of Algorithm 11.5.1, which requires $n \cdot (n-1)/2$ bits (one bit per pair of variables) in the worst case.

The domain $1, \ldots, n$, as suggested above, results in a state space of size n^n. We can do better if we do not insist on a uniform domain allocation, which allocates the same domain to all variables.

Theorem 11.24. *Assume for each formula $\varphi^{\mathrm{E}} \in \varPhi_n$, $var(\varphi^{\mathrm{E}}) = \{x_1, \ldots, x_n\}$. The domain allocation $D := \{x_i \mapsto \{1, \ldots, i\} \mid 1 \leq i \leq n\}$ is adequate for $\varPhi_n$.*

Proof. As argued in the proof of Theorem 11.23, every satisfying assignment α to $\varphi^{\mathrm{E}} \in \varPhi_n$ reflects a partition of the variables into equivalence classes. We construct an assignment α' as follows:

For each equivalence class C:

Aside: The Complexity Gap

Why is there a complexity gap between domain allocation and the encoding method that we described in Sect. 11.5? Where is the wasted work in EQUALITY-LOGIC-TO-PROPOSITIONAL-LOGIC? Both algorithms merely partition the variables into classes of equal variables, but they do it in a different way. Instead of asking "which *subset* of $\{v_1, \ldots, v_n\}$ is each variable equal to?", with the domain-allocation technique we ask instead "which *value* in the range $\{1, \ldots, n\}$ is each variable equal to?". For each variable, rather than exploring the range of subsets of $\{v_1, \ldots, v_n\}$ to which it may be equal, we instead explore the range of values $\{1, \ldots, n\}$. The former requires one bit per element in this set, or a total of n bits, while the latter requires only $\log n$ bits.

- Let x_i be the variable with the lowest index in C.
- Assign i to all the variables in C.

Since all the other variables in C have indices higher than i, i is in their domain, and hence this assignment is feasible. Since each variable appears in exactly one equivalence class, every class of variables is assigned a different value, which means that α' satisfies the same equality predicates as α. This implies that α' satisfies φ^{E}. ▟

The adequate domain suggested in Theorem 11.24 has a smaller state space, of size $n!$. In fact, it is conjectured that $n!$ is also a lower bound on the size of domain allocations adequate for this class of formulas.

Let us now consider the case in which the formula contains constants.

$\boxed{\Phi_{n,k}}$

Theorem 11.25. *Let $\Phi_{n,k}$ be the set of equality logic formulas with n variables and k constants. Assume, without loss of generality, that the constants are $c_1 < \cdots < c_k$. The domain allocation*

$$D := \{x_i \mapsto \{c_1, \ldots, c_k, c_k + 1, \ldots, c_k + i\} \mid 1 \leq i \leq n\} \tag{11.45}$$

is adequate for $\Phi_{n,k}$.

The proof is left as an exercise (Problem 11.8).

The adequate domain suggested in Theorem 11.25 results in a state space of size $(k + n)!/k!$. As stated in Sect. 4.1.3, constants can be eliminated by adding more variables and constraints (k variables in this case), but note that this would result in a larger state space.

The next few sections are dedicated to an algorithm that reduces the allocated domain further, based on an analysis of the equality graph associated with the input formula.

— — —

Sects. 11.6.2, 11.6.3, and 11.6.4 cover advanced topics.

11.6.2 Graph-Based Domain Allocation

The formula sets Φ_n and $\Phi_{n,k}$ utilize only a simple structural characteristic common to all of their members, namely the number of variables and constants. As a result, they group together many formulas of radically different nature. It is not surprising that the best size of adequate domain allocation for the whole set is so high. By paying attention to additional structural similarities of formulas, we can form smaller sets of formulas and obtain much smaller adequate domain allocations.

$\boxed{\Phi(e)}$ $\boxed{E(\varphi^{\mathrm{E}})}$

As before, we assume that φ^{E} is given in negation normal form. Let e denote a set of equality literals and $\Phi(e)$ the set of all equality logic formulas whose literals set is equal to e. Let $E(\varphi^{\mathrm{E}})$ denote the set of φ^{E}'s literals. Thus, $\Phi(E(\varphi^{\mathrm{E}}))$ is the set of all equality logic formulas that have the same set of literals as φ^{E}. Obviously, $\varphi^{\mathrm{E}} \in \Phi(E(\varphi^{\mathrm{E}}))$. Note that $\Phi(e)$ can include both satisfiable and unsatisfiable formulas. For example, let e be the set

$$\{x_1 = x_2, x_1 \neq x_2\} \ . \tag{11.46}$$

Then $\Phi(e)$ includes both the satisfiable formula

$$x_1{=}x_2 \lor x_1{\neq}x_2 \tag{11.47}$$

and the unsatisfiable formula

$$x_1{=}x_2 \land x_1{\neq}x_2 \ . \tag{11.48}$$

An adequate domain, recall, is concerned only with the *satisfiable* formulas that can be constructed from literals in the set. Thus, we should not worry about (11.48). We should, however, be able to satisfy (11.47), as well as formulas such as $x_1 = x_2 \land (\text{TRUE} \lor x_1 \neq x_2)$ and $x_1 \neq x_2 \land (\text{TRUE} \lor x_1 = x_2)$. One adequate domain for the set $\Phi(e)$ is

$$D := \ \{x_1 \mapsto \{0\},\ x_2 \mapsto \{0, 1\}\} \ . \tag{11.49}$$

It is not hard to see that this domain is minimal, i.e., there is no adequate domain with a state space smaller than 2 for $\Phi(e)$.

How do we know, then, which subsets of the literals in $E(\varphi^{\mathrm{E}})$ we need to be able to satisfy within the domain D, in order for D to be adequate for $\Phi(E(\varphi^{\mathrm{E}}))$? The answer is that we need only to be able to satisfy *consistent* subsets of literals, i.e., subsets for which the conjunction of literals in each of them is satisfiable.

A set e of equality literals is consistent if and only if it does not contain one of the following two patterns:

1. A chain of the form $x_1 = x_2$, $x_2 = x_3$, $\ldots$, $x_{r-1} = x_r$ together with the formula $x_1 \neq x_r$.
2. A chain of the form $c_1 = x_2$, $x_2 = x_3$, $\ldots$, $x_{r-1} = c_r$ where c_1 and c_r represent different constants.

In the equality graph corresponding to e, the first pattern appears as a contradictory cycle (Definition 11.12) and the second as an equality path (Definition 11.10) between two constants.

To summarize, a domain allocation D is adequate for $\Phi(E(\varphi^{\mathrm{E}}))$ if every consistent subset $e \subseteq E(\varphi^{\mathrm{E}})$ is satisfiable within D. Hence, finding an adequate domain for $\Phi(E(\varphi^{\mathrm{E}}))$ is reduced to the following problem:

> Associate with each variable x_i a set of integers $D(x_i)$ such that every consistent subset $e \in E(\varphi^{\mathrm{E}})$ can be satisfied with an assignment from these sets.

We wish to find sets of this kind that are as small as possible, in polynomial time.

11.6.3 The Domain Allocation Algorithm

$\boxed{G^{\mathrm{E}}_{=}}$

$\boxed{G^{\mathrm{E}}_{\neq}}$

Let $G^{\mathrm{E}}(\varphi^{\mathrm{E}})$ be the equality graph (see Definition 11.9) corresponding to φ^{E}, defined by $(V, E_=, E_{\neq})$. Let $G^{\mathrm{E}}_{=}$ and $G^{\mathrm{E}}_{\neq}$ denote two subgraphs of $G^{\mathrm{E}}(\varphi^{\mathrm{E}})$, defined by $(V, E_=)$ and $(V, E_{\neq})$, respectively. As before, we use dashed edges to represent $G^{\mathrm{E}}_{=}$ edges and solid edges to represent $G^{\mathrm{E}}_{\neq}$ edges. A vertex is called *mixed* if it is adjacent to edges in both $G^{\mathrm{E}}_{=}$ and $G^{\mathrm{E}}_{\neq}$.

On the basis of the definitions above, Algorithm 11.6.1 computes an economical domain allocation D for the variables in a given equality formula φ^{E}. The algorithm receives as input the equality graph $G^{\mathrm{E}}(\varphi^{\mathrm{E}})$, and returns as output a domain which is adequate for the set $\Phi(E(\varphi^{\mathrm{E}}))$. Since $\varphi^{\mathrm{E}} \in \Phi(E(\varphi^{\mathrm{E}}))$, this domain is adequate for φ^{E}.

$\boxed{\mathit{char}}$

We refer to the values that were added in steps I.A.2, I.C, II.A.1, and II.B as the *characteristic* values of these vertices. We write $\mathit{char}(x_i) = u_i$ and $\mathit{char}(x_k) = u_{C_=}$. Note that every vertex is assigned a single characteristic value. Vertices that are assigned their characteristic values in steps I.C and II.A.1 are called *individually assigned vertices*, whereas the vertices assigned characteristic values in step II.B are called *communally assigned vertices*. We assume that new values are assigned in ascending order, so that $\mathit{char}(x_i) < \mathit{char}(x_j)$ implies that x_i was assigned its characteristic value before x_j. Consequently, we require that all new values are larger than the largest constant C_{max}. This assumption is necessary only for simplifying the proof in later sections.

$\boxed{MV}$

The description of the algorithm presented above leaves open the order in which vertices are chosen in step II.A.1. This order has a strong impact on the size of the resulting state space. Since the values given in this step are distributed on the graph $G^{\mathrm{E}}_{=}$ in step II.A.2, we would like to keep this set as small as possible. Furthermore, we would like to partition the graph quickly, in order to limit this distribution. A rather simple yet effective heuristic for this purpose is to choose vertices according to a greedy criterion, where mixed vertices are chosen in descending order of their degree in $G^{\mathrm{E}}_{\neq}$. We denote the set of vertices chosen in step II.A.1 by MV, to remind ourselves that they are

Algorithm 11.6.1: DOMAIN-ALLOCATION-FOR-EQUALITIES

```
Input:  An equality graph $G^E$
Output: An adequate domain (in the form of a set of integers for each
        variable-vertex) for the set of formulas over literals that are
        represented by $G^E$ edges

I. Eliminating constants and preprocessing
Initially, $D(x_i) = \emptyset$ for all vertices $x_i \in G^E$.
A. For each constant-vertex $c_i$ in $G^E$, do:
    1. (Empty item, for the sake of symmetry with step II.A.)
    2. Assign $D(x_j) := D(x_j) \cup \{c_i\}$ for each vertex $x_j$, such that there is an
       equality path from $c_i$ to $x_j$ not through any other constant-vertex.
    3. Remove $c_i$ and its adjacent edges from the graph.
B. Remove all $G^E_{\neq}$ edges that do not lie on a contradictory cycle.
C. For every singleton vertex (a vertex comprising a connected component by
   itself) $x_i$, add to $D(x_i)$ a new value $u_i$. Remove $x_i$ and its adjacent edges
   from the graph.
II. Value allocation
A. While there are mixed vertices in $G^E$ do:
    1. Choose a mixed vertex $x_i$. Add $u_i$, a new value, to $D(x_i)$.
    2. Assign $D(x_j) := D(x_j) \cup \{u_i\}$ for each vertex $x_j$, such that there is an
       equality path from $x_i$ to $x_j$.
    3. Remove $x_i$ and its adjacent edges from the graph.
B. For each (remaining) connected $G^E_=$ component $C_=$, add a common new
   value $u_{C_=}$ to $D(x_k)$, for every $x_k \in C_=$.

Return $D$.
```

mixed vertices.

Example 11.26. We wish to check whether (11.27), replicated below, is satisfiable:

$$\neg\varphi^{\mathrm{E}} := \left(\begin{array}{c} (x_1 \neq x_2 \lor y_1 \neq y_2 \lor f_1 = f_2) \land \\ (u_1 \neq f_1 \lor u_2 \neq f_2 \lor g_1 = g_2) \land \\ u_1 = f_1 \land u_2 = f_2 \land z = g_1 \end{array} \right) \land z \neq g_2 \,. \tag{11.50}$$

The sets $E_=$ and $E_{\neq}$ are

$$\begin{array}{ll} E_= := & \{(f_1 = f_2),\ (g_1 = g_2),\ (u_1 = f_1),\ (u_2 = f_2),\ (z = g_1)\} \\ E_{\neq} := & \{(x_1 \neq x_2),\ (y_1 \neq y_2),\ (u_1 \neq f_1),\ (u_2 \neq f_2),\ (z \neq g_2)\}\,, \end{array} \tag{11.51}$$

and the corresponding equality graph $G^{\mathrm{E}}(\neg\varphi^{\mathrm{E}})$ reappears in Fig. 11.6.

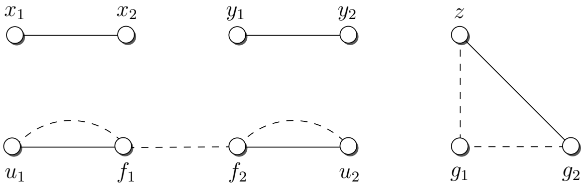


Fig. 11.6. The equality graph $G^{\mathrm{E}}(\neg\varphi^{\mathrm{E}})$

We refrain in this example from applying preprocessing, in order to make the demonstration of the algorithm more informative and interesting. This example results in a state space of size 11^{11} if we use the domain $\{1,\ldots,n\}$ as suggested in Theorem 11.23, and a state space of size 11! ($\approx 4 \times 10^7$) if we use the domain suggested in Theorem 11.24. Applying Algorithm 11.6.1, on the other hand, results in an adequate domain spanning a state space of size 48, as can be seen in Fig. 11.7.

Step	x_1	x_2	y_1	y_2	u_1	f_1	f_2	u_2	g_2	z	g_1	Removed
I.B												edges $(x_1 - x_2)$, $(y_1 - y_2)$
I.C	0	1	2	3								x_1, x_2, y_1, y_2
II.A					4	4	4	4				f_1
II.A							5	5				f_2
II.A									6	6	6	g_2
II.B					7							
II.B								8				
II.B										9	9	
Final D-sets	0	1	2	3	4, 7	4	4, 5	4, 5, 8	6	6, 9	6, 9	State space = 48

Fig. 11.7. Application of Algorithm 11.6.1 to (11.50)

Using a small improvement concerning the new values allocated in step II.A.1, this allocation can be reduced further, down to a domain of size 16. This improvement is the subject of Problem 11.12.

For demonstration purposes, consider a formula φ^{E} where g_1 is replaced by the constant "3". In this case the component (z, g_1, g_2) is handled as follows: In step I.A, "3" is added to $D(g_2)$ and $D(z)$. The edge (z, g_2), now no longer part of a contradictory cycle, is then removed in step I.B and a distinct new value is added to each of these variables in step I.C. ▟

Algorithm 11.6.1 is polynomial in the size of the input graph: steps I.A and II.A are iterated a number of times not more than the number of vertices in the graph; step I.B is iterated not more than the number of edges in $G^{\mathrm{E}}_{\neq}$; steps I.A.2, I.B, II.A.2, and II.B can be implemented with depth-first search (DFS).

11.6.4 A Proof of Soundness

In this section, we argue for the soundness of Algorithm 11.6.1. We begin by describing a procedure which, given the allocation D produced by this algorithm and a consistent subset e, assigns to each variable $x_i \in G^{\mathrm{E}}$ an integer value $a_e(x_i) \in D(x_i)$. We then continue by proving that this assignment satisfies the literals in e. $\boxed{a_e}$

An Assignment Procedure

Given a consistent subset of literals e and its corresponding equality graph $G^{\mathrm{E}}(e)$, assign to each variable-vertex $x_i \in G^{\mathrm{E}}(e)$ a value $a_e(x_i) \in D(x_i)$, according to the following rules:

R1 If x_i is connected by a (possibly empty) $G^{\mathrm{E}}_{=}(e)$-path to an individually assigned vertex x_j, assign to x_i the minimal value of $char(x_j)$ among such x_j's.

R2 Otherwise, assign to x_i its communally assigned value $char(x_i)$.

To see why all vertices are assigned a value by this procedure, observe that every vertex is allocated a characteristic value before it is removed. This can be an individual characteristic value allocated in steps I.C and II.A.1, or a communal value allocated in step II.B. Every vertex x_i that has an individual characteristic value can be assigned a value $a_e(x_i)$ by **R1**, because it has at least the empty equality path leading to an individually allocated vertex, namely itself. All other vertices are allocated a communal value that makes them eligible for a value assignment by **R2**.

Example 11.27. Consider the D-sets in Fig. 11.7. Let us apply the above assignment procedure to a consistent subset e that contains all edges, excluding the two edges between u_1 and f_1, the dashed edge between g_1 and g_2, and the solid edge between f_2 and u_2 (see Fig. 11.8).

The assignment is as follows:

- By **R1**, x_1, x_2, y_1, and y_2 are assigned the characteristic values “0”, “1”, “2”, and “3”, respectively, which they received in step I.C.
- By **R1**, f_1, f_2, and u_2 are assigned the value $char(f_1) =$“4”, because f_1 was the first mixed vertex in the subgraph $\{f_1,\, f_2,\, u_2\}$ that was removed in step II.A, and consequently it has the minimal characteristic value.

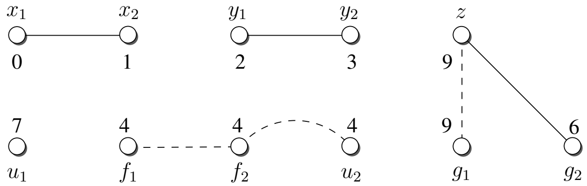


Fig. 11.8. The consistent set of edges e considered in Example 11.27 and the values assigned by the assignment procedure

- By **R1**, g_2 is assigned the value $char(g_2) =$"6", which it received in step II.A.
- By **R2**, z and g_1 are assigned the value "9", which they received in step II.B.
- By **R2**, u_1 is assigned the value "7", which it received in step II.B.

■

Theorem 11.28. *The assignment procedure is feasible (i.e., the value assigned to a node by the procedure belongs to its D-set).*

Proof. Consider first the two classes of vertices that are assigned a value by **R1**. The first class includes vertices that are removed in step I.C. These vertices have only one (empty) $G^{\mathrm{E}}_{=}(e)$-path to themselves, and are therefore assigned the characteristic value that they received in that step. The second class includes vertices that have a (possibly empty) $G^{\mathrm{E}}_{=}(e)$-path to a vertex from MV. Let x_i denote such a vertex, and let x_j be the vertex with the minimal characteristic value that x_i can reach on $G^{\mathrm{E}}_{=}(e)$. Since x_i and all the vertices on this path were still part of the graph when x_j was removed in step II.A, then $char(x_j)$ was added to $D(x_i)$ according to step II.A.2. Thus, the assignment of $char(x_j)$ to x_i is feasible.

Next, consider the vertices that are assigned a value by **R2**. Every vertex that was removed in step I.C or II.A is clearly assigned a value by **R1**. All the other vertices were communally assigned a value in step II.B. In particular, the vertices that do not have a path to an individually assigned vertex were assigned such a value. Thus, the two steps of the assignment procedure are feasible. ■

Theorem 11.29. *If e is a consistent set, then the assignment a_e satisfies all the literals in e.*

Proof. Consider first the case of two variables x_i and x_j that are connected by a $G^{\mathrm{E}}_{=}(e)$-edge. We have to show that $a_e(x_i) = a_e(x_j)$. Since x_i and x_j are $G^{\mathrm{E}}_{=}(e)$-connected, they belong to the same $G^{\mathrm{E}}_{=}(e)$-connected component. If they were both assigned a value by **R1**, then they were assigned the minimal value of an individually assigned vertex to which they are both $G^{\mathrm{E}}_{=}(e)$-connected. If, on the other hand, they were both assigned a value by **R2**, then they were assigned the communal value assigned to the $G^{\mathrm{E}}_{=}$ component to which they both belong. Thus, in both cases they are assigned the same value.

Next, consider the case of two variables x_i and x_j that are connected by a $G^{\mathrm{E}}_{\neq}(e)$-edge. To show that $a_e(x_i) \neq a_e(x_j)$, we distinguish three cases:

- If both x_i and x_j were assigned values by **R1**, they must have inherited their values from two distinct individually assigned vertices, because, otherwise, they are both connected by a $G^{\mathrm{E}}_{=}(e)$-path to a common vertex, which together with the (x_i, x_j) $G^{\mathrm{E}}_{\neq}(e)$-edge closes a contradictory cycle, excluded by the assumption that e is consistent.
- If one of x_i, x_j was assigned a value by **R1** and the other acquired its value from **R2**, then since any communal value is distinct from any individually assigned value, $a_e(x_i)$ must differ from $a_e(x_j)$.
- The remaining case is when both x_i and x_j were assigned values by **R2**. The fact that they were not assigned values in **R1** implies that their characteristic values are not individually allocated, but communally allocated. Assume falsely that $a_e(x_i) = a_e(x_j)$. This means that x_i and x_j were allocated their communal values in the same step, II.B, of the allocation algorithm, which implies that they had an equality path between them (moreover, this path was still part of the graph at the beginning of step II.B). Hence, x_i and x_j belong to a contradictory cycle, and the solid edge (x_i, x_j) was therefore still part of $G^{\mathrm{E}}_{=}(e)$ at the beginning of step II.A. According to the loop condition of this step, at the end of this step there are no mixed vertices left, which rules out the possibility that (x_i, x_j) was still part of the graph at that stage. Thus, at least one of these vertices was individually assigned a value in step II.A.1, and, consequently, the component that it belongs to is assigned a value by **R1**, in contradiction to our assumption. ■

Theorem 11.30. *The formula φ^{E} is satisfiable if and only if φ^{E} is satisfiable over D.*

Proof. By Theorems 11.28 and 11.29, D is adequate for $E_{=} \cup E_{\neq}$. Consequently, D is adequate for $\Phi(At(\varphi^{\mathrm{E}}))$, and in particular D is adequate for φ^{E}. Thus, by the definition of adequacy, φ^{E} is satisfiable if and only if φ^{E} is satisfiable over D. ■

11.6.5 Summary

To summarize Sect. 11.6, the domain allocation method can be used as the first stage of a decision procedure for equality logic. In the second stage, the allocated domains can be enumerated by a standard BDD or by a SAT-based tool. Domain allocation has the advantage of not changing (in particular, not increasing) the original formula, unlike the algorithm that we studied in Sect. 11.5. Moreover, Algorithm 11.6.1 is highly effective in practice in allocating very small domains.

11.7 Ackermann's vs. Bryant's Reduction: Where Does It Matter?

We conclude this chapter by demonstrating how the two reductions lead to different equality graphs and hence change the result of applying any of the algorithms studied in this chapter that are based on this equality graph.

Example 11.31. Suppose that we want to check the satisfiability of the following (satisfiable) formula:

$$\varphi^{\mathrm{UF}} := \; x_1 = x_2 \lor (F(x_1) \neq F(x_2) \land \textsc{false}) \; . \tag{11.52}$$

With Ackermann's reduction, we obtain

$$\varphi^{\mathrm{E}} := \; (x_1 = x_2 \implies f_1 = f_2) \land (x_1 = x_2 \lor (f_1 \neq f_2 \land \textsc{false})) \; . \tag{11.53}$$

With Bryant's reduction, we obtain

$$\mathit{flat}^{\mathrm{E}} := \; x_1 = x_2 \lor (F_1^\star \neq F_2^\star \land \textsc{false}) \; , \tag{11.54}$$

$$\begin{aligned} FC^{\mathrm{E}} := \; & F_1^\star = f_1 & \land \\ & F_2^\star = \begin{pmatrix} \text{case } x_1 = x_2 : f_1 \\ \qquad \textsc{true} \quad : f_2 \end{pmatrix} , \end{aligned} \tag{11.55}$$

and, as always,

$$\varphi^{\mathrm{E}} := \; FC^{\mathrm{E}} \land \mathit{flat}^{\mathrm{E}} \; . \tag{11.56}$$

The equality graphs corresponding to the two reductions appear in Fig. 11.9. Clearly, the allocation for the right graph (due to Bryant's reduction) is smaller.

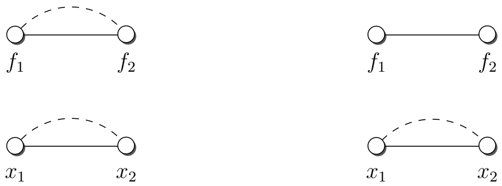


Fig. 11.9. The equality graph corresponding to Example 11.31 obtained with Ackermann's reduction (*left*) and with Bryant's reduction (*right*)

Indeed, an adequate range for the graph on the right is

$$D := \; \{x_1 \mapsto \{0\}, \, x_2 \mapsto \{0, 1\}, \, f_1 \mapsto \{2\}, \, f_2 \mapsto \{3\}\} \; . \tag{11.57}$$

These domains are adequate for (11.56), since we can choose the satisfying assignment

$$\{x_1 \mapsto 0,\ x_2 \mapsto 0,\ f_1 \mapsto 2,\ f_2 \mapsto 3\}\ . \tag{11.58}$$

On the other hand, this domain is not adequate for (11.53).

In order to satisfy (11.53), it must hold that $x_1 = x_2$, which implies that $f_1 = f_2$ must hold as well. But the domains allocated in (11.57) do not allow an assignment in which f_1 is equal to f_2, which means that the graph on the right of Fig. 11.9 is not adequate for (11.53). ▟

So what has happened here? Why does Ackermann's reduction require a larger range?

The reason is that, when two function instances $F(x_1)$ and $F(x_2)$ have equal arguments, in Ackermann's reduction the two variables representing the functions, say f_1 and f_2, are constrained to be equal. But if we force f_1 and f_2 to be different (by giving them a singleton domain composed of a unique constant), this forces FC^{E} to be FALSE, and, consequently φ^{E} to be FALSE. On the other hand, in Bryant's reduction, if the arguments x_1 and x_2 are equal, the terms $F_1^\star$ and $F_2^\star$ that represent the two functions are both assigned the value of f_1. Thus, even if $f_2 \neq f_1$, this does not necessarily make FC^{E} FALSE.

In the bibliographic notes of this chapter, we mention several publications that exploit this property of Bryant's reduction for reducing the allocated range and even constructing smaller equality graphs. It turns out that not all of the edges that are associated with the functional-consistency constraints are necessary, which, in turn, results in a smaller allocated range.

11.8 Problems

Problem 11.1 (practicing Ackermann's and Bryant's reductions). Given the formula

$$\begin{array}{l} F(F(x_1)) \neq F(x_1)\ \wedge \\ F(F(x_1)) \neq F(x_2)\ \wedge \\ x_2 = F(x_1)\ , \end{array} \tag{11.59}$$

reduce its validity problem to a validity problem of an equality logic formula through Ackermann's reduction and Bryant's reduction.

Problem 11.2 (Ackermann's reduction). Extend Algorithm 11.2.1 to multiple function symbols and to functions with multiple arguments.

Problem 11.3 (Bryant's reduction). Suppose that, in Algorithm 11.2.2, the definition of F_i is replaced by

$$F_i^\star = \left(\begin{array}{ll} \text{case } \mathcal{T}^\star(arg(F_1^\star)) = \mathcal{T}^\star(arg(F_i^\star)) & : F_1^\star \\ \quad\vdots & \\ \quad\mathcal{T}^\star(arg(F_{i-1}^\star)) = \mathcal{T}^\star(arg(F_i^\star)) & : F_{i-1}^\star \\ \quad\text{TRUE} & : f_i \end{array}\right), \tag{11.60}$$

the difference being that the terms on the right refer to the $F_j^\star$ variables, $1 \leq j < i$, rather than to the f_j variables. Does this change the value of $F_i^\star$? Prove a negative answer or give an example.

Problem 11.4 (abstraction/refinement). Frequently, the functional-consistency constraints become the bottleneck in the verification procedure, as their number is quadratic in the number of function instances. In such cases, even solving the first iteration of Algorithm 4.4.1 is too hard.

Show an abstraction/refinement algorithm that begins with $\mathit{flat}^{\mathrm{E}}$ and gradually adds functional-consistency constraints.

Hint: note that, given an assignment α' that satisfies a formula with only some of the functional-consistency constraints, checking whether α' respects functional consistency is not trivial. This is because α' does not necessarily refer to all variables (if the formula contains nested functions, some may disappear in the process of abstraction). Hence α' cannot be tested directly against a version of the formula that contains all functional-consistency constraints.

Problem 11.5 (eager encodings for linear arithmetic). (Based on [266]) In Sect. 11.5 we saw eager encoding of equality logic, based on the propositional skeleton and additional constraints to enforce transitivity of equality. It is possible to extend this principle to linear arithmetic, based on Fourier–Motzkin elimination, which was described in Sect. 5.4. The idea is the following, given a propositional combination of linear constraints φ:

- Compute the propositional skeleton $e(\varphi)$.
- Apply Fourier–Motzkin elimination to the set of predicates in φ until all variables are eliminated (i.e., do not stop upon reaching a contradiction). Each time a predicate p is derived from a pair of predicates p_i, p_j, add the constraint $e(p_i) \wedge e(p_j) \implies e(p)$. As a special case, if p_i, p_j are contradictory, then $e(p)$ is the constant FALSE.

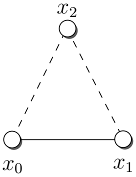
The resulting formula is satisfiable if and only if φ is.

1. Apply this algorithm to the following formula:
$$\begin{aligned}\varphi := \;& (2x_1 - x_2 \leq 0) \wedge \\ & ((2x_2 - 4x_3 \leq 0) \vee (x_3 - x_1 \leq -1) \vee ((0 \leq x_3) \wedge (x_2 \leq 1)))\,.\end{aligned} \tag{11.61}$$
Check that the resulting formula is indeed equisatisfiable with φ.
2. What is the complexity of solving (disjunctive) linear arithmetic this way, given that the input formula has n variables and m linear predicates?

11.8.1 Reductions

Problem 11.6 (correctness of the simplification step). Prove the correctness of Algorithm 11.4.1. You may use the proof strategy suggested in Sect. 11.4.

Problem 11.7 (reduced transitivity constraints). (Based on [194, 247].) Consider the equality graph in Fig. 11.10. The sparse method generates $\mathcal{B}_{trans}$ with three transitivity constraints (recall that it generates three constraints for each triangle in the graph, regardless of the polarity of the edges). Now consider the following claim: the single transitivity constraint $\mathcal{B}_{rtc} = (e_{0,2} \wedge e_{1,2} \implies e_{0,1})$ is sufficient (the subscript *rtc* stands for "reduced transitivity constraints").



	α_{rtc}	α_{trans}
$e_{0,1}$	TRUE	TRUE
$e_{1,2}$	TRUE	TRUE
$e_{0,2}$	FALSE	TRUE

Fig. 11.10. Taking polarity into account allows us to construct a less constrained formula. For this graph, the constraint $\mathcal{B}_{rtc} = (e_{0,2} \wedge e_{1,2} \implies e_{0,1})$ is sufficient. An assignment α_{rtc} that satisfies $\mathcal{B}_{rtc}$ but breaks transitivity can always be "fixed" so that it *does* satisfy transitivity, while still satisfying the propositional skeleton $e(\varphi^{\mathrm{E}})$. The assignment α_{trans} demonstrates such a "fixed" version of the satisfying assignment

To justify this claim, it is sufficient to show that, for every assignment α_{rtc} that satisfies $e(\varphi^{\mathrm{E}}) \wedge \mathcal{B}_{rtc}$, there exists an assignment α_{trans} that satisfies $e(\varphi^{\mathrm{E}}) \wedge \mathcal{B}_{trans}$. Since this, in turn, implies that φ^{E} is satisfiable as well, we obtain the result that φ^{E} is satisfiable if and only if $e(\varphi^{\mathrm{E}}) \wedge \mathcal{B}_{rtc}$ is satisfiable.

We are able to construct such an assignment α_{trans} because of the monotonicity of NNF (see Theorem 1.14, and recall that the polarity of the edges in the equality graph is defined according to their polarity in the NNF representation of φ^{E}). There are only two satisfying assignments to $\mathcal{B}_{rtc}$ that do not satisfy $\mathcal{B}_{trans}$. One of these assignments is shown in the α_{rtc} column in the table to the right of the drawing. The second column shows a corresponding assignment α_{trans}, which clearly satisfies $\mathcal{B}_{trans}$.

However, we still need to prove that every formula $e(\varphi^{\mathrm{E}})$ that corresponds to the above graph is still satisfied by α_{trans} if it is satisfied by α_{rtc}. For example, for $e(\varphi^{\mathrm{E}}) = (\neg e_{0,1} \vee e_{1,2} \vee e_{0,2})$, both $\alpha_{rtc} \models e(\varphi^{\mathrm{E}}) \wedge \mathcal{B}_{rtc}$ and $\alpha_{trans} \models e(\varphi^{\mathrm{E}}) \wedge \mathcal{B}_{trans}$. Intuitively, this is guaranteed to be true because α_{trans} is derived from α_{rtc} by flipping an assignment of a positive (unnegated) predicate ($e_{0,2}$) from FALSE to TRUE. We can equivalently flip an assignment to a negated predicate ($e_{0,1}$ in this case) from TRUE to FALSE.

1. Generalize this example into a claim: given a (polar) equality graph, which transitivity constraints are necessary and sufficient?
2. Show an algorithm that computes the constraints that you suggest in the item above. What is the complexity of your algorithm? (*Hint*: there exists

a polynomial algorithm, which is hard to find. An exponential algorithm will suffice as an answer to this question).

11.8.2 Domain Allocation

Problem 11.8 (adequate domain for $\Phi_{n,k}$). Prove Theorem 11.25.

Problem 11.9 (small-domain allocation). Prove the following lemma:

Lemma 11.32. *If a domain D is adequate for $\Phi(e)$ and $e' \subseteq e$, then D is adequate for $\phi(e')$.*

Problem 11.10 (small-domain allocation: an adequate domain). Prove the following theorem:

Theorem 11.33. *If all the subsets of $E(\varphi^{\mathrm{E}})$ are consistent, then there exists an allocation R such that $|R| = 1$.*

Problem 11.11 (formulation of the graph-theoretic problem). Give a self-contained formal definition of the following decision problem: given an equality graph G and a domain allocation D, is D adequate for G?

Problem 11.12 (small-domain allocation: an improvement to the allocation heuristic). Step II.A.1 of Algorithm 11.6.1 calls for allocation of *distinct* characteristic values to the mixed vertices. The following example proves that this is not always necessary:

Consider the subgraph $\{u_1, f_1, f_2, u_2\}$ of the graph in Fig. 11.3. Application of the basic algorithm to this subgraph may yield the following allocation, where the characteristic values assigned are underlined: $R_1 : u_1 \mapsto \{0, \underline{2}\}, f_1 \mapsto \{\underline{0}\}, f_2 \mapsto \{0, \underline{1}\}, u_2 \mapsto \{0, 1, \underline{3}\}$. This allocation leads to a state space complexity of 12. By relaxing the requirement that all individually assigned characteristic values should be distinct, we can obtain the allocation $R_2 : u_1 \mapsto \{0, \underline{2}\}, f_1 \mapsto \{\underline{0}\}, f_2 \mapsto \{\underline{0}\}, u_2 \mapsto \{0, \underline{1}\}$ with a state-space complexity of 4. This reduces the size of the state space of the entire graph from 48 to 16.

It is not difficult to see that R_2 is adequate for the subgraph considered.

What are the conditions under which it is possible to assign equal values to mixed variables? Change the basic algorithm so that it includes this optimization.

11.9 Bibliographic Notes

The following are some bibliographic details about the development of the eager encoding framework. Lazy encoding frameworks, including DPLL(T), are covered in Sect. 3.6.

Some of the algorithms presented in earlier chapters are in fact eager-style decision procedures. The reduction methods for equality logic that are presented in Sect. 11.5 are such algorithms [57, 194]. A similar procedure for difference logic was suggested by Ofer Strichman, Sanjit Seshia, and Randal Bryant in [267]. Procedures that are based on small-domain instantiation (see Sect. 11.6 and a similar procedure for difference logic in [273]) can also be seen as eager encodings, although the connection is less obvious: the encoding is based not on the skeleton and additional constraints, but rather on an encoding of predicates (equalities, inequalities, etc., depending on the theory) over finite-range variables. The original procedure in [227] used multiterminal BDDs rather than SAT to solve the resulting propositional formula. We should also mention that there are hybrid approaches, combining encodings based on small-domain instantiation and explicit constraints, such as the work by Seshia et al. on difference logic [256].

The first proof-based reduction corresponding to an eager encoding (from integer- and real-valued linear arithmetic) was introduced in [266]. The procedure was not presented as part of a more general framework of using deductive rules as described in this chapter. The proof was generated in an eager manner using Fourier–Motzkin variable elimination for the reals and the Omega test for the integers.

There are only a few publicly available, supported decision procedures based on eager encoding, most notably UCLID [58], which was developed by Randal Bryant, Shuvendu Lahiri, and Sanjit Seshia. There is little research in this field, which makes it hard to determine if the eager approach is inherently inferior to the lazy one or is just not pushed forward as strongly.

We now survey some of the highlights from the history of deciding equality logic with uninterpreted functions. Please also refer to our survey in Sect. 4.7.

We failed to find an original reference for the fact that the range $\{1, \ldots, n\}$ is adequate for formulas with n variables. This is usually referred to as a "folk theorem" in the literature. The work by Hojati, Kuehlmann, German, and Brayton in [147] and Hojati, Isles, Kirkpatrick, and Brayton in [146] was the first, as far as we know, where anyone tried to decide equalities with finite instantiation, while trying to derive a value k, $k \leq n$ that was adequate as well, by analyzing the equality graph. The method presented in Sect. 11.6 was the first to consider a different range for each variable and, hence, is much more effective. It is based on work by Pnueli, Rodeh, Siegel, and Strichman in [227, 228]. These papers suggest that Ackermann's reduction should be used, which results in large formulas, and, consequently, large equality graphs and correspondingly large domains (but much smaller than the range $\{1, \ldots, n\}$).

In [245, 246], Rodeh and Strichman presented a generalization of positive equality that enjoys benefits from both worlds: on the one hand, it does not add all the edges that are associated with the functional-consistency constraints (it adds only a small subset of them based on an analysis of the formula), but on the other hand, it assigns small ranges to *all* variables as in [228] and, in particular, a single value to all the terms that would be assigned a single value by the technique of [56]. This method decreases the size of the equality graph in the presence of uninterpreted functions, and consequently the allocated ranges (for example, it allocates a domain with a state space of size 2 for the running example in Sect. 11.6.3). Rodeh showed in his thesis [245] (also see [226]) an extension of range allocation to *dynamic* range allocation. This means that each variable is assigned not one of several constants, as prescribed by the allocated domain, but rather one of the variables that represent an immediate neighbor in $G^{\mathrm{E}}_{=}$, or a unique constant if it has one or more neighbors in $G^{\mathrm{E}}_{\neq}$. The size of the state space is thus proportional to $\log n$, where n is the number of neighbors.

Bryant, German, and Velev suggested in [56] what we refer to as Bryant's reduction in Sect. 11.2.2. This technique enabled them to exploit what they called the *positive equality* structure in formulas for assigning unique constants to some of the variables and a full range to the others. Using the terminology of this chapter, these variables are adjacent only to solid edges in the equality graph corresponding to the original formula (a graph built *without* referring to the functional-consistency constraints, and hence the problem of a large graph due to Ackermann's constraints disappears). A more robust version of this technique, in which a larger set of variables can be replaced with constants, was later developed by Lahiri, Bryant, Goel, and Talupur [174].

Goel, Sajid, Zhou, Aziz, and Singhal were the first to encode each equality with a new Boolean variable [132]. They built a BDD corresponding to the encoded formula, and then looked for transitivity-preserving paths in the BDD. Bryant and Velev suggested in [57] that the same encoding should be used but added explicit transitivity constraints instead. They considered several translation methods, only the best of which (the sparse method) was presented in this chapter. One of the other alternatives is to add such a constraint for every three variables (regardless of the equality graph). A somewhat similar approach was considered by Zantema and Groote [294]. The sparse method was later superseded by the method of Meir and Strichman [194] and later by that of Rozanov and Strichman [247], where the polar equality graph is considered rather than the nonpolar one, which leads to a smaller number of transitivity constraints. This direction is mentioned in Problem 11.7.

11.10 Glossary

The following symbols were used in this chapter:

Symbol	Refers to ...	First used on page ...
$\mathcal{T}$	A function that transforms an input formula or term by replacing each uninterpreted function F_i with a new variable f_i	247
FC^{E}	Functional-consistency constraints	247
flat^{E}	Equal to $\mathcal{T}(\varphi^{\text{UF}})$ in Ackermann's reduction, and to $\mathcal{T}^\star(\varphi^{\text{UF}})$ in Bryant's reduction	247, 250
$F_i^\star$	In Bryant's reduction, a macro variable representing the case expression associated with the function instance $F_i()$ that was substituted by F_i	250
$\mathcal{T}^\star$	A function similar to $\mathcal{T}$, that replaces each uninterpreted function F_i with $F_i^\star$	250
$At(\varphi^{\text{E}})$	The set of atoms in the formula φ^{E}	253
$E_=, E_{\neq}$	Sets of equality and inequality predicates, and also the edges in the equality graph	253
G^{E}	Equality graph	254
$x =^* y$	There exists an equality path between x and y in the equality graph	254
$x \neq^* y$	There exists a disequality path between x and y in the equality graph	255
$e(\varphi^{\text{E}})$	The propositional skeleton of φ^{E}	259
$\mathcal{B}_{trans}$	The transitivity constraints due to the reduction from φ^{E} to $\mathcal{B}_{sat}$ by the sparse method	259
G^{E}_{NP}	Nonpolar equality graph	259
$var(\varphi^{\text{E}})$	The set of variables in φ^{E}	262
D	A domain allocation function. See (11.44)	262

continued on next page

continued from previous page

Symbol	**Refers to ...**	**First used on page ...**
$\lvert D\rvert$	The state space spanned by a domain	263
$\varPhi_n$	The (infinite) set of equality logic formulas with n variables	263
$\varPhi_{n,k}$	The (infinite) set of equality logic formulas with n variables and k constants	264
$\phi(e)$	The (infinite) set of equality formulas with a set of literals equal to e	265
$E(\varphi^{\mathrm{E}})$	The set of literals in φ^{E}	265
$G^{\mathrm{E}}_{=}, G^{\mathrm{E}}_{\neq}$	The projections of the equality graph on the $E_{=}$ and $E_{\neq}$ edges, respectively	266
$char(v)$	The characteristic value of a node v in the equality graph	266
MV	The set of mixed vertices that are chosen in step II.A.1 of Algorithm 11.6.1	266
$a_e(x)$	An assignment to a variable x from its allocated domain $D(x)$	269

12

Applications in Software Engineering and Computational Biology

Sect. 12.1–12.3 were written with the help of Nikolaj Bjørner and Leonardo de Moura from the *research in software engineering* (Rise) group in Microsoft Research Redmond. Section 12.4 was written by Hillel Kugler, from the Biological Computation Group in Microsoft Research Cambridge.

12.1 Introduction

One cannot overemphasize the importance of eliminating program errors ("bugs"), and the fact that this problem only gets more acute, given the growing prevalence of software in business-related and safety-critical systems.

The traditional method to detect and diagnose software defects is to *test* the program, i.e., the program is executed using a limited set of inputs. This method is typically effective, but cannot guarantee the absence of errors. *Formal verification*, on the other hand, has a much more ambitious goal: the goal is to decide whether *for all possible inputs* a given specification is satisfied by the program. As an example, the specification could require that at a given location in the program no division-by-0 is possible, or that it holds that $x < y$ for two program variables x and y. These specifications are an instance of *reachability*, which is the problem of checking whether a given program state occurs in any execution of the program. The reachability problem is undecidable, which means that there cannot exist an algorithm—a decision procedure—that will always give the correct answer in a finite amount of time. What makes this problem undecidable? The short answer is *unbounded allocation of memory*, because it implies that we cannot bound the number of states that the program can reach (note the aside on the topic). Consequently, software verification is undecidable as well.

Like in the case of many other undecidable problems that are sufficiently important, many partial solutions have been invented through the years, and many of them are used on a daily basis. By "partial" we mean that the type of programs that can be handled is restricted; for other programs the solution

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0_12

Aside: Verification and Finite Memory
One may take the view that the memory of a computer is finite, which is obviously correct and theoretically solves the reachability problem. However, this does not help in practice for automatic reasoning unless it is assumed that the memory bound is very small. It also restricts the proof to a specific bound on the size of the memory. Hence, verification techniques for software typically do not rely on the finiteness of the memory. There is a similar problem with unbounded recursion even if the recursive function does not allocate heap memory, because the stack may grow arbitrarily. The discussion regarding the conditions under which this leads to undecidability is beyond the scope of this book. Finally, we assume that the arithmetic in the program is performed within a finite range, e.g., by encoding numbers with a fixed number of bits.

is incomplete (see Definition 1.6), which means that the verification system may occasionally give up (returning a “don’t-know” answer) or not terminate. *Testing*, for example, can be seen as such a partial solution: typically it cannot declare a program “correct” because it does not try all possible inputs; it can only declare it to be incorrect if it happens to find a test that violates the program’s specification.

A key component in many of these solutions (including automated test generation) is a reasoning engine, and at the core of this engine is a decision procedure. Modern tools use solvers for Satisfiability Modulo Theories (SMT), which are decision procedures based on the architecture described in Chap. 3. A significant industrial application domain for these solvers is the analysis, verification and testing of programs.

It is not trivial to see the connection between programs and logic. Programs are *dynamic*: they execute instructions one at a time, reuse variables, allocate memory, and so on. Decision procedures for quantifier-free first-order theories are *static*: they can only check whether there exists a *simultaneous* assignment to the variables that satisfies a given logical formula. Bridging this gap is one of the topics of this chapter. Whereas our emphasis is on modern techniques that aim at fully automatic reasoning, we note that the assignment of logical meaning to programs dates back to seminal works of Floyd [116] and Hoare [144] from the 1960s, which were focused on manual proofs.

As pointed out earlier, unbounded loops with memory allocation are the reason for the undecidability of the software verification problem. One way to avoid undecidability is to impose a bound on the loops by discarding all executions in which a loop is iterated more than a predetermined number of times. This means that we examine a new program that *underapproximates* the original program. Using such restrictions (or in case the program is bounded to begin with), it is possible to build a logical formula that represents the input–output relation of the program. This can be done with the help of a representation called *static single assignment* (SSA) form. In Sect. 12.2 we will see techniques based on this representation for bridging the gap between a bounded sequence of instructions and quantifier-free first-order formulas.

We will also study a technique that *overapproximates* the original program, by using *abstraction*. This technique may generate false alarms, but in return is frequently able to reason about unbounded program executions. We will present such techniques in Sect. 12.3.

12.2 Bounded Program Analysis

12.2.1 Checking Feasibility of a Single Path

We will begin by describing a basic building block of many program analyzers: a method to determine whether the program can execute a given path. An **execution path** is a sequence of program instructions executed during a run of a program. The path can be *partial*, meaning that it does not have to reach an exit point of the program.

An **execution trace** is a sequence of *states* that are observed along an execution path. There can be many such traces along a single path, corresponding to different inputs. For each path we can build a formula that represents it: the satisfying assignments to this formula correspond to traces along this path. This symbolic representation of traces is critical in many applications, and techniques that use it are said to be based on (path-based) **symbolic simulation**. Popular uses of symbolic simulation include *automatic test generation*, *detection of dead code*, and *verification* of properties given in the form of **assertions**. An assertion is a program instruction that takes a condition as argument, and if the condition evaluates to FALSE, it reports an error and aborts. Verifying an assertion means proving that for all inputs the condition of the assertion evaluates to TRUE.

We will use Program 12.2.1 as a running example of verifying assertions. It contains the definition of a procedure `ReadBlocks`. The program is artificial, but it exemplifies low-level parsing code that is typically used when processing file formats, such as the media formats JPEG, AVI or MPEG.

The input array `data` encodes implicitly a sequence of blocks of data. The first element of each block contains the index of the next block. `ReadBlocks` is supposed to process all the data in these blocks, while skipping data that is equal to the input parameter `cookie`. We will assume that the variable N denotes the number of elements in the array.

The example contains two array-bounds errors, which means that the index is outside of the range $0, \ldots, (N-1)$. In order to detect them, we will generate an assertion for each access to the array `data`. In the case of our example program, we will add assertions that check that the array index is within range.

For now, assume that the verification tool has some heuristic to choose an execution path—later we will see that there is no need to focus on a single

Program 12.2.1 Reading blocks from an array

```
 1 void ReadBlocks(int data[], int cookie)
 2 {
 3   int i = 0;
 4   while (true)
 5   {
 6     int next;
 7     next = data[i];
 8     if (!(i < next && next < N)) return;
 9     i = i + 1;
10     for (; i < next; i = i + 1) {
11         if (data[i] == cookie)
12            i = i + 1;
13         else
14            Process(data[i]);
15     }
16   }
17 }
```

path because multiple paths can be checked simultaneously. Suppose, then, that the heuristic picks the following path:

1. Begin at Line 3.
2. Run through the `for` loop once, take the `else` branch, and then exit the `for` loop.
3. Exit the `while` loop during the second iteration in Line 8.

Table 12.1 shows the sequence of instructions that corresponds to this execution. The line numbers correspond to the original program. In the rightmost column we record the instructions. Every time the "then" branch of an `if` statement is taken along the path, we record the *guard*, which is the condition of the "if". If it is the "else" branch that is taken, we record its negation.

As a second step, we rewrite these instructions and conditions into the **static single assignment** (SSA) representation. In SSA we create a "time-stamped version" of the program variables: every time a variable is written, a new symbol is introduced for it.

Let us illustrate the idea with the trace given as Table 12.1. We will ignore the call to the procedure `Process`, which we here assume does not change the values of `data`. The SSA form of the trace is shown as Table 12.2. The values of the inputs are time-stamped with 0, hence `data` and `cookie` are renamed to $\texttt{data}_0$ and $\texttt{cookie}_0$, respectively. The first executable statement in `ReadBlocks` is the initialization of `i`. The value of `i` after this assignment is denoted by $\texttt{i}_1$. Every time a variable is read, we rename it by adding the

Line	Kind	Instruction or condition
3	Assignment	`i = 0;`
7	Assignment	`next = data[i];`
8	Branch	`i < next && next < N`
9	Assignment	`i = i + 1;`
10	Branch	`i < next`
11	Branch	`data[i] != cookie`
14	Function call	`Process(data[i]);`
10	Assignment	`i = i + 1;`
10	Branch	`!(i < next)`
7	Assignment	`next = data[i];`
8	Branch	`!(i < next && next < N)`

Table 12.1. Sequence of statements along a path of Program 12.2.1

current timestamp to its identifier. Every time the variable is assigned, we increment the timestamp by one, and only then rename the variable.

Line	Kind	Instruction or condition
3	Assignment	$i_1 = 0;$
7	Assignment	$next_1 = data_0[i_1];$
8	Branch	$i_1 < next_1 \&\& next_1 < N_0$
9	Assignment	$i_2 = i_1 + 1;$
10	Branch	$i_2 < next_1$
11	Branch	$data_0 [i_2] != cookie_0$
14	Function call	$Process(data_0 [i_2]);$
10	Assignment	$i_3 = i_2 + 1;$
10	Branch	$!(i_3 < next_1)$
7	Assignment	$next_2 = data_0 [i_3];$
8	Branch	$!(i_3 < next_2 \&\& next_2 < N_0)$

Table 12.2. SSA form of the trace from Table 12.1

The SSA form can now be translated into a logical formula, which is called the **path constraint**. We obtain this formula by replacing the assignments with equalities, and including all branch conditions as conjuncts. The formula generated in this way is

$$\begin{aligned} \mathit{ssa} \iff & i_1 = 0 && \wedge \\ & \mathit{next}_1 = \mathit{data}_0[i_1] && \wedge \\ & (i_1 < \mathit{next}_1 \;\wedge\; \mathit{next}_1 < N_0) && \wedge \\ & i_2 = i_1 + 1 && \wedge \\ & i_2 < \mathit{next}_1 && \wedge \\ & \mathit{data}_0[i_2] \neq \mathit{cookie}_0 && \wedge \\ & i_3 = i_2 + 1 && \wedge \\ & \neg(i_3 < \mathit{next}_1) && \wedge \\ & \mathit{next}_2 = \mathit{data}_0[i_3] && \wedge \\ & \neg(i_3 < \mathit{next}_2 \;\wedge\; \mathit{next}_2 < N_0) \;. \end{aligned} \tag{12.1}$$

Note that the equality symbol in the formula denotes mathematical equality, whereas it denotes an assignment in Table 12.2.

All valuations of the inputs data_0 and cookie_0 that satisfy the formula ssa above correspond to a trace for the path given in Table 12.1.

Assertion Checking

Now consider a path that leads to an assertion. We can use the corresponding path constraint for checking whether that assertion can be violated. In order to do this, we need to add the *negation* of the assertion to the path constraint.

Consider again Program 12.2.1, and specifically the path that executes the assignment in Line 3 and then checks that the variable `i` is within the required range for the array access in Line 7. The corresponding constraint is

$$i_1 = 0 \quad \wedge \quad \neg(0 \le i_1 \wedge i_1 < N_0) \;. \tag{12.2}$$

If (12.2) is satisfiable, then the assertion can be violated, and we can decide whether this is the case by applying an appropriate decision procedure. Since we are given a C program, i_1 is a bit vector and N_0 can be modeled with a bit vector[1] as well—we discussed decision procedures for bit vectors in Chap. 6. This formula is satisfied by the assignment

$$\{i_1 \mapsto 0,\; N_0 \mapsto 0\} \;,$$

i.e., `data` is an array of length 0, which means that we just found a potential error. To summarize, we reduced the problem of verifying the correctness of a path in a program to a problem of checking the satisfiability of a formula. Such a formula is accordingly called the **verification condition** (VC) corresponding to our verification problem.

The program has an additional, more subtle error. The path that leads to this violation is given in Table 12.3.

We now translate this path into a path constraint, and add the negation of the last assertion on the path:

[1] According to the C standard, an array has a bounded number of elements. Furthermore, an index of size `size_t` (typically 32 or 64 bits) is sufficient to access all of them.

Line	Kind	Instruction or condition
3	Assignment	`i = 0;`
7	Assignment	`next = data[i];`
8	Branch	`i < next && next < N`
9	Assignment	`i = i + 1;`
10	Branch	`i < next`
11	Branch	`data[i] = cookie`
12	Assignment	`i = i + 1;`
10	Assignment	`i = i + 1;`
10	Branch	`!(i < next)`
7	Assertion	`0 <= i && i < N`

Table 12.3. A second path in Program 12.2.1

$$\begin{array}{lc} i_1 = 0 & \wedge \\ \mathit{next}_1 = \mathit{data}_0[i_1] & \wedge \\ (i_1 < \mathit{next}_1 \;\wedge\; \mathit{next}_1 < N_0) & \wedge \\ i_2 = i_1 + 1 & \wedge \\ i_2 < \mathit{next}_1 & \wedge \\ \mathit{data}_0[i_2] = \mathit{cookie}_0 & \wedge \\ i_3 = i_2 + 1 & \wedge \\ i_4 = i_3 + 1 & \wedge \\ \neg(i_4 < \mathit{next}_1) & \wedge \\ \neg(0 \le i_4 \;\wedge\; i_4 < N_0) & \end{array} \tag{12.3}$$

This formula includes constraints over the input array, which requires combining the bit-vector decision procedure (Chap. 6) with the array decision procedure (Chap. 7). The formula is satisfied, for example, by the assignment

$$\begin{array}{l} \{i_1 \mapsto 0,\ N_0 \mapsto 3,\ \mathit{next}_1 \mapsto 2,\ \mathit{data}_0 \mapsto \langle 2,6,5\rangle, \\ \ i_2 \mapsto 1,\ \mathit{cookie}_0 \mapsto 6,\ i_3 \mapsto 2,\ i_4 \mapsto 3\}\ , \end{array} \tag{12.4}$$

and thus we just identified an additional error in the code. The fact that the array $\langle 2,6,5\rangle$ does not correspond to a legal media format that can be sent as input to the function—recall that we expect the indices to be increasing since they represent offsets in a list of blocks — is immaterial since the error is there regardless of the last element. If we wish to check this function while enforcing this expectation, we can do so by adding appropriate constraints to (12.3).

12.2.2 Checking Feasibility of All Paths in a Bounded Program

The number of paths through a program can grow exponentially in the number of branches. Hence, an exhaustive analysis of a bounded program, in the style described in the previous section, may require solving an exponential number of decision problems.

Instead, we can generate SSA for bounded *programs* that contain branches (rather than just for a *path*). The SSA can subsequently be converted into a single formula that encodes all possible paths. As a first step, loops are unfolded k times, where k is specified by the user. The second step is to assign the condition of each `if` statement to a new variable; we will use the variable γ for this purpose. It will serve as a convenient shorthand whenever we want to refer to any of the conditions.

$\boxed{\gamma}$

Finally, we identify those program locations in which control-flow branches meet again, i.e., locations at which the control flow *reconverges*. Then, statements are added that assign the correct value to all those variables that have been modified in any of the branches. In the compiler literature, these additional assignments are called ϕ-instructions.

Let us illustrate this idea with the `for` loop in Program 12.2.1. The outcome of unfolding this loop twice ($k = 2$) is shown on the left-hand side of Program 12.2.2. Note that the branch that is implicit in the `for` construct is now an explicit `if` statement. The corresponding SSA form appears on the right. Its construction follows the same principles that we used for individual paths, except for the way that branches are treated: we now introduce the assignment of the `if` condition to the new variable γ. It receives a timestamp just as the other variables. The variable γ is used in the ϕ-instructions.

Program 12.2.2 An unfolding of the `for` loop in Program 12.2.1 and its SSA form with ϕ-instructions

```
 1 if (i < next) {                  |  1 γ_1 = (i_0 < next_0);
 2   if (data[i] == cookie)         |  2 γ_2 = (data_0[i_1] == cookie_0);
 3      i = i + 1;                  |  3 i_1 = i_0 + 1;
 4   else                           |  4
 5      Process(data[i]);           |  5
 6                                  |  6 i_2 = γ_2 ? i_1 : i_0;
 7   i = i + 1;                     |  7 i_3 = i_2 + 1;
 8                                  |  8
 9   if (i < next) {                |  9 γ_3 = (i_3 < next_0);
10     if (data[i] == cookie)       | 10 γ_4 = (data_0[i_3] == cookie_0);
11        i = i + 1;                | 11 i_4 = i_3 + 1;
12     else                         | 12
13        Process(data[i]);         | 13
14                                  | 14 i_5 = γ_4 ? i_4 : i_3;
15     i = i + 1;                   | 15 i_6 = i_5 + 1;
16   }                              | 16 i_7 = γ_3 ? i_6 : i_3;
17 }                                | 17 i_8 = γ_1 ? i_7 : i_0;
```

Consider Line 6 in Program 12.2.2 (left), where the two branches of the second `if` statement reconverge. The only variable written in either branch is `i`. To define the value of `i` after the branch, we use the C notation for a conditional term *con* ? *a* : *b*; this term evaluates to *a* if *c* is true and to *b* otherwise. The value of `i` right after the `if` statement is encoded in Line 6 in SSA form as follows: in case the condition γ_2 holds, we assign $\texttt{i}_1$, which is the value of `i` after executing the "then" branch. Otherwise, we assign $\texttt{i}_0$, which encodes the fact that the value of `i` remains unchanged.

Given the SSA form of the (unfolded) program, we can construct a formula that captures exactly all the possible traces that it can execute. As before, in order to check an assertion in the program, we need to add its negation to the formula. The resulting formula is then finally given to a suitable decision procedure.

12.3 Unbounded Program Analysis

We now study a technique for verifying *unbounded* programs. As mentioned in the introduction to this chapter, this problem is generally undecidable, and therefore there cannot be a decision procedure that solves it for all programs. The technique that we will now see may indeed not terminate or may give up, depending on the details of the implementation. We begin with a very coarse overapproximation, and then make it more precise.

12.3.1 Overapproximation with Nondeterministic Assignments

In Sect. 12.2.2 we have seen a technique that can translate a program into a formula by limiting the depth of loops to a given bound. This *underapproximates* the behavior of the program. It is not possible to conclude that the assertion holds for executions that exceed this bound.

We now introduce a program transformation that turns a program with loops into a program that is loop-free but *overapproximates* the original behavior. When successful, this method enables us to make claims about executions of arbitrary length. We perform the transformation in three steps:

1. For each loop and each program variable that is modified by the loop, add an assignment at the beginning of the loop that assigns a nondeterministic value to the variable.
2. After each loop, add an **assumption** that the negation of the loop condition holds. An assumption is a program statement `assume(`*c*`)` that aborts any path that does not satisfy *c*.
3. Replace each `while` loop with an `if` statement using the condition of the loop as the condition of the `if` statement.

We will illustrate this transformation with a simple example. The left-hand side of Program 12.3.1 gives a small program fragment that includes a `while` loop. The loop iterates until the newline character is found in an array. The

Program 12.3.1 Example of an overapproximating transformation of program loops

```
1 int i=0, j=0;
2
3 while(data[i] != '\n')
4 {
5   i++;
6   j=i;
7 }
8
9 assert(i == j);
```

$\longrightarrow$

```
1  int i=0, j=0;
2
3  if(data[i] != '\n')
4  {
5    i=⋆;
6    j=⋆;
7    i++;
8    j=i;
9  }
10
11 assume(data[i] == '\n');
12
13 assert(i == j);
```

loop therefore has no bound on the number of iterations that can be known apriori.

On the right-hand side we see the program after the transformation described above has been applied. We can now further translate this program into an SSA constraint using the technique that we have seen in Sect. 12.2.2. Note that it is particularly simple to translate the assignments that have a nondeterministically chosen value on the right-hand side. It suffices to increase the SSA counter for the variable that is assigned. The assumption in Line 11 is translated by adding the condition as a conjunct. We then conjoin the SSA constraint with the negation of the assertion and obtain the following formula:

$$\begin{array}{ll}
i_1 = 0 & \wedge \\
j_1 = 0 & \wedge \\
\gamma_1 = (\mathit{data}_0[i_1] \neq \texttt{'\textbackslash n'}) & \wedge \\
i_3 = i_2 + 1 & \wedge \\
j_3 = i_3 & \wedge \\
i_4 = \gamma_1 \mathbin{?} i_3 : i_1 & \wedge \\
j_4 = \gamma_1 \mathbin{?} j_3 : j_1 & \wedge \\
\mathit{data}_0[i_4] = \texttt{'\textbackslash n'} & \wedge \\
i_4 \neq j_4 &
\end{array} \tag{12.5}$$

Note that we use i_2 and not i_1 in the fourth line, because of the nondeterministic assignment. It is left to pass (12.5) to an appropriate decision procedure, which determines in this case that the formula is unsatisfiable. This implies that the assertion holds for an arbitrary number of loop iterations.

While the technique has successfully proven the assertion in this particular example, this is not possible for arbitrary programs. It worked because the correctness of the assertion does not depend on the previous iterations of the loop. We will next see a program for which the abstraction is too coarse, and will then discuss a method to refine the abstraction.

12.3.2 The Overapproximation Can Be Too Coarse

Consider the program fragment in Program 12.3.2. It is an excerpt of a typical Windows device driver. We will use it as an example in which the simple overapproximation of Sect. 12.3.1 does not work. The example program processes a sequence of requests, which are obtained by calling a function named `GetNextRequest`. The call is protected by a *lock* to allow multiple concurrent threads to access the queue data structure where requests are stored. Once a request is dequeued and released, the lock is no longer needed because the data associated with a specific request is not accessed by more than one thread. It is important to release the lock before processing the request, as this can take a long time or acquire different locks. Furthermore, note that it should not be possible to exit the loop without owning the lock. If it were, then the call to `unlock()` right after the loop would release the lock twice. This violates how locks are to be used: `unlock()` should never be called by a thread without that thread owning the lock.

Program 12.3.2 Processing requests using locks

```
do {
  lock();
  old_count = count;
  request = GetNextRequest();
  if (request != NULL) {
    ReleaseRequest(request);
    unlock();
    ProcessRequest(request);
    count = count + 1;
  }
}
while(old_count != count);
unlock();
```

More generally, the locking policy that we would like to assure is illustrated with a state diagram in Fig. 12.1: the goal is to never reach the error state. Thus, the program should alternate strictly between locking and unlocking.

Whereas a locking mechanism only makes sense in a multi-threaded program, for the purpose of verification we will look at a single-threaded version of the program, which is sufficient for checking that this thread does not lock or unlock twice in a row as specified by the state diagram. We will instrument this specification into Program 12.3.2 as follows:

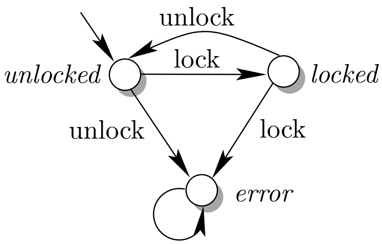


Fig. 12.1. The specification of the lock mechanism given as a state machine

1. We add a variable `state_of_lock`, which reflects the state of the state machine. The variable is initialized with `unlocked`.
2. When the program performs an action that affects the state of the state machine (locking or unlocking), we add assignments to the variable `state_of_lock` accordingly.
3. Finally, we add assertions to the program that capture the case that the state machine transitions to the *error* state.

The program after these transformations is shown in Program 12.3.3. Our goal is to check the locking specification, as expressed in the assertions.

Program 12.3.3 Program 12.3.2 after abstracting the locking mechanism; the locking specification is modeled with assertions

```
state_of_lock = unlocked;
do {
     assert(state_of_lock == unlocked);
     state_of_lock = locked;
     old_count = count;
     request = GetNextRequest();
     if (request != NULL) {
        ReleaseRequest(request);
        assert(state_of_lock == locked);
        state_of_lock = unlocked;
        ProcessRequest(request);
        count = count + 1;
     }
}
while (old_count != count);
assert(state_of_lock == locked);
state_of_lock = unlocked;
```

We will now apply the transformation technique described in the previous section to try to prove the assertions. Program 12.3.4 gives the result of the transformation. Assume for this example that `GetNextRequest()` returns a nondeterministially chosen pointer and that it does not alter the state of the program. Similarly, assume that both `ReleaseRequest` and `ProcessRequest` do not change any of our variables. We can then transform the program into a formula, and pass it to a decision procedure.

Program 12.3.4 Program 12.3.3 after application of the overapproximating transformation

```
 1   state_of_lock = unlocked;
 2
 3   state_of_lock = *;
 4   old_count = *;
 5   count = *;
 6   request = *;
 7
 8   assert(state_of_lock == unlocked);
 9   state_of_lock = locked;
10   old_count = count;
11   request = GetNextRequest();
12   if (request != NULL) {
13     ReleaseRequest(request);
14     assert(state_of_lock == locked);
15     state_of_lock = unlocked;
16     ProcessRequest(request);
17     count = count + 1;
18   }
19
20   assume(old_count == count);
21
22   assert(state_of_lock == locked);
23   state_of_lock = unlocked;
```

We will, however, find that the obtained formula is satisfiable. In particular, there is a trivial counterexample to the first assertion, in which the variable `state_of_lock` is assigned the value `locked` in the beginning of the loop, just before Line 3. Clearly the state cannot be `locked` in the beginning of the loop in the concrete (real) program; it is an artifact of the abstraction. This example shows us that beginning the loop with an arbitrary state is too coarse an abstraction, as it may lead to false alarms. We need to *refine* this abstraction, namely make it closer to the concrete program at hand: this will remove such spurious states that fail the proof despite the fact that they do not exist in the concrete program. Next, we consider a strategy for coping with this problem. In Sect. 12.3.4 we will show how it solves the verification problem for our program.

12.3.3 Loop Invariants

A key tool for program analysis is the **loop invariant**. A loop invariant is any predicate that holds at the beginning of the body, irrespective of how many times the loop iterates.

Consider, for example, the following fragment of C code:

```
int i=0;

while(i != 10) {
      ...
      i++;
}
```

The following predicate is an invariant of this loop, because it is true in the beginning of the loop's body regardless of which iteration we are at:

$$0 \leq i < 10 \,. \tag{12.6}$$

How can we prove that a given predicate is a loop invariant? The answer is that we can use **induction**. Suppose that our program matches the following template:

```
A;
while(C) {
      assert(I);
      B;
}
```

Both code fragments A and B are required to be free of loops, but may contain branching. The condition C and the candidate invariant I must be free of side-effects. We prove that I is an invariant by induction:

1. *Base case*: prove that the loop invariant is satisfied when entering the loop for the first time.
2. *Step case*: prove that, beginning in a state that satisfies the invariant, executing the loop body once brings us to a state that satisfies the invariant as well.

Both parts of the proof must succeed to conclude that the invariant holds. We will now construct two loop-free programs that check the two conditions above. The program we need to check for the base case is simple:

```
A;
assert(C ⟹ I);
```

The program for the induction step first restricts the entry state of the loop body to one that satisfies both the loop condition and the loop invariant. It then executes the body once, and asserts that the invariant still holds.

```
assume($C \land I$);
$B$;
assert($C \Longrightarrow I$);
```

As both programs are loop-free and thus trivially bounded, we can use the techniques described in Sect. 12.2.2 to check whether the assertions in them hold for all inputs. If so, we have established that I is an invariant for the loop.

We will illustrate the process using the small `while` loop given above. To validate that `i >= 0 && i < 10` is indeed an invariant, we first build the base case program:

```
int i=0;
assert(i != 10 $\Longrightarrow$ i >= 0 && i < 10);
```

The assertion in the base case program passes trivially. We now construct the program for the induction step case:

```
assume(i != 10 && i >= 0 && i < 10);
i++;
assert(i != 10 $\Longrightarrow$ i >= 0 && i < 10);
```

The program above can again be checked by means of a suitable decision procedure. We will find in this case that the formula is unsatisfiable (i.e., the assertion passes). Thus, we have established our invariant.

We leave it for the reader to adapt this procedure for `Do-While` and `For` loops (see Problem 12.1).

12.3.4 Refining the Abstraction with Loop Invariants

Let us now improve the precision of the program abstraction procedure explained in Sect. 12.3.1 with loop invariants. The method permits separate loop invariants for each loop. Denote the set of loops by $\mathcal{L}$, and the invariant of loop $\ell \in \mathcal{L}$ by I_ℓ. We now add three further steps to the transformation procedure of Sect. 12.3.1. These steps are performed for each loop $\ell \in \mathcal{L}$.

4. Add an assertion that I_ℓ holds before the nondeterministic assignments to the loop variables. This establishes the base case.
5. Add an assumption that I_ℓ holds after the nondeterministic assignments to the loop variables. This is the induction hypothesis.
6. Add an assertion that $C \Longrightarrow I_\ell$ holds at the end of the loop body. This proves the induction step.

Note that the base case and the step case are checked together with just one program.

It is time to apply this for verifying the Windows driver excerpt from Sect. 12.3.2. Recall that our abstraction suffered from the fact that the assertion `state_of_lock == unlocked` failed when entering the loop. We therefore *guess* that this predicate is a potential invariant of the loop. We then construct a new abstraction using the procedure above. This results in Program 12.3.5.

Program 12.3.5 Program 12.3.4 after refinement using the candidate loop invariant `state_of_lock == unlocked`. The verification fails if this is not indeed an invariant (Lines 3, 25), or if it is not strong enough to prove the property (Lines 12, 18, 29)

```
 1  state_of_lock = unlocked;
 2
 3  assert(state_of_lock == unlocked); // induction base case
 4
 5  state_of_lock = *;
 6  old_count = *;
 7  count = *;
 8  request = *;
 9
10  assume(state_of_lock == unlocked); // induction hypothesis
11
12  assert(state_of_lock == unlocked); // property
13  state_of_lock = locked;
14  old_count = count;
15  request = GetNextRequest();
16  if (request != NULL) {
17    ReleaseRequest(request);
18    assert(state_of_lock == locked); // property
19    state_of_lock = unlocked;
20    ProcessRequest(request);
21    count = count + 1;
22  }
23
24   // induction step case
25  assert(old_count != count ⟹ state_of_lock == unlocked);
26
27  assume(old_count == count);
28
29  assert(state_of_lock == locked); // property
30  state_of_lock = unlocked;
```

The assertions in lines 3, 12, and 18 pass trivially. It is more difficult to see why the two assertions in Lines 25 and 29 pass. Splitting the analysis into two cases can help:

- If `request != NULL`, the program executes the "then" branch of the `if` statement in Line 16. As a result, the variable `state` has the value `unlocked`. Thus, the assertion for the induction step case (Line 25) passes. Furthermore, since `count` is equal to `old_count` plus one, `old_count` is different from `count`, which means that the assertion in Line 29 is not even executed owing to the assumption in Line 27.
- If `request == NULL`, then the execution skips over the "then"-branch. The variable `old_count` is equal to `count` and `state` is equal to `locked`. As a consequence, both assertions pass trivially.

The challenge is to find an invariant that is strong enough to prove the property as in the example above (recall that there the invariant was simply guessed). As an extreme example, the constant "true" is also an invariant, but it is not helpful for verification: it does not restrict the states that are explored by the verification engine. Finding suitable loop invariants is an area of active research. Simple options include choosing predicates that appear in the program text, or constructing new predicates from the program variables and the usual relational operators. A heuristic selects candidates and then uses the decision procedure as described above in an attempt to confirm the invariant and the properties. In general there is no algorithm that always finds an invariant that is strong enough to prove a given property.

12.4 SMT-Based Methods in Biology

Computing has contributed to the biological sciences by making it possible to store and analyze large amounts of experimental data. Recently, the development and application of computational methods and models that capture key biological processes and mechanisms is increasingly utilized towards helping biologists in gaining a clearer understanding of living systems and improving predictive capabilities. The ability to effectively explore and analyze such biological models is key to making scientific progress in the field. The biological modeling applications can be divided into two general areas: 1) engineering biological circuits; and 2) understanding and predicting behavior of natural biological systems. Broadly speaking, engineering of biological circuits is a newer emerging field with a smaller community of interdisciplinary researchers, whereas understanding and prediction in biology has a wider and more established community with potentially thousands of researchers that can benefit from using computational modeling tools.

12.4.1 DNA Computing

A representative domain for the first type of applications is DNA computing, an emerging field in the area of engineering biological circuits. DNA computing research aims to understand the forms of computation that can be performed using biological material. A main goal of DNA computing is to robustly design and build circuits from DNA that can perform specified types of computation. Complex DNA circuits, composed of basic components can now be designed and constructed in the lab [234, 235, 69]. A long term goal is to explore whether biological computers can replace existing computers in certain areas and applications, based on the massively parallel nature of biological computation. Engineered biological circuits are also of significant interest due to their ability to interface with biological material, potentially opening new areas of applications in material design and medicine. Research efforts are now also directed towards integrating engineered circuits within living cells. If these techniques mature, applying formal methods to guarantee 'correctness' of circuit designs will be important for any medical application.

Chemical reaction networks (CRNs) is a convenient formalism to model such circuits. Figure 12.2, for example, shows a CRN of a simple DNA circuit that implements a logical AND gate. We will come back to this figure a little later. The encoding of a CRN as a transition system is quite direct. Assume we have an integer for storing the number of molecules for each species. If a reaction $A + B \rightarrow C$ fires, the number of molecules of `A` and of `B` is decreased by one and the number of molecules of `C` is increased by one, while the number of molecules of all other species remains unchanged. A precondition for firing this reaction is that there is at least one molecule of `A` and at least one molecule of `B`. Hence, such behavior can be modeled with a transition system, and as such it can be analyzed with standard verification techniques in order to prove safety properties or identify concrete bugs in the design.

Formally, a CRN is defined as a pair $(\mathcal{S}, \mathcal{R})$ of species and reactions, where a reaction $r \in \mathcal{R}$ is a pair of multisets $r = (R_r, P_r)$ describing the reactants and products of r with their stoichiometries (the number of participating molecules of a given species, where non-participating species have value 0). We formalize the behavior of a CRN as the transition system $\mathcal{T} = (Q, T)$ where

- Q is a set of multisets of species. For $q \in Q$ we denote by $q(s)$ the number of molecules of species s that are available in a state q.
- $T \subseteq Q \times Q$ is the transition relation defined as

$$T(q, q') \leftrightarrow \bigvee_{r \in \mathcal{R}} \Big(on(r, q) \wedge \bigwedge_{s \in \mathcal{S}} q'(s) = q(s) - R_r(s) + P_r(s) \Big) \, , \tag{12.7}$$

where $on(r, q)$ is TRUE if in state q there are enough molecules of each reactant of r for it to fire, meaning that reaction r is enabled in q.

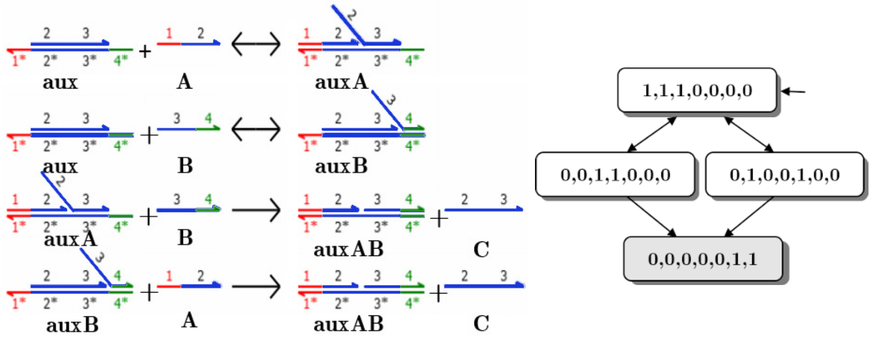


Fig. 12.2. On the left of the figure, we see a DNA circuit implementing a logical AND gate. The inputs are DNA strands of species A, B and the output is a DNA strand of species C. Other species that participate in the computation are aux, auxA, auxB, auxAB (aux is the prefix of "auxiliary"). For a single molecule of species aux, the output C is produced at the end of the computation if and only if both A and B are present, which captures the definition of the AND gate. For each of the seven species, domains labeled by $1, \ldots, 4$ represent different DNA sequences, while complementary sequences are denoted by $*$. The binding of complementary domains and the subsequent displacement of adjacent complementary sequences determines the possible chemical reactions between the species. The reactions are shown in the left side of the figure, using a visual representation introduced in [221]. The notation $\rightharpoonup, \leftharpoondown$ designates that the end of the DNA is on the right or left, respectively. There are in total six chemical reactions in this circuit. The first two pairs are reversible ones: aux + A $\leftrightarrow$ auxA, aux + B $\leftrightarrow$ auxB. The remaining reactions are irreversible: auxA+B $\rightarrow$ auxAB+C and auxB+A $\rightarrow$ auxAB+C. On the right part of the figure, the DNA circuit is represented as a state machine where a state captures the number of molecules from each species. The initial state has one molecule for each of the species (aux,A,B) and 0 molecules for other species and is thus labeled $1, 1, 1, 0, 0, 0, 0$. The computation terminates if it reaches a state in which no additional reactions are possible. In our example the state $0, 0, 0, 0, 0, 1, 1$ is a "sink" state for the system and will eventually be reached.

From (12.7) it is evident that we can encode CRNs with the theory of linear integer arithmetic. Integers are needed due to the potentially unbounded number of molecules. In practice, for many models it is possible to prove upper bounds on the integer representation, thereby allowing the use of bit-vector encodings of appropriate size without sacrificing precision [291].

One approach for implementing CRNs, termed *DNA strand displacement*, is based on the complementarity of DNA sequences. The binding of DNA base pairs (A-T and G-C), provides a mechanism for engineering chemical reaction networks using DNA. In this approach, various single and double-stranded DNA molecules are designated as chemical species. The binding, unbinding and displacement reactions possible between the complementary DNA domains of these species form the desired CRN. Specific computational operations can be implemented using such a strategy, resulting in a system called a *DNA circuit.*

Figure 12.2 shows a simple DNA circuit implementing a logical AND gate (see caption). A state of the system captures the number of available molecules from each DNA species, which change as reactions are fired, leading to the transition system representation. For such a system we would like to prove that the computation stabilizes in a given state and the correct result is computed. Specifically in this case we would like to prove that the result of the AND operation is, eventually, $q(C) = 1$. Formally, this means that we want to check the temporal property

$$(q_0(A) = 1 \wedge q_0(B) = 1) \implies \mathbf{FG}(q(C) = 1) \ , \tag{12.8}$$

where q_0 is the initial state and $\mathbf{F}$ and $\mathbf{G}$ are the *Eventually* and *Always* temporal logic operators, respectively. The transition relation is extended to allow an idling transition in the case where no reaction is enabled. Put in other words, the property is that if in the initial state the number of molecules in `A` and `B` are 1, then eventually we will reach a state from which the number of molecules in `C` is 1 indefinitely. It was demonstrated in [293] that an SMT-based approach makes it possible to effectively analyze complex systems of this type and check such properties. A key towards scaling up the analysis has been identifying invariants of the system that are based on conservation properties, specifically that the number of basic strands in the system does not vary, but basic strands can stick together or detach.

12.4.2 Uncovering Gene Regulatory Networks

An important area for understanding and predicting the behavior of natural biological systems is the study of Gene Regulatory Networks (GRNs) and how they govern the behavior of cells during development [182]. The cell is a fundamental unit of biological systems, its activity and function is controlled by a complex network of interactions specifying patterns of gene expression. A GRN can be viewed as a biological 'program' that activates and represses the activity of genes, determining the functional behavior of the cell. Computational modeling of GRNs has been an active area of research and applications for several decades [89]. The motivation is to synthesize experimental data into useful predictions.

Figure 12.3 shows a simple GRN with genes `A`, `B`, and `C`. Typical regulation relationships between genes are *activation* and *repression*, e.g., `A` activates `B`, or `A` represses itself. In a Boolean network the value of genes is modeled using Boolean variables (A represents `A`, B represents `B`, etc), and their regulation is modeled using predicates. For example in Fig. 12.3 $C' = A \vee B$ denotes that the value of gene `C` is updated to be the logical OR of the values of genes `A` and `B`.

A Boolean network is represented as a pair $(\mathcal{G}, \mathcal{F})$ of genes and update functions. For each gene g in the set of genes $\mathcal{G}$, a state of the system q assigns

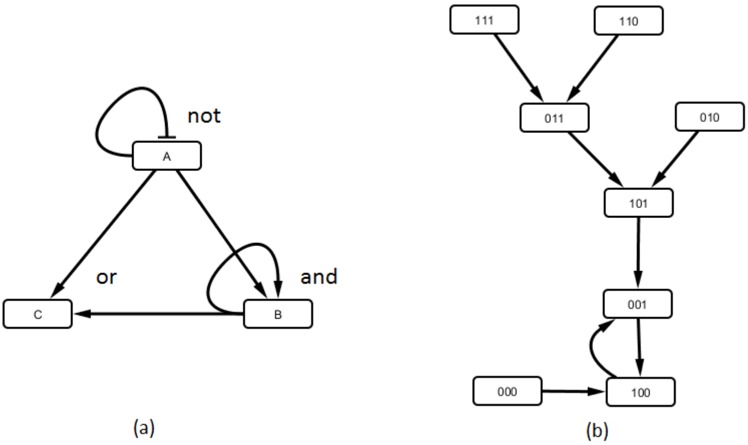


Fig. 12.3. (a) A Boolean network representing three genes A, B, and C. The interactions between genes is represented graphically using pointed arrows (positive interaction) or ⊤-arrows (negative interaction). An update function for each gene should be defined, and specifies the logical combination of inputs that determine the next value of a gene. In this example the next state of genes A, denoted A', is given by $A' = \neg A$. Similarly for genes B and C we have $B' = A \wedge B$ and $C' = A \vee B$, respectively. (b) The Boolean network is represented as a transition system where all nodes are updated synchronously. This system does not stabilize in a single state but instead reaches a cycle where the values of genes A and C oscillate, moving between state 001 to state 100 where each bit in the state name represents the value of genes A,B,C.

a Boolean value depending whether the gene is on or off, thus $q(g) = 1$ if g is on in state q and $q(g) = 0$ if g is off in state q. Each gene is assigned an update function f_g that defines the value of gene g in the next step given the current values of all genes in $\mathcal{G}$.

We formalize the behavior of a Boolean Network as the transition system $\mathcal{T} = (Q, T)$ where

- $Q = \mathbb{B}^{|\mathcal{G}|}$ and $q(g) \in \mathbb{B}$ indicates if gene g is On or Off in state q.
- $T \subseteq Q \times Q$ is the transition relation defined as

$$T(q, q') \leftrightarrow \left(\bigwedge_{g \in \mathcal{G}} q'(g) = f_g(q) \right) . \tag{12.9}$$

This definition assumes *synchronous* semantics, where all genes are updated in the same step, and can be extended to the *asynchronous* case where only a single gene is updated at each step. Understanding the most adequate modeling assumptions in terms of synchronous vs. asynchronous semantics and potential time delays is an area of active research. It appears that in many cases even the ‘simpler’ synchronous semantics is a good approximation. This transition system can naturally be encoded in SMT [291, 292] using the theory of bit vectors and uninterpreted functions (UFBV).

Such networks turn out to be a useful abstraction, despite the simplification of viewing genes as being either on or off, and indeed they are now a well established formalism. A typical analysis question that is studied in such networks is: “Does behavior stabilize in a fixed state of gene expression when starting from a given initial condition?”. In the example in Fig. 12.3 from any initial state the system eventually oscillates between two states where one of the genes `A` and `C` is on and the other is off. Another question of interest is identifying all attractors of a GRN, where an attractor is a set of states such that once it is reached the system remains in them permanently.

SMT solvers are applied to the synthesis of a GRN that satisfies a set of experimental constraints. The motivation is that experimentalists are aiming to uncover the GRN controlling a cell’s behavior by performing a set of measurements, including steady-state gene expression measurements and effects of perturbing certain genes. Reasoning about potential GRNs that can explain known data becomes extremely challenging as the size of the network and the number of experiments grows, thus the application of formal analysis is crucial [216]. A synthesized GRN can be used to predict the dynamics of a cell under new conditions that have not been measured yet in the lab. In particular it is also useful to consider the set of all possible models that explain given data, as this can allow more accurate predictions in a situation where all models agree on certain outcomes [107, 290].

12.5 Problems

12.5.1 Warm-up Exercises

Problem 12.1 (loop invariants). Similarly to the discussion in Sect. 12.3.3, show how loop invariants can be checked, where the loop is given in the form of a `do-while` template, and when it is given in the form of a `for` template, both according to the semantics of the C programming language.

12.5.2 Bounded Symbolic Simulation

Problem 12.2 (static single assignment representation). For the program that appears below:

1. Show the SSA form corresponding to an unfolding of the program, and the corresponding formula. The `for` loop has to be unfolded three times, and the `Next` function should be inlined.
2. Now add an assertion after the exit from the `for` loop, that $y > x$, and rewrite the formula so it is satisfiable if and only if this assertion always holds. Do you expect the assertion to always hold?

```
int main(int x, int y)
{
  int result;
  if (x < y)
     x = x + y;
  for (int i = 0; i < 3; ++i) {
      y = x + Next(y);
  }
  result = x + y;
  return result;
}

int Next(int x) {
  return x + 1;
}
```

Problem 12.3 (SSA with arrays). Give an example of a program that writes into an array using an index which is a variable. Give its SSA form, and the SSA formula that is obtained from it.

Problem 12.4 (SSA with pointers). We have not explained how to construct SSA for programs that dereference pointers. Now consider the following program:

```
int i;

void my_function(int *p)
{
  int j = 0, *q = &j;

  j += *p + *q;
}
```

1. Assume that the program only contains variables of type `int` and `int *`, and that dereferenced pointers are only read. Explain how to build SSA with this restriction. Apply your method to the program above.
2. Repeat the first item, but now allow pointers to pointers, and pointers to pointers to pointers, and so on. Give an example.
3. Repeat the first item, but now allow assigning to dereferenced pointers, as a means for writing to integer variables, e.g., `*i = 5;`.

12.5.3 Overapproximating Programs

Problem 12.5 (pointers and overapproximation). Explain how to overapproximate programs that use pointers. Demonstrate your approach using the following program. Use a loop invariant that is suitable for proving the assertion in the program.

```
void my_function(int *p) {

  for(*p=0; *p < 10; (*p)++) {
     ...
  }

  assert(*p == 10);
}
```

Problem 12.6 (loop invariants). Consider the following program:

```
char mybuf[256];

void concatenate(
  char buf1[], unsigned len1,
  char buf2[], unsigned len2)
{
  if(len1+len2 > 256)
    return;

  for(unsigned i = 0; i != len1; i++)
    mybuf[i] = buf1[i];

  for(unsigned i = 0; i != len2; i++)
    mybuf[len1 + i] = buf2[i];
}
```

1. Annotate the program with assertions that check that the array indices are within the bounds. You can assume that the size of `buf1` is `len1` and that the size of `buf2` is `len2`.
2. Now assume that the type `unsigned` is the set of unbounded natural numbers. Under this assumption, prove the assertions you have written by means of suitable invariants for both loops.
3. In ANSI-C programs, the type `unsigned` is in fact a bit vector type, typically with 32 bits. Thus recall the semantics of unsigned bit-vector arithmetic as introduced in Chap. 6. Do your assertions still hold? Justify your answer.

12.6 Bibliographic Notes

Formal program verification is a major research area dating back to at least the 1950s, and we have neither intention nor hope to be able to describe its milestones or main contributors here. We will only mention some references related directly to the material covered in this chapter, namely path-based symbolic simulation, Bounded Model Checking, and proofs with invariants.

Path-Based Symbolic Simulation

The use of decision procedures and path-based symbolic simulation to generate test cases for software was already reported in 1974 [165], and since then, hundreds of publications on this subject have appeared. We will therefore focus on a few well-known approaches. How is the execution path chosen? In static symbolic simulation, the execution path can be chosen by extending a partial path with new conjuncts as long as the resulting formula is satisfiable. As soon as the formula becomes unsatisfiable, some other path can be explored by negating one of the branch conditions encountered on the path. By contrast, in dynamic symbolic simulation, the choice of path is driven by a concrete execution.

The JAVA PATH FINDER (JPF) [280, 163], originally an explicit-state model checker for multi-threaded Java Bytecode, now features a hybrid state representation that includes path constraints. JPF instruments the code such that it builds a symbolic formula when executed. JPF tries to avoid exploring invalid paths by calling the decision procedure to check whether there exists an execution that follows the path. DART [131] integrates path-based symbolic execution into a random test generator. It replaces symbolic values by explicit values in case the solver is unable to handle the symbolic constraint. Concurrency is not supported in the DART implementation, but it is supported in its successor, CUTE [255].

Symbolic execution has recently found significant industrial adoption in the context of security analysis. The SAGE [130] tool, which is based on path-based symbolic execution, is used to find most of the bugs identified by Microsoft's fuzz-testing infrastructure. It has been used on hundreds of parsers for various media formats and is administrated in a data-center test environment. The KLEE tool [62], similarly, has been instrumental in finding a large number of security vulnerabilities in code deployed on Windows and Linux. Finally, the PEX [130] tool offers an integration of symbolic execution with the .NET runtime. It can thus be used on any .NET language, including C#. PEX lets programmers directly take advantage of the symbolic execution technology for generating test inputs to .NET code. It offers a sophisticated integration with the .NET type system that enables it to generate test cases for complex, structured data. PREFIX is used to analyze millions of lines of Microsoft source code on a routine basis. The COVERITY [111] analyzer contains analogous techniques to PREFIX, including bit-precise analysis, as does GrammaTech's CODESONAR tool. A common trait of these tools is that they do not aim to give strong guarantees about absence of runtime errors. They are bug-hunting tools.

Symbolic execution remains of interest also for the security community including "white", "blue", and "black" "hats" (jargon for industrial, legitimate, and hackers with a shady purpose). Microsoft's PREFIX [61] analysis tool pioneered a bottom-up analysis of procedures: it summarizes basic (bounded) procedures as a set of guarded transitions, and then uses these guarded transitions when analyzing procedures that call them. The set of guarded transitions correspond to an execution path, and include automatic assertions (e.g., no null dereferences). An SMT solver is used to check whether there are inputs that drive an execution along the path and violate the assertion.

Bounded Analyzers

Bounded Model Checking (BMC) [34] was originally developed for verifying logic designs of hardware circuits. It is based on unrolling the design k times, for a given bound k, and encoding the resulting circuit as a propositional formula. A constraint representing the negation of the property is added, and hence the resulting formula is satisfiable if and only if an error trace of length k or less exists. The adaptation of this idea to software is what is described in Sect. 12.2. In 2000, Currie et al. proposed this approach in a tool that unwinds loops in assembly programs running on DSPs [83]. The approach was extended to ANSI-C programs in 2003 with an application to hardware/software co-verification by the CBMC software model checker [170]. CBMC uses a combination of the array theory with the bit-vector theory (pointers are represented as pairs of bit vectors, an index of an object, and an offset within it). It can export the verification condition to external SAT and SMT solvers. An extension to bounded verification of concurrent software based on an explicit enumeration of thread interleavings was presented in 2005 [237]. Given a bound on the number of context switches, thread interleavings can also be encoded into sequential code [179, 154]. Alternatively, they can be encoded symbolically, e.g., using constraints given as partial orders [3]. F-SOFT [155] uses BDD and SAT-based engines to verify bit-accurate models of C programs. Instances of this approach also include ESBMC [80] and LLBMC [195].

Abstraction-Based Methods

Predicate abstraction was introduced by Susanne Graf and Hassen Saïdi [135] in the context of interactive theorem proving. It was later used by Microsoft's SLAM [12], the tool BLAST [139], and to a limited degree also by JAVA PATH FINDER [280], all of which can be considered the pioneers of software verifiers that support industrial programming languages. SLAM has evolved into SDV (for *Static Driver Verifier*) [13], which is now part of Microsoft's Windows Driver Kit. SDV is used to verify device drivers, and is considered to be the first wide-scale industrial application of formal software verification. In the decade that followed, at least ten further software verifiers based on predicate abstraction were introduced. Both MAGIC [65] and SATABS [73] can also verify concurrent programs. The BFC Model Checker for Boolean programs can verify the *parametric case*, i.e., programs in which the number of concurrent threads is not bounded apriori [161]. With the Windows 8.1 release, SDV now uses Corral as verifier [177, 178], which uses techniques described in Sect. 12.3.

The annual competition for propositional SAT solvers has resulted in a remarkable surge in the performance and quality of the solvers. To this end, the Competition on Software Verification (SV-COMP) was founded in 2012, and is held annually in association with the conference TACAS. The benchmarks are split into numerous categories according to particular language features that are exercised, including "Bit-Vectors", "Concurrency", and "HeapManipulation".

Beyond Safety Checking

This chapter focuses on methods for checking reachability properties. However, decision procedures have applications beyond reachability checking in program analysis, and we will give a few examples. *Termination checkers* attempt to answer the question "does this program run forever, or will it eventually terminate?" Proving program termination is typically done by finding a ranking function for the program states, and the generation of these ranking functions relies on a decision procedure for the appropriate theory. Typical instances use linear arithmetic over the rationals (e.g., [76]) and the bit vectors [74, 86, 68]. A further applications of program analysis that goes beyond reachability checking is the computation of quantitative properties of programs, e.g., information leakage [141].

A

SMT-LIB: a Brief Tutorial

A.1 The Satisfiability-Modulo-Theory Library and Standard (SMT-LIB)

A bit of history: The growing interest and need for decision procedures such as those described in this book led to the **SMT-LIB initiative** (short for Satisfiability-Modulo-Theory Library). The main purpose of this initiative was to streamline the research and tool development in the field to which this book is dedicated. For this purpose, the organizers developed the **SMT-LIB standard** [239], which formally specifies the theories that attract enough interest in the research community, and that have a sufficiently large set of publicly available benchmarks. As a second step, the organizers started collecting benchmarks in this format, and today (2016) the SMT-LIB repository includes more than 100 000 benchmarks in the SMT-LIB 2.5 format, classified into dozens of logics. A third step was to initiate **SMT-COMP**, an annual competition for SMT solvers, with a separate track for each division.

These three steps have promoted the field dramatically: only a few years back, it was very hard to get benchmarks, every tool had its own language standard and hence the benchmarks could not be migrated without translation, and there was no good way to compare tools and methods.[1] These problems have mostly been solved because of the above initiative, and, consequently, the number of tools and research papers dedicated to this field is now steadily growing.

The SMT-LIB initiative was born at FroCoS 2002, the fourth Workshop on Frontiers of Combining Systems, after a proposal by Alessandro Armando. At the time of writing this appendix, it is co-led by Clark Barrett, Pascal Fontaine, and Cesare Tinelli. Clark Barrett, Leonardo de Moura, and Cesare Tinelli currently manage the SMT-LIB benchmark repository.

[1] In fact, it was reported in [94] that each tool tended to be the best on its own set of benchmarks.

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0

The current state: The current SMT-LIB standard is at version 2.5 (as of 2015). It supports the theories that are presented in Fig. A.1. The symbols should be interpreted as follows:

- QF for the restriction to quantifier-free formulas
- A or AX for arrays without or with extensionality
- BV for fixed-size bit-vectors
- FP for Floating-Point
- IA for integer arithmetic
- RA for real arithmetic
- IRA for mixed integer arithmetic
- IDL for integer difference logic
- RDL for rational difference logic
- L before IA, RA, or IRA for the linear fragment of those arithmetics
- N before IA, RA, or IRA for the nonlinear fragment of those arithmetics
- UF for the extension allowing free sort and function symbols

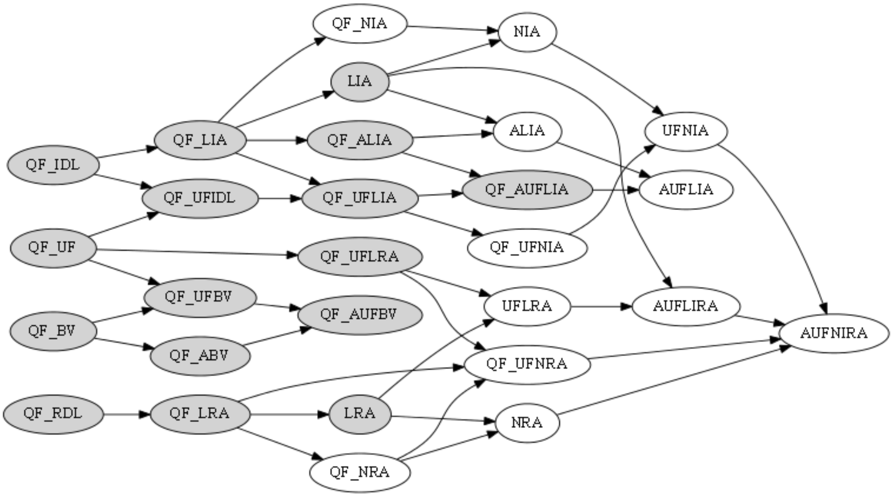


Fig. A.1. The theories supported by the SMT-LIB standard have associated benchmarks and at least one tool that (attempts to) solve them. An arrow (T_1, T_2) means that T_1 is a special case of T_2. The greyed nodes are theories that are covered in this book. The figure is copied (with permission) from the SMT-LIB web-site

A.2 The SMT-LIB File Interface

The SMT-LIB standard defines a file format for describing decision problems. The benefit of a standardized file format is that it is easy to experiment with a range of solvers, and to replace the solver used in case better solvers are developed. The description below refers to ver. 2.0 of the standard, but ver. 2.5 is backward-compatible.

SMT-LIB files are ASCII text files, and as a consequence can be written with any text editor that can save plain text files. The syntax is derived from that of Common Lisp's S-expressions. All popular solvers are able to read formulas from files or the standard input of the program, which permits the use of POSIX pipes to communicate with the solver. We will refrain from giving a formal syntax and semantics for SMT-LIB files, and will instead give examples for the most important theories.

A.2.1 Propositional Logic

We will begin with an example in propositional logic. Suppose we wanted to check the satisfiability of

$$(a \lor b) \land \neg a \ .$$

We first need to *declare* the Boolean variables a and b. The SMT-LIB syntax offers the command `declare-fun` for declaring functions, i.e., mappings from some sequence of function arguments to the domain of the function. Variables are obtained by creating a function without arguments. Thus, we will write

```
(declare-fun a () Bool)
(declare-fun b () Bool)
```

to obtain two Boolean variables named a and b. Note the empty sequence of arguments after the name of the variable.

We can now write constraints over these variables. The syntax for the usual Boolean constants and connectives is as follows:

TRUE	`true`
FALSE	`false`
$\neg a$	`(not a)`
$a \implies b$	`(=> a b)`
$a \land b$	`(and a b)`
$a \lor b$	`(or a b)`
$a \oplus b$	`(xor a b)`

Using the operators in the table, we can write the formula above as follows:

```
(and (or a b) (not a))
```

Constraints are given to the SMT solver using the command `assert`. We can add the formula above as a constraint by writing

```
(assert (and (or a b) (not a)))
```

As our formula is a conjunction of two constraints, we could have equivalently written

```
(assert (or a b))
(assert (not a))
```

After we have passed all constraints to the solver, we can check satisfiability of the constraint system by issuing the following command:

```
(check-sat)
```

The solver will reply to this command with unsat or sat, respectively. In the case of the formula above, we will get the answer sat. To inspect the satisfying assignment, we issue the get-value command.

```
(get-value (a b))
```

This command takes a list of variables as argument. This makes it possible to query the satisfying assignment for any subset of the variables that have been declared.

A.2.2 Arithmetic

The SMT-LIB format standardizes syntax for arithmetic over integers and over reals. The type of the variable is also called the *sort*. The SMT-LIB syntax has a few constructs that can be used for all sorts. For instance, we can write (= x y) to denote equality of x and y, provided that x and y have the same sort. Similarly, we can write (disequal x y) to say that x and y are different. The operator disequal can be applied to more than two operands, e.g., as in (disequal a b c). This is equivalent to saying that all the arguments are different.

The SMT-LIB syntax furthermore offers a trinary if-then-else operator, which is denoted as (ite c x y). The first operand must be a Boolean expression, whereas the second and third operands may have any sort as long as the sort of x matches that of y. The expression evaluates to x if c evaluates to TRUE, and to y otherwise.

To write arithmetic expressions, SMT-LIB offers predefined sorts called Real and Int. The obvious function symbols are defined, as given in the table below.

addition	+
subtraction	−
unary minus	−
multiplication	*
division	/ (reals) div (integers)
remainder	mod (integers only)
relations	< > <= >=

Many of the operators can be *chained*, with the obvious meaning, as in, for example, (+ x y z). The solver will check that the variables in an expression have the same sort. The nonnegative integer and decimal constant symbols 1, 2, 3.14, and so on are written in the obvious way. Thus, the expression $2x + y$ is written as (+ (* 2 x) y). To obtain a negative number, one uses the unary minus operator, as in (- 1). By contrast, -1 is not accepted.

A.2.3 Bit-Vector Arithmetic

The SMT-LIB syntax offers a *parametric* sort BitVec, where the parameter indicates the number of bits in the bit vector. The underscore symbol is used to indicate that BitVec is parametric. As an example, we can define an 8-bit bit vector a and a 16-bit bit vector b as follows:

```
(declare-fun a () (_ BitVec 8))
(declare-fun b () (_ BitVec 16))
```

Constants can be given in hexadecimal or binary notation, e.g., as follows:

```
(assert (= a #b11110000))
(assert (= b #xff00))
```

The operators are given in the table below. Recall from Chap. 6 that the semantics of some of the arithmetic operators depend on whether the bit vector is interpreted as an unsigned integer or as two's complement. In particular, the semantics differs for the division and remainder operators, and the relational operators.

	Unsigned	Two's complement
addition	bvadd	
subtraction	bvsub	
multiplication	bvmul	
division	bvudiv	bvsdiv
remainder	bvurem	bvsrem
relations	bvult, bvugt, bvule, bvuge	bvslt, bvsgt, bvsle, bvsge
left shift	bvshl	
right shift	bvlshr	bvashr

Bit vector concatenation is done with concat. A subrange of the bits of a bit vector can be extracted with (_ extract i j), which extracts the bits from index j to index i (inclusive).

A.2.4 Arrays

Recall from Chap. 7 that arrays map an index type to an element type. As an example, we would write

```
(declare-fun a () (Array Int Real))
```

in SMT-LIB syntax to obtain an array a that maps integers to reals. The SMT-LIB syntax for $a[i]$ is (select a i), and the syntax for the array update operator $a\{i \leftarrow e\}$ is (store a i e).

A.2.5 Equalities and Uninterpreted Functions

Equality logic can express equalities and disequalities over variables taken from some unspecified set. The only assumption is that this set has an infinite number of elements. To define the variables, we first need to declare the set itself, i.e., in SMT-LIB terminology, we declare a new sort. The command is `declare-sort`. We obtain a new sort `my_sort` and variables a, b, and c of that sort as follows:

```
(declare-sort my_sort 0)
(declare-fun a () my_sort)
(declare-fun b () my_sort)
(declare-fun c () my_sort)
(assert (= a b))
(assert (disequal a b c))
```

The number zero in the `declare-sort` command is the arity of the sort. The arity can be used for subtyping, e.g., the arrays from above have arity two.

B

A C++ Library for Developing Decision Procedures

B.1 Introduction

A decision procedure is always more than one algorithm. A lot of infrastructure is required to implement even simple decision procedures. We provide a large part of this infrastructure in the form of the DPLIB library in order to simplify the development of new procedures. DPLIB is available for download,[1] and consists of the following parts:

- A template class for a basic data structure for graphs, described in Sect. B.2.
- A parser for a simple fragment of first-order logic given in Sect. B.3.
- Code for generating propositional SAT instances in CNF format, shown in Sect. B.4.
- A template for a decision procedure that performs a lazy encoding, described in Sect. B.5.

To begin with, the decision problem (the formula) has to be read as input by the procedure. The way this is done depends on how the decision procedure interfaces with the program that generates the decision problem.

In industrial practice, many decision procedures are embedded into larger programs in the form of a subprocedure. Programs that use a decision procedure are called *applications*. If the run time of the decision procedure dominates the total run time of the application, solvers for decision problems are often interfaced to by means of a *file interface*. This chapter provides the basic ingredients for building a decision procedure that uses a file interface. We focus on the C/C++ programming language, as all of the best-performing decision procedures are written in this language.

The components of a decision procedure with a file interface are shown in Fig. B.1. The first step is to *parse* the input file. This means that a sequence of characters is transformed into a *parse tree*. The parse tree is subsequently

[1] http://www.decision-procedures.org/

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0

checked for type errors (e.g., adding a Boolean to a real number can be considered a type error). This step is called *type checking*. The module of the program that performs the parsing and type-checking phases is usually called the *front end*.

Most of the decision procedures described in this book permit an arbitrary Boolean structure in the formula, and thus have to reason about propositional logic. The best method to do so is to use a modern SAT solver. We explain how to interface to SAT solvers in Sect. B.4. A simple template for a decision procedure that implements an incremental translation to propositional logic, as described in Chap. 3, is given in Sect. B.5.

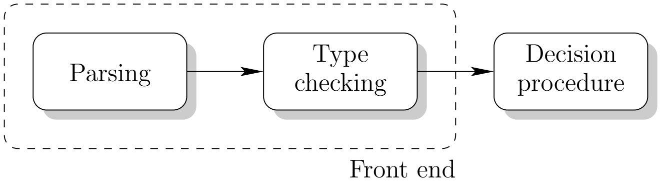


Fig. B.1. Components of a decision procedure that implements a file interface

B.2 Graphs and Trees

Graphs are a basic data structure used by many decision procedures, and can serve as a generalization of many more data structures. As an example, trees and directed acyclic graphs are obvious special cases of graphs. We have provided a template class that implements a generic graph container.

This class has the following design goals:

- It provides a *numbering* of the nodes. Accessing a node by its number is an $O(1)$ operation. The node numbers are *stable*, i.e., stay the same even if the graph is changed or copied.
- The data structure is optimized for *sparse* graphs, i.e., with few edges. Inserting or removing edges is an $O(\log k)$ operation, where k is the number of edges. Similarly, determining if a particular edge exists is also $O(\log k)$.
- The nodes are stored densely in a `vector`, i.e., with very little overhead per node. This permits a large number (millions) of nodes. However, adding or removing nodes may invalidate references to already existing nodes.

An instance of a graph named `G` is created as follows:

```
#include "graph.h"
...
  graph<graph_nodet<> > G;
```

Initially, the graph is empty. Nodes can be added in two ways: a single node is added using the method add_node(). This method adds one node, and returns the number of this node. If a larger number of nodes is to be added, the method resize(i) can be used. This changes the number of nodes to i by either adding or removing an appropriate number of nodes. Means to erase individual nodes are not provided.

The class graph can be used for both directed and undirected graphs. Undirected graphs are simply stored as directed graphs where edges always exist in both directions. We write $a \longrightarrow b$ for a directed edge from a to b, and $a \longleftrightarrow b$ for an undirected edge between a and b.

Class:	graph<T>	
Methods:	add_edge(a, b)	adds $a \longrightarrow b$
	remove_edge(a, b)	removes $a \longrightarrow b$, if it exists
	add_undirected_edge(a, b)	adds $a \longleftrightarrow b$
	remove_undirected_edge(a, b)	removes $a \longleftrightarrow b$
	remove_in_edges(a)	removes $x \longrightarrow a$, for any node x
	remove_out_edges(a)	removes $a \longrightarrow x$, for any node x
	remove_edges(a)	removes $a \longrightarrow x$ and $x \longrightarrow a$, for any node x

Table B.1. Interface of the template class graph<T>

The methods of this template class are shown in Table B.1. The method has_edge(a, b) returns true if and only if $a \longrightarrow b$ is in the graph. The set of nodes x such that $x \longrightarrow a$ is returned by in(a), and the set of nodes x such that $a \longrightarrow x$ is returned by out(a).

The class graph provides an implementation of the following two algorithms:

- The set of nodes that are reachable from a given node a can be computed using the method visit_reachable(a). This method sets the member .visited of all nodes that are reachable from node a to true. This member can be set for all nodes to false by calling the method clear_visited().
- The shortest path from a given node a to a node b can be computed with the method shortest_path(a, b, p), which takes an object p of type graph::patht (a list of node numbers) as its third argument, and stores the shortest path between a and b in there. If b is not reachable from a, then p is empty.

B.2.1 Adding "Payload"

Many algorithms that operate on graphs may need to store additional information per node or per edge. The container class provides a convenient way to do so by defining a new class for this data, and using this new class as a template argument for the template `graph`. As an example, this can be used to define a graph that has an additional string member in each node:

```
#include "graph.h"

class my_nodet {
public:
  std::string name;
};
...

  graph<my_nodet> G;
```

Data members can be added to the edges by passing a class type as a second template argument to the template `graph_nodet`. As an example, the following fragment allows a `weight` to be associated with each edge:

```
#include "graph.h"

class my_edget {
  int weight;

  my_edget():weight(0) {
  }
};

class my_nodet {
};
...

  graph<my_nodet, my_edget> G;
```

Individual edges can be accessed using the method `edge()`. The following example sets the weight of edge $a \longrightarrow b$ to 10:

```
  G.edge(a, b).weight=10;
```

B.3 Parsing

B.3.1 A Grammar for First-Order Logic

Many decision problems are stored in a file. The decision procedure is then passed the name of the file. The first step of the program that implements

id : `[a-zA-Z_$][a-zA-Z0-9_$]`$^+$
$\mathbb{N}$-*elem* : `[0-9]`$^+$
$\mathbb{Q}$-*elem* : `[0-9]`*`.[0-9]`$^+$
infix-function-id : $+ \mid - \mid * \mid / \mid$ *mod*
boolop-id : $\wedge \mid \vee \mid \Leftrightarrow \mid \Longrightarrow$
infix-relop-id : $< \mid > \mid \leq \mid \geq \mid =$
quantifier : $\forall \mid \exists$
term : *id*
| $\mathbb{N}$-*elem* | $\mathbb{Q}$-*elem*
| *id* `(` *term-list* `)`
| *term infix-function-id term*
| $-$ *term*
| `(` *term* `)`
formula : *id*
| *id* `(` *term-list* `)`
| *term infix-relop-id term*
| *quantifier variable-list* **:** *formula*
| `(` *formula* `)`
| *formula boolop-id formula*
| $\neg$ *formula*
| `true` | `false`

Fig. B.2. Simple BNF grammar for formulas

the decision procedure is therefore to parse the file. The file is assumed to follow a particular syntax. We have provided a parser for a simple fragment of first-order logic with quantifiers.

Figure B.2 shows a grammar of this fragment of first-order logic. The grammar in Fig. B.2 uses mathematical notation. The corresponding ASCII representations are listed in Table B.2.

All predicates, variables, and functions have *identifiers*. These identifiers must be declared before they are used. Declarations of variables come with a *type*. These types allow a problem that is in, for example, linear arithmetic over the integers to be distinguished from a problem in linear arithmetic over the reals. Figure B.3 lists the types that are predefined. The *domain* $\mathbb{U}$ is used for types that do not fit into the other categories.

$\mathbb{B}$	`boolean`
$\mathbb{N}_0$	`natural`
$\mathbb{Z}$	`int`
$\mathbb{R}$	`real`
$\mathbb{B}^N$	`unsigned [N]`
$\mathbb{B}^N$	`signed [N]`
$\mathbb{U}$	`untyped`

Fig. B.3. Supported types and their ASCII representations

Mathematical symbol	Operation	ASCII
$\neg$	Negation	not, !
$\wedge$	Conjunction	and, &
$\vee$	Disjunction	or, |
$\Leftrightarrow$ $\Longrightarrow$	Biimplication Implication	<=> =>
$<$ $>$ $\leq$ $\geq$ $=$	Less than Greater than Less than or equal to Greater than or equal to Equality	< > <= >= =
$\forall$ $\exists$	Universal quantification Existential quantification	forall exists
$-$	Unary minus	-
$\cdot$ $/$ mod	Multiplication Division Modulo (remainder)	* / mod
$+$ $-$	Addition Subtraction	+ -

Table B.2. Built-in function symbols

Table B.2 also defines the precedence of the built-in operators: the operators with higher precedence are listed first, and the precedence levels are separated by horizontal lines. All operators are left-associative.

B.3.2 The Problem File Format

The input files for the parser consist of a sequence of *declarations* (Fig. B.4 shows an example). All variables, functions, and predicates are declared. The declarations are separated by semicolons, and the elements in each declaration are separated by commas. Each variable declaration is followed by a type (as listed in Fig. B.3), which specifies the type of all variables in that declaration.

A declaration may also define a formula. Formulas are *named* and *tagged.* Each entry starts with the name of the formula, followed by a colon and one of the keywords `theorem`, `axiom`, or `formula`. The keyword is followed by a formula. Note that the formulas are not necessarily *closed*: the formula `simplex` contains the unquantified variables `i` and `j`. Variables that are not quantified explicitly are implicitly quantified with a universal quantifier.

```
a, b, x, p, n: int;
el: natural;
pi: real;
i, j: real;
u: untyped;        -- an untyped variable
abs: function;
prime, divides: predicate;

absolute: axiom    forall a: ((a >=0 ==> abs(a) = a) and
                    (a < 0 ==> abs(a) = -a)) ==>
                    (exists el: el = abs(a));
divides: axiom     (forall a, b: divides (a, b) <=>
                    exists x: b = a * x);
simplex: formula   (i + 5*j <= 3) and
                   (3*i < 3.7) and
                   (i > -1) and (j > 0.12)
```

Fig. B.4. A realistic example

B.3.3 A Class for Storing Identifiers

Decision problems often contain a large set of variables, which are represented by identifier strings. The main operation on these identifiers is comparison. We therefore provide a specialized string class that features string comparison in time $O(1)$. This is implemented by storing all identifiers inside a hash table. Comparing strings then reduces to comparing indices for that table.

Identifiers are stored in objects of type `dstring`. This class offers most of the methods that the other string container classes feature, with the exception of any method that modifies the string. Instances of type `dstring` can be copied, compared, ordered, and destroyed in time $O(1)$, and use as much space as an integer variable.

B.3.4 The Parse Tree

The parse tree is stored in a graph class `ast::astt` and is generated from a file as follows (Fig. B.5):

1. Create an instance of the class `ast::astt`.
2. Call the method `parse(file)` with the name of the file as an argument. The method returns `true` if an error was encountered during parsing.

The class `ast::astt` is a specialized form of a graph, and stores nodes of type `ast::nodet`. The root node is returned by the method `root()` of the class `ast::astt`. Each node stores the following information:

```
#include "parsing/ast.h"

...

  ast::astt ast;

  if(ast.parse(argv[1])) {
    std::cerr << "parsing␣failed" << std::endl;
    exit(1);
  }
```

Fig. B.5. Generating a parse tree

1. Each node has a numeric label (an integer). This is used to distinguish the operators and the terminal symbols. Table B.3 contains a list of the symbolic constants that are used for the numeric labels.
2. Nodes that contain identifiers or a numeric constant also have a string label, which is of type `dstring` (see Sect. B.3.3). We use strings for the numeric constants instead of the numeric types offered by C++ in order to support unbounded numbers.
3. Each node may have up to two child nodes.

As described in Sect. B.2, the nodes of the graph are numbered. In fact, the `ast::nodet` class is only a wrapper around these numbers, and thus can be copied efficiently. The methods it offers are shown in Table B.4. The methods `c1()` and `c2()` return NIL if there is no first or second child node, respectively.

For convenience, the `ast::astt` class provides a *symbol table*, which is a mapping from the set of identifiers to their types. Given an identifier s, the method `get_type_node(`s`)` returns the node in the parse tree that corresponds to the type of s.

B.4 CNF and SAT

B.4.1 Generating CNF

The library provides algorithms for converting propositional logic into CNF using Tseitin's method (see Sect. 1.3). The resulting clauses can be passed directly to a propositional SAT solver. Alternatively, they can be written to disk in the DIMACS format. The interface to both back ends is defined in the `propt` base class. This class is used wherever the specific propositional back end is to be left unspecified. Literals (i.e., variables or their negations) are

Name	Used for
N_IDENTIFIER	Identifier
N_INTEGER	Integer constant
N_RATIONAL	Rational constant
N_INT	Integer type
N_REAL	Real type
N_BOOLEAN	Boolean type
N_UNSIGNED	Unsigned type
N_SIGNED	Signed type
N_AXIOM	Axiom
N_DECLARATION	Declaration
N_THEOREM	Theorem
N_CONJUNCTION	$\wedge$
N_DISJUNCTION	$\vee$
N_NEGATION	$\neg$
N_BIIMPLICATION	$\iff$
N_IMPLICATION	$\implies$
N_TRUE	True
N_FALSE	False
N_ADDITION	$+$
N_SUBTRACTION	$-$
N_MULTIPLICATION	$*$
N_DIVISION	$/$
N_MODULO	mod
N_UMINUS	Unary minus
N_LOWER	$<$
N_GREATER	$>$
N_LOWEREQUAL	$\leq$
N_GREATEREQUAL	$\geq$
N_EQUAL	$=$
N_FORALL	$\forall$
N_EXISTS	$\exists$
N_LIST	A list of nodes
N_PREDICATE	Predicate
N_FUNCTION	Function

Table B.3. Numeric labels of nodes and their meanings

Class:	ast::nodet	
Methods:	id()	Returns the numeric label
	string()	Returns the string label
	c1()	Returns the first child node
	c2()	Returns the second child node
	number()	Returns the number of the node
	is_nil()	Returns true if the node is NIL

Table B.4. Interface of the class ast::nodet

stored in objects of type literalt. The constants TRUE and FALSE are returned by const_literal(true) and const_literal(false), respectively.

Class:	propt	
Methods:	land(a, b)	Returns a literal l with $l \iff a \wedge b$
	land(v)	Given a vector $v = \langle v_1, \ldots, v_n \rangle$, returns a literal l with $l \iff \bigwedge_i v_i$
	lor(a, b)	Returns a literal l with $l \iff a \vee b$
	lor(v)	Given a vector $v = \langle v_1, \ldots, v_n \rangle$, returns a literal l with $l \iff \bigvee_i v_i$
	lxor(a, b)	Returns a literal l with $l \iff a \oplus b$
	lnot(a, b)	Returns a literal l with $l \iff \neg a$
	lnand(a, b)	Returns a literal l with $l \iff \neg(a \wedge b)$
	lnor(a, b)	Returns a literal l with $l \iff \neg(a \vee b)$
	lequal(a, b)	Returns a literal l with $l \iff (a \iff b)$
	limplies(a, b)	Returns a literal l with $l \iff (a \implies b)$
	lselect(a, b, c)	Returns a literal l with $(a \implies (l \iff b)) \wedge (\neg a \implies (l \iff c))$
	set_equal(a, b)	Adds the constraint $a \iff b$
	new_variable()	Returns a new variable
	const_literal(c)	Returns a literal with a constant Boolean truth value given by c

Table B.5. Interface of the class propt

The interface of the class propt is specified in Table B.5. The classes satcheckt and dimacs_cnft are derived from this class. An implementation of a state-of-the-art propositional SAT solver is given by the class satcheckt. The additional methods it provides are shown in Table B.6. The class dimacs_cnft is used to store the clauses and dump them into a text file that uses the DIMACS CNF format. Its interface is given in Table B.7.

Class:	`satcheckt`, derived from `propt`	
Methods:	`prop_solve()`	Returns `P_SATISFIABLE` if the formula is SAT
	`l_get(`l`)`	Returns the value of l in the satisfying assignment
	`solver_text()`	Returns a string that identifies the solver

Table B.6. Interface of the class `satcheckt`

Class:	`dimacs_cnft`, derived from `propt`	
Methods:	`write_dimacs_cnf(s)`	Dumps the formula in DIMACS CNF format into the stream s

Table B.7. Interface of the class `dimacs_cnft`

B.4.2 Converting the Propositional Skeleton

The **propositional skeleton** (see Chap. 3) of a parse tree can be generated using the class `skeletont`. This offers an operator `()`, which can be applied as follows, where `root_node` is the root node of a formula, and `prop` is an instance of `propt`:

```
#include "sat/skeleton.h"

...

  skeletont skeleton;

  skeleton(root_node, prop);
```

Besides converting the propositional part, the method also generates a vector `skeleton.nodes`, where each element corresponds to a node in the parse tree. Each node has two attributes:

- The attribute `type` is one of `PROPOSITIONAL` or `THEORY`, and distinguishes the skeleton from the theory atoms.
- In the case of a skeleton node, the attribute `l` is the literal that encodes the node.

B.5 A Template for a Lazy Decision Procedure

The library provides two templates for decision procedures that compute a propositional encoding of a given formula φ in the lazy manner. These algorithms are described in detail under the names Lazy-DPLL (Algorithm 3.3.2) and DPLL(T) (Algorithm 3.4.1) in Chap. 3.

We first define a common interface for any kind of decision procedure. This interface is defined by a class decision_proceduret (Table B.8). This class offers a method is_satisfiable(φ), which returns TRUE if and only if the formula φ is satisfiable. If so, one may call the methods print_assignment(s) and get(n). The method print_assignment(s) dumps the entire satisfying assignment into a stream, whereas get(n) permits querying the value of an individual node n of φ.

Class:	decision_proceduret	
Methods:	is_satisfiable(φ)	Returns TRUE if the formula φ is found to be SAT
	print_assignment(s)	Dumps the satisfying assignment into the stream s
	get(n)	Returns the value assigned to node n of φ

Table B.8. Interface of the class decision_proceduret

Class:	lazy_dpllt, derived from decision_proceduret	
Methods:	assignment(n, v)	This method is called by the SAT solver for every assignment to a Σ-literal in φ. The node it corresponds to is n; the value assigned is given by v.
	deduce()	This method is called once a satisfying assignment to the current propositional encoding is found.
	add_clause(c)	Called by deduce() to add a clause as a consequence of a T-inconsistent assignment
Members:	f	A copy of φ
	skeleton	An instance of skeletont

Table B.9. Interface of the classes lazy_dpllt and dpll_tt, which are implementations of Lazy-DPLL (Algorithm 3.3.2) and DPLL(T) (Algorithm 3.4.1). The theory T is assumed to be defined over a signature Σ

The templates that we have provided implement two of the algorithms given in Chap. 3: Lazy-DPLL and DPLL(T). These templates include the conversion of the propositional skeleton of φ into CNF, and the interface to the SAT solver. We provide a common interface to both algorithms, which is given in Table B.9.

Class:	`dpll_tt`, derived from `decision_proceduret`	
Methods:	`deduce()`	This method is called by the SAT solver to check a partial assignment for T-consistency.
	`add_clause(`c`)`	Called to add a clause as consequence of `assignment`
	`theory_implication(`n`,` v`)`	Called to communicate a T-implication to the SAT solver: n is the node implied, and v is the value.
Members:	`f`	A copy of φ
	`skeleton`	An instance of `skeletont`

Table B.10. Interface of the class `dpll_tt`, an implementation of DPLL(T) (Algorithm 3.4.1)

The only part that is left open is the interface to the decision procedure for the conjunction of Σ-literals. In the case of both algorithms, this is the method `deduce()`. The assignment to the Σ-literals is passed from the SAT solver to the deductive engine by means of calls to the method `assignment(`n`,` v`)`, where n is the node and v is the value that is assigned.

The method `deduce()` inspects this assignment to the Σ-literals. If it is found to be consistent, `deduce()` is expected to return TRUE. Otherwise, it is expected to add appropriate constraints using the method `add_clause`, and to return FALSE.

In the case of LAZY-DPLL, `deduce()` is called only for full assignments, whereas DPLL(T) may call `deduce()` for partial assignments.

References

1. W. Ackermann. *Solvable cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. V. S. Adve and J. M. Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In J. W. Davidson, K. D. Cooper, and A. M. Berman, editors, *PLDI*, pages 186–198. ACM, 1998.
3. J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In N. Sharygina and H. Veith, editors, *Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013.
4. A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *5th European Conference on Planning (ECP)*, volume 1809 of *LNCS*, pages 97–108. Springer, 1999.
5. A. Armando and E. Giunchiglia. Embedding complex decision procedures inside an interactive theorem prover. *Annals of Mathematics and Artificial Intelligence*, 8(3–4):475–502, 1993.
6. A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Inf. Comput.*, 183(2):140–164, 2003.
7. G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In *18th International Conference on Automated Deduction (CADE)*, volume 2392 of *LNCS*, pages 195–210. Springer, 2002.
8. G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In C. Boutilier, editor, *21st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 399–404, 2009.
9. A. Ayari and D. A. Basin. QUBOS: deciding quantified boolean logic using propositional satisfiability solvers. In M. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design, 4th International Conference (FMCAD)*, volume 2517 of *LNCS*, pages 187–201. Springer, 2002.
10. L. Bachmair and A. Tiwari. Abstract congruence closure and specializations. In D. A. McAllester, editor, *17th International Conference on Automated Deduction (CADE)*, volume 1831 of *LNCS*, pages 64–78. Springer, 2000.
11. E. Balas, S. Ceria, G. Cornuejols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19:1–9, 1996.

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0

12. T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *LNCS*, pages 268–283. Springer, 2001.
13. T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, volume 2057 of *LNCS*, 2001.
14. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *International Conference on Computer-Aided Verification (CAV)*, volume 3114 of *LNCS*. Springer, 2004.
15. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *23rd International Conference on Computer Aided Verification (CAV)*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.
16. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In M. K. Srivas and A. J. Camilleri, editors, *Formal Methods in Computer-Aided Design, First International Conference (FMCAD)*, volume 1166 of *LNCS*, pages 187–201. Springer, 1996.
17. C. Barrett and C. Tinelli. CVC3. In *19th International Conference on Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.
18. C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Design Automation Conference (DAC)*, pages 522–527. ACM Press, 1998.
19. C. W. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 236–249. Springer, 2002.
20. C. W. Barrett, D. L. Dill, and A. Stump. A generalization of Shostak's method for combining decision procedures. In A. Armando, editor, *Frontiers of Combining Systems, 4th International Workshop (FroCos)*, volume 2309 of *LNCS*, pages 132–146. Springer, 2002.
21. R. Bayardo and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In B. Kuipers and B. L. Webber, editors, *Fourteenth National Conference on Artificial Intelligence (AAAI)*, pages 203–208. AAAI Press, 1997.
22. P. Beame, H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.
23. E. Ben-Sasson and A. Wigderson. Short proofs are narrow – resolution made simple. *J. ACM*, 48(2):149–169, 2001.
24. M. Benedikt, T. W. Reps, and S. Sagiv. A decidable logic for describing linked data structures. In *Programming Languages and Systems, 8th European Symposium on Programming (ESOP)*, volume 1576 of *LNCS*, pages 2–19. Springer, 1999.
25. J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In *Foundations of Software Technology and Theoretical Computer Science, 24th International Conference (FSTTCS)*, volume 3328 of *LNCS*, pages 97–109. Springer, 2004.

26. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *Programming Languages and Systems, 3rd Asian Symposium, (APLAS)*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.
27. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects (FMCO)*, volume 4111 of *LNCS*, pages 115–137. Springer, 2006.
28. D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In *Computer Aided Verification, 18th International Conference (CAV)*, volume 4144 of *LNCS*, pages 532–546. Springer, 2006.
29. A. Biere. Resolve and expand. In *7th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2004.
30. A. Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.
31. A. Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT race 2010. SAT race: system description, 2010.
32. A. Biere, D. L. Berre, E. Lonca, and N. Manthey. Detecting cardinality constraints in CNF. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing (SAT), 17th International Conference*, volume 8561 of *LNCS*, pages 285–301. Springer, 2014.
33. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.
34. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference (DAC)*, pages 317–320. ACM, 1999.
35. A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* IOS Press, Amsterdam, The Netherlands, 2009.
36. J. D. Bingham and Z. Rakamaric. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3855 of *LNCS*, pages 207–221. Springer, 2006.
37. C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *23rd International Conference on Data Engineering (ICDE)*, pages 506–515. IEEE, 2007.
38. C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. QAGen: generating query-aware test databases. In *ACM SIGMOD International Conference on Management of Data*, pages 341–352. ACM, 2007.
39. N. Bjørner. Linear quantifier elimination as an abstract decision procedure. In J. Giesl and R. Hähnle, editors, *Automated Reasoning, 5th International Joint Conference (IJCAR)*, volume 6173 of *LNCS*, pages 316–330. Springer, 2010.
40. M. P. Bonacina, S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decidability and undecidability results for Nelson–Oppen and rewrite-based decision procedures. In *Automated Reasoning (IJCAR)*, volume 4130 of *LNCS*, pages 513–527. Springer, 2006.
41. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *17th International Conference on Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 335–349, 2005.

42. A. R. Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87, 2011.
43. A. R. Bradley and Z. Manna. *The Calculus of Computation*. Springer, 2007.
44. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference (VMCAI)*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.
45. M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design*, 45(2):213–245, 2014.
46. M. Brain, L. Hadarean, D. Kroening, and R. Martins. Automatic generation of propagation complete SAT encodings. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 536–556. Springer, 2016.
47. M. Brain, C. Tinelli, P. Rümmer, and T. Wahl. An automatable formal semantics for IEEE-754 floating-point arithmetic. In *22nd IEEE Symposium on Computer Arithmetic (ARITH)*, pages 160–167. IEEE, 2015.
48. A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures & Algorithms*, 27(2):201–226, 2005.
49. A. Brillout, D. Kroening, and T. Wahl. Mixed abstractions for floating-point arithmetic. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 69–76. IEEE, 2009.
50. R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *Proceedings of VLSI Design*, pages 741–746. IEEE, 2002.
51. R. Brummayer and A. Biere. C32SAT: Checking C expressions. In *Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 294–297. Springer, 2007.
52. R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In S. Kowalewski and A. Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *LNCS*, pages 174–177. Springer, 2009.
53. R. Brummayer and A. Biere. Lemmas on demand for the extensional theory of arrays. *JSAT*, 6(1-3):165–201, 2009.
54. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A lazy and layered SMT($\mathcal{BV}$) solver for hard industrial verification problems. In *Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 547–560. Springer, 2007.
55. R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.
56. R. Bryant, S. German, and M. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *11th International Conference on Computer Aided Verification (CAV)*, volume 1633 of *LNCS*. Springer, 1999.
57. R. Bryant and M. Velev. Boolean satisfiability with transitivity constraints. In *12th International Conference on Computer Aided Verification (CAV)*, volume 1855 of *LNCS*. Springer, 2000.
58. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*. Springer, 2002.

59. H. K. Büning, M. Karpinski, and A. Flögel. Resolution for quantified Boolean formulas. *Information and Computation*, 117(1):12–18, 1995.
60. R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
61. W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw., Pract. Exper.*, 30(7):775–802, 2000.
62. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In R. Draves and R. van Renesse, editors, *OSDI*, pages 209–224. USENIX Association, 2008.
63. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *Static Analysis, 13th International Symposium (SAS)*, volume 4134 of *LNCS*, pages 182–203. Springer, 2006.
64. C. Calcagno, H. Yang, and P. W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 2245 of *LNCS*, pages 108–119. Springer, 2001.
65. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In L. A. Clarke, L. Dillon, and W. F. Tichy, editors, *ICSE*, pages 385–395. IEEE Computer Society, 2003.
66. S. Chaki, A. Gurfinkel, and O. Strichman. Regression verification for multi-threaded programs (with extensions to locks and dynamic thread creation). *Formal Methods in System Design*, 47(3):287–301, 2015.
67. P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *12th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 331–337, 1991.
68. H. Chen, C. David, D. Kroening, P. Schrammel, and B. Wachter. Synthesising interprocedural bit-precise termination proofs. In M. B. Cohen, L. Grunske, and M. Whalen, editors, *Automated Software Engineering (ASE)*, pages 53–64. IEEE, 2015.
69. Y. Chen, N. Dalchau, N. Srinivas, A. Phillips, L. Cardelli, D. Soloveichik, and G. Seelig. Programmable chemical controllers made from DNA. *Nature nanotechnology*, 8(10):755–762, 2013.
70. J. Christ and J. Hoenicke. Weakly equivalent arrays. In P. Rümmer and C. M. Wintersteiger, editors, *Satisfiability Modulo Theories (SMT)*, volume 1163 of *CEUR Workshop Proceedings*, pages 39–49, 2014.
71. A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In N. Piterman and S. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.
72. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.
73. E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.
74. B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger. Ranking function synthesis for bit-vector relations. *Formal Methods in System Design*, 43(1):93–120, 2013.

75. B. Cook, D. Kroening, and N. Sharygina. Accurate theorem proving for program verification. In *Leveraging Applications of Formal Methods, First International Symposium (ISoLA)*, volume 4313 of *LNCS*, pages 96–114. Springer, 2006.
76. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In M. I. Schwartzbach and T. Ball, editors, *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI)*, pages 415–426. ACM, 2006.
77. S. Cook. The complexity of theorem-proving procedures. In *3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.
78. D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, pages 91–100. Edinburgh University Press, 1972.
79. F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *13th International Conference on Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 436–453, 2001.
80. L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based Bounded Model Checking for embedded ANSI-C software. In *ASE*, pages 137–148. IEEE Computer Society, 2009.
81. F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. *Formal Methods in System Design*, 10(1):7–46, 1997.
82. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*, chapter 25.3, pages 532–536. MIT Press, 2000.
83. D. W. Currie, A. J. Hu, and S. P. Rajan. Automatic formal verification of DSP software. In *Design Automation Conference (DAC)*, pages 130–135. ACM, 2000.
84. G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1963.
85. C. David, D. Kroening, and M. Lewis. Propositional reasoning about safety and termination of heap-manipulating programs. In J. Vitek, editor, *Programming Languages and Systems, 24th European Symposium on Programming (ESOP)*, volume 9032 of *LNCS*, pages 661–684. Springer, 2015.
86. C. David, D. Kroening, and M. Lewis. Unrestricted termination and non-termination arguments for bit-vector programs. In J. Vitek, editor, *Programming Languages and Systems, 24th European Symposium on Programming (ESOP)*, volume 9032 of *LNCS*, pages 183–204. Springer, 2015.
87. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.
88. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
89. H. de Jong. Modeling and simulation of genetic regulatory systems: A literature review. *Journal of Computational Biology*, 9(1):67–103, 2002.
90. L. de Moura. System description: Yices 0.1. Technical report, Computer Science Laboratory, SRI International, 2005.
91. L. de Moura and N. Bjørner. Model-based theory combination. In *Satisfiability Modulo Theories (SMT)*, 2007.
92. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and*

Analysis of Systems, 14th International Conference (TACAS), volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

93. L. de Moura and H. Ruess. Lemmas on demand for satisfiability solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, 2002.
94. L. de Moura and H. Ruess. An experimental evaluation of ground decision procedures. In *16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 162–174. Springer, 2004.
95. L. de Moura, H. Ruess, and N. Shankar. Justifying equality. In *Second Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR)*, 2004.
96. L. M. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *Automated Deduction, 21st International Conference on Automated Deduction (CADE)*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.
97. L. M. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *9th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 45–52, 2009.
98. R. Dechter. *Constraint Processing.* The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, 2003.
99. N. Dershowitz, Z. Hanna, and A. Nadel. A clause-based heuristic for SAT solvers. In F. Bacchus and T. Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference (SAT)*, volume 3569 of *LNCS*, pages 46–60. Springer, 2005.
100. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science (Volume B): Formal Models and Semantics*, pages 243–320. MIT Press, 1990.
101. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.
102. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Programming Language Design and Implementation (PLDI)*, pages 230–241. ACM, 1994.
103. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.
104. A. Dolzmann, T. Sturm, and V. Weispfenning. Real quantifier elimination in practice. Technical Report MIP9720, FMI, Universität Passau, Dec. 1997.
105. P. J. Downey. Undecidability of Presburger arithmetic with a single monadic predicate letter. Technical Report TR-18-72, Center for Research in Computing Technology, Harvard University, 1972.
106. P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, October 1980.
107. S. Dunn, G. Martello, B. Yordanov, S. Emmott, and A. Smith. Defining an essential transcription factor program for naïve pluripotency. *Science*, 344(6188), 2014.
108. B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *18th International Conference on Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.
109. B. Dutertre and L. de Moura. Integrating Simplex with DPLL(T). Technical Report SRI-CSL-06-01, Stanford Research Institute (SRI), 2006.

110. N. Eén and N. Sörensson. An extensible SAT-solver [ver 1.2]. In *Theory and Applications of Satisfiability Testing*, volume 2919 of *LNCS*, pages 512–518. Springer, 2003.
111. D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *18th ACM Symposium on Operating System Principles (SOSP)*, pages 57–72. ACM, 2001.
112. J. Ferrante and C. Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM J. Comput.*, 4(1):69–76, 1975.
113. J. Filliatre, S. Owre, H. Ruess, and N. Shankar. ICS: Integrated canonizer and solver. In *13th International Conference on Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 246–249. Springer, 2001.
114. M. J. Fischer and M. O. Rabin. Super-exponential complexity of Presburger arithmetic. In *SIAM-AMS Symposium in Applied Mathematics*, volume 7, pages 27–41, 1974.
115. C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explication. In *Computer Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 355–367. Springer, 2003.
116. R. Floyd. Assigning meanings to programs. *Symposia in Applied Mathematics*, 19:19–32, 1967.
117. V. Ganesh, S. Berezin, and D. Dill. Deciding Presburger arithmetic by model checking and comparisons with other methods. In *Formal Methods in Computer-Aided Design (FMCAD)*, volume 2517 of *LNCS*, pages 171–186. Springer, 2002.
118. V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 519–531. Springer, 2007.
119. H. Ganzinger. Shostak light. In A. Voronkov, editor, *Automated Deduction, 18th International Conference on Automated Deduction (CADE)*, volume 2392 of *LNCS*, pages 332–346. Springer, 2002.
120. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.
121. Y. Ge, C. W. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. *Ann. Math. Artif. Intell.*, 55(1-2):101–122, 2009.
122. A. V. Gelder and Y. Tsuji. Incomplete thoughts about incomplete satisfiability procedures. In *2nd DIMACS Challenge Workshop: Cliques, Coloring and Satisfiability*, 1993.
123. R. Gershman, M. Koifman, and O. Strichman. Deriving small unsatisfiable cores with dominators. In *18th International Conference on Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 109–122. Springer, 2006.
124. R. Gershman and O. Strichman. HaifaSat: A new robust SAT solver. In *1st International Haifa Verification Conference*, volume 3875 of *LNCS*, pages 76–89. Springer, 2005.
125. M. Ghasemzadeh, V. Klotz, and C. Meinel. Embedding memoization to the semantic tree search for deciding QBFs. In *Australian Conference on Artificial Intelligence*, pages 681–693, 2004.

126. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Deciding extensions of the theory of arrays by integrating decision procedures and instantiation strategies. In *Logics in Artificial Intelligence (JELIA)*, volume 4160 of *LNCS*, pages 177–189. Springer, 2006.
127. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision procedures for extensions of the theory of arrays. *Ann. Math. Artif. Intell.*, 50(3-4):231–254, 2007.
128. E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. `www.qbflib.org`.
129. F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures – the case study of modal K. In *Automated Deduction (CADE)*, volume 1104 of *LNCS*, pages 583–597. Springer, 1996.
130. P. Godefroid, J. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25(5):30–37, 2008.
131. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In V. Sarkar and M. W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.
132. A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal. BDD based procedures for a theory of equality with uninterpreted functions. In *Computer Aided Verification, 10th International Conference (CAV)*, volume 1427 of *LNCS*, pages 244–255. Springer, 1998.
133. E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT-solver. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, page 142, 2002.
134. R. Gomory. An algorithm for integer solutions to linear problems. In *Recent Advances in Mathematical Programming*, pages 269–302, New York, 1963. McGraw-Hill.
135. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification, 9th International Conference (CAV)*, volume 1254 of *LNCS*. Springer, 1997.
136. L. Hadarean, K. Bansal, D. Jovanovic, C. Barrett, and C. Tinelli. A tale of two solvers: Eager and lazy approaches to bit-vectors. In A. Biere and R. Bloem, editors, *Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, pages 680–695. Springer, 2014.
137. A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.
138. J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2008.
139. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.
140. M. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Haifa Verification Conference*, volume 7261 of *LNCS*, pages 50–65. Springer, 2011.
141. J. Heusser and P. Malacaria. Quantifying information leaks in software. In C. Gates, M. Franz, and J. P. McDermott, editors, *Twenty-Sixth Annual Computer Security Applications Conference (ACSAC)*, pages 261–269. ACM, 2010.
142. F. Hillier and G. Lieberman. *Introduction to Mathematical Programming*. McGraw-Hill, 1990.

143. E. A. Hirsch and A. Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*, 43(1):91–111, 2005.
144. C. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.
145. C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2(4):335–355, December 1973.
146. R. Hojati, A. Isles, D. Kirkpatrick, and R. Brayton. Verification using uninterpreted functions and finite instantiations. In *1st International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, volume 1166 of *LNCS*, pages 218–232. Springer, 1996.
147. R. Hojati, A. Kuehlmann, S. German, and R. Brayton. Validity checking in the theory of equality using finite instantiations. In *International Workshop on Logic Synthesis*, 1997.
148. J. N. Hooker. Solving the incremental satisfiability problem. *Journal of Logic Programming*, 15:177–186, 1993.
149. I. Horrocks. The FaCT system. In H. de Swart, editor, *TABLEAUX-98*, volume 1397 of *LNAI*, pages 307–312. Springer, 1998.
150. J. Huang. MUP: A minimal unsatisfiability prover. In *10th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 432–437, 2005.
151. G. Huet and D. Oppen. Equations and rewrite rules: A survey. In *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, 1980.
152. A. P. Hurst, P. Chong, and A. Kuehlmann. Physical placement driven by sequential timing analysis. In *International Conference on Computer-Aided Design (ICCAD)*, pages 379–386. IEEE Computer Society/ACM, 2004.
153. N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *Computer Science Logic (CSL)*, volume 3210 of *LNCS*, pages 160–174. Springer, 2004.
154. O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In A. Biere and R. Bloem, editors, *Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, pages 585–602. Springer, 2014.
155. F. Ivancic, I. Shlyakhter, A. Gupta, and M. K. Ganai. Model checking C programs using F-SOFT. In *International Conference on Computer Design (ICCD)*, pages 297–308. IEEE Computer Society, 2005.
156. J. Jaffar. Presburger arithmetic with array segments. *Inf. Process. Lett.*, 12(2):79–82, 1981.
157. G. Johnson. Separating the insolvable and the merely difficult. New York Times, July 13 1999.
158. D. Jovanovic and C. Barrett. Polite theories revisited. In C. G. Fermüller and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 17th International Conference (LPAR)*, volume 6397 of *LNCS*, pages 402–416. Springer, 2010.
159. D. Jovanovic and C. Barrett. Sharing is caring: Combination of theories. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium (FroCoS)*, volume 6989 of *LNCS*, pages 195–210. Springer, 2011.

160. T. Jussila, A. Biere, C. Sinz, D. Kroening, and C. M. Wintersteiger. A first step towards a unified proof checker for QBF. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 4501 of *LNCS*, pages 201–214. Springer, 2007.
161. A. Kaiser, D. Kroening, and T. Wahl. Efficient coverability analysis by proof minimization. In M. Koutny and I. Ulidowski, editors, *CONCUR*, volume 7454 of *LNCS*, pages 500–515. Springer, 2012.
162. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic, 2000.
163. S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference (TACAS)*, pages 553–568, 2003.
164. J. Kim, J. Whittemore, J. Silva, and K. Sakallah. Incremental Boolean satisfiability and its application to delay fault testing. In *IEEE/ACM International Workshop on Logic Synthesis (IWLS)*, June 1999.
165. J. C. King. A new approach to program testing. In C. Hackl, editor, *Programming Methodology*, volume 23 of *LNCS*, pages 278–290. Springer, 1974.
166. S. Kirkpatrick and B. Selman. Critical behavior in the satisfiability of random Boolean expressions. *Science*, 264(5163):1297–1301, 1994.
167. D. E. Knuth. *The Art of Computer Programming Volume 4, Fascicle 1.* Addison-Wesley Professional, 2009. Bitwise tricks & techniques; Binary Decision Diagrams.
168. D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6.* Pearson Education, 2016. Satisfiability.
169. L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In N. Sharygina and H. Veith, editors, *Computer Aided Verification, 25th International Conference (CAV)*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.
170. D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Design Automation Conference (DAC)*, pages 368–371. ACM, 2003.
171. D. Kroening and O. Strichman. A framework for satisfiability modulo theories. *Formal Aspects of Computing*, 21(5):485–494, 2009.
172. V. Kuncak and M. C. Rinard. Existential heap abstraction entailment is undecidable. In *Static Analysis (SAS)*, volume 2694 of *LNCS*, pages 418–438. Springer, 2003.
173. R. Kurshan. *Computer Aided Verification of Coordinating Processes.* Princeton University Press, 1994.
174. S. K. Lahiri, R. E. Bryant, A. Goel, and M. Talupur. Revisiting positive equality. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 1–15. Springer, 2004.
175. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Principles of Programming Languages (POPL)*, pages 115–126. ACM, 2006.
176. S. K. Lahiri and S. Qadeer. Back to the future: Revisiting precise program verification using SMT solvers. In *Principles of Programming Languages (POPL)*, pages 171–182. ACM, 2008.
177. A. Lal and S. Qadeer. Powering the Static Driver Verifier using Corral. In S. Cheung, A. Orso, and M. D. Storey, editors, *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 202–212. ACM, 2014.

178. A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification (CAV)*, volume 7358 of *LNCS*, pages 427–443. Springer, 2012.
179. A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In A. Gupta and S. Malik, editors, *Computer Aided Verification (CAV)*, volume 5123 of *LNCS*, pages 37–51. Springer, 2008.
180. S. Lee and D. Plaisted. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, 9:25–42, 1992.
181. K. R. M. Leino. *Toward reliable modular programs.* PhD thesis, CalTech, 1995. Available as Technical Report Caltech-CS-TR-95-03.
182. M. Levine and E. Davidson. Gene regulatory networks for development. *Proceedings of the National Academy of Sciences*, 102(14):4936–4942, 2005.
183. R. Loos and V. Weispfenning. Applying linear quantifier elimination. *Comput. J.*, 36(5):450–462, 1993.
184. I. Lynce and J. Marques-Silva. On computing minimum unsatisfiable cores. In *International Symposium on Theory and Applications of Satisfiability Testing (SAT)*, pages 305–310, 2004.
185. M. Mahfoudh. *On Satisfiability Checking for Difference Logic.* PhD thesis, Verimag, France, 2003.
186. M. Mahfoudh, P. Niebert, E. Asarin, and O. Maler. A satisfiability checker for difference logic. In *5th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 222–230, 2002.
187. R. Manevich, E. Yahav, G. Ramalingam, and S. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference (VMCAI)*, volume 3385 of *LNCS*, pages 181–198. Springer, 2005.
188. P. Manolios, S. K. Srinivasan, and D. Vroon. Automatic memory reductions for RTL model verification. In *International Conference on Computer-Aided Design (ICCAD)*, pages 786–793. ACM, 2006.
189. P. Manolios and D. Vroon. Efficient circuit to CNF conversion. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 4501 of *LNCS*, pages 4–9. Springer, 2007.
190. P. Mateti. A decision procedure for the correctness of a class of programs. *J. ACM*, 28(2):215–232, 1981.
191. J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Proceedings Symposium in Applied Mathematics, Volume 19, Mathematical Aspects of Computer Science*, pages 33–41. American Mathematical Society, 1967.
192. K. McMillan. *Symbolic Model Checking.* Kluwer Academic, 1993.
193. K. L. McMillan. Interpolation and SAT-based model checking. In *15th International Conference on Computer Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 1–13. Springer, Jul 2003.
194. O. Meir and O. Strichman. Yet another decision procedure for equality logic. In *17th International Conference on Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 307–320. Springer, 2005.
195. F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In R. Joshi, P. Müller, and A. Podelski, editors, *VSTTE*, volume 7152 of *LNCS*, pages 146–161. Springer, 2012.
196. M. Mezard, G. Parisi, and R. Zecchina. Analytic and algorithmic solution of random satisfiability problems. *Science*, 297(5582):812–815, 2002.

197. D. G. Mitchell, B. Selman, and H. J. Levesque. Hard and easy distributions for SAT problems. In *Tenth National Conference on Artificial Intelligence*, pages 459–465. AAAI Press, 1992.
198. A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 221–231. ACM, 2001.
199. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *13th International Conference on Computer Science Logic*, volume 1683 of *LNCS*, pages 111–125. Springer, 1999.
200. R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic phase transitions. *Nature*, 400(8):133–137, 1999.
201. U. Montanari. Networks of constraints: fundamental properties and applications to picture processing. *Information Science*, 7, 1976.
202. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference (DAC)*, June 2001.
203. A. Nadel. Boosting minimal unsatisfiable core extraction. In R. Bloem and N. Sharygina, editors, *10th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 221–229. IEEE, 2010.
204. G. Nelson. Verifying reachability invariants of linked structures. In *Tenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 38–47. ACM, 1983.
205. G. Nelson and D. C. Oppen. Fast decision procedures based on UNION and FIND. In *18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 114–119. IEEE, 1977.
206. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, Oct 1979.
207. G. Nelson and F. F. Yao. Solving reachability constraints for linear lists. Technical report, 1982.
208. G. Nemhauser and L. Wosley. *Integer and Combinatorial Optimization.* Wiley-Interscience, 1999.
209. R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *17th International Conference on Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 321–334. Springer, 2005.
210. R. Nieuwenhuis and A. Oliveras. Proof-producing congruence closure. In *16th International Conference on Term Rewriting and Applications (RTA)*, volume 3467 of *LNCS*, pages 453–468. Springer, 2005.
211. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.
212. E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Principles and Practice of Constraint Programming (CP)*, pages 438–452, 2004.
213. C. Oh. Between SAT and UNSAT: the fundamental difference in CDCL SAT. In M. Heule and S. Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT*, volume 9340 of *LNCS*, pages 307–323, 2015.

214. Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. AMUSE: A minimally-unsatisfiable subformula extractor. In *Design Automation Conference (DAC)*, pages 518–523, 2004.
215. D. C. Oppen. Complexity, convexity and combinations of theories. *Theoretical Computer Science*, 12(3):291–302, 1980.
216. N. Paoletti, B. Yordanov, Y. Hamadi, C. Wintersteiger, and H. Kugler. Analyzing and synthesizing genomic logic functions. In *CAV*, volume 8559 of *LNCS*. Springer, 2014.
217. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 247–258. ACM, 2005.
218. G. Parthasarathy, M. K. Iyer, K.-T. Cheng, and L.-C. Wang. An efficient finite-domain constraint solver for circuits. In *Design Automation Conference (DAC)*, pages 212–217. ACM Press, 2004.
219. P. F. Patel-Schneider. DLP system description. In E. Franconi, G. D. Giacomo, R. M. MacGregor, W. Nutt, and C. A. Welty, editors, *International Workshop on Description Logics (DL)*, volume 11 of *CEUR Workshop Proceedings*, pages 87–89, 1998.
220. J. Petke and P. Jeavons. The order encoding: From tractable CSP to tractable SAT. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 6695 of *Lecture Notes in Computer Science*, pages 371–372. Springer, 2011.
221. A. Phillips and L. Cardelli. A programming language for composable DNA circuits. *Journal of The Royal Society Interface*, 6(4):1470–1485, 2009.
222. K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *10th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 294–299, 2007.
223. K. Pipatsrisawat and A. Darwiche. RSAT 2.0: SAT solver description, 2007. SAT solvers competition.
224. R. Piskac, L. de Moura, and N. Bjørner. Deciding effectively propositional logic with equality. Technical Report MSR-TR-2008-181, Microsoft Research, 2008.
225. D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.
226. A. Pnueli, Y. Rodeh, and O. Shtrichman. Range allocation for equivalence logic. In R. Hariharan, M. Mukund, and V. Vinay, editors, *21st Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 2245 of *LNCS*, pages 317–333. Springer, 2001.
227. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domains instantiations. In *11th International Conference on Computer Aided Verification (CAV)*, volume 1633 of *LNCS*. Springer, 1999.
228. A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel. The small model property: How small can it be? *Information and Computation*, 178(1):279–293, Oct. 2002.
229. A. Pnueli and O. Strichman. Reduced functional consistency of uninterpreted functions. In *Pragmatics of Decision Procedures for Automated Reasoning (PDPAR)*, number 898 in Electronic Notes in Computer Science, 2005.
230. A. Podelski and T. Wies. Boolean heaps. In *Static Analysis, 12th International Symposium (SAS)*, volume 3672 of *LNCS*, pages 268–283. Springer, 2005.
231. V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, 1977.

232. M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*, pages 92–101, Warszawa, 1929.
233. W. Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992.
234. L. Qian and E. Winfree. Scaling up digital circuit computation with DNA strand displacement cascades. *Science*, 332(6034):1196–1201, 2011.
235. L. Qian, E. Winfree, and J. Bruck. Neural network computation with DNA strand displacement cascades. *Nature*, 475(7356):368–372, 2011.
236. M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, July 1969.
237. I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification, 17th International Conference (CAV)*, volume 3576 of *LNCS*, pages 82–97. Springer, 2005.
238. S. Ranise, C. Ringeissen, and C. G. Zarba. Combining data structures with nonstably infinite theories using many-sorted logic. In B. Gramlich, editor, *Frontiers of Combining Systems, 5th International Workshop (FroCoS)*, volume 3717 of *LNCS*, pages 48–64. Springer, 2005.
239. S. Ranise and C. Tinelli. The SMT-LIB standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at www.SMT-LIB.org.
240. T. W. Reps, S. Sagiv, and R. Wilhelm. Static program analysis via 3-valued logic. In *Computer Aided Verification, 16th International Conference (CAV)*, volume 3114 of *LNCS*, pages 15–30. Springer, 2004.
241. J. C. Reynolds. Reasoning about arrays. *Communications of the ACM*, 22(5):290–299, 1979.
242. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74. IEEE Computer Society, 2002.
243. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.
244. J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.
245. Y. Rodeh. *Techniques in Decision Procedures for Equality Logic and Improved Model Checking Methods*. PhD thesis, Weizmann Institute of Science, 2003.
246. Y. Rodeh and O. Shtrichman. Finite instantiations in equivalence logic with uninterpreted functions. In *Computer Aided Verification (CAV)*, 2001.
247. M. Rozanov and O. Strichman. Generating minimum transitivity constraints in P-time for deciding equality logic. In *Satisfiability Modulo Theories (SMT)*, 2007.
248. L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.
249. V. Ryvchin and O. Strichman. Faster extraction of high-level minimal unsatisfiable cores. In K. A. Sakallah and L. Simon, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 6695 of *LNCS*, pages 174–187. Springer, 2011.

250. A. Sabharwal, C. Ansótegui, C. P. Gomes, J. W. Hart, and B. Selman. QBF modeling: Exploiting player symmetry for simplicity and efficiency. In *9th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 382–395, 2006.
251. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, pages 105–118, 1999.
252. A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1998.
253. R. Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, 3, 2007.
254. B. Selman, H. J. Levesque, and D. G. Mitchell. A new method for solving hard satisfiability problems. In *10th National Conference on Artificial Intelligence (AAAI)*, pages 440–446, 1992.
255. K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In T. Ball and R. B. Jones, editors, *18th International Conference on Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 419–423. Springer, 2006.
256. S. Seshia, S. Lahiri, and R. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *Proc. of Design Automation Conference (DAC)*, pages 425–430, 2003.
257. N. Shankar and H. Ruess. Combining Shostak theories. In S. Tison, editor, *Rewriting Techniques and Applications, 13th International Conference (RTA)*, volume 2378 of *LNCS*, pages 1–18. Springer, 2002.
258. R. E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, 1978.
259. R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.
260. O. Shtrichman. Sharing information between instances of a propositional satisfiability (SAT) problem, Dec 2000. US Patent 2002/0123867 A1.
261. O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *LNCS*, pages 58–70. Springer, 2001.
262. J. Silva and K. Sakallah. GRASP – a new search algorithm for satisfiability. Technical Report TR-CSE-292996, University of Michigan, 1996.
263. E. Singerman. Challenges in making decision procedures applicable to industry. In PDPAR 2005. *Electronic Notes in Computer Science*, 144(2), 2006.
264. L. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1977.
265. L. Stockmeyer and A. Meyer. Word problems requiring exponential time. In *5th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–9. ACM, 1973.
266. O. Strichman. On solving Presburger and linear arithmetic with SAT. In *Formal Methods in Computer-Aided Design (FMCAD)*, volume 2517 of *LNCS*, pages 160–170. Springer, 2002.
267. O. Strichman, S. Seshia, and R. Bryant. Deciding separation formulas with SAT. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 209–222. Springer, July 2002.
268. A. Stump, C. Barrett, and D. Dill. CVC: a cooperating validity checker. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 500–504. Springer, 2002.

269. A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for an extensional theory of arrays. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 29–37. IEEE, 2001.
270. A. Stump and L.-Y. Tan. The algebra of equality proofs. In *Term Rewriting and Applications, 16th International Conference (RTA)*, volume 3467 of *LNCS*, pages 469–483. Springer, 2005.
271. G. Sutcliffe and C. Suttner. The state of CASC. *AI Communications*, 19(1):35–48, 2006.
272. N. Suzuki and D. Jefferson. Verification decidability of Presburger array programs. *J. ACM*, 27(1):191–205, 1980.
273. M. Talupur, N. Sinha, O. Strichman, and A. Pnueli. Range-allocation for separation logic. In *16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 148–161. Springer, July 2004.
274. C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Proc. 8th European Conference on Logics in Artificial Intelligence*, volume 2424 of *LNAI*, pages 308–319. Springer, 2002.
275. C. Tinelli and M. T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In *Frontiers of Combining Systems: 1st International Workshop*, pages 103–120. Kluwer Academic, 1996.
276. C. Tinelli and C. Zarba. Combining nonstably infinite theories. *Journal of Automated Reasoning*, 34(3):209–238, 2005.
277. G. Tseitin. On the complexity of proofs in propositional logics. In *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*, volume 2. Springer, 1983. Originally published 1970.
278. R. J. Vanderbei. *Linear Programming: Foundations and Extensions.* Kluwer, 1996.
279. M. Veksler and O. Strichman. Learning general constraints in CSP. *Artificial Intelligence*, 238:135–153, 2016.
280. W. Visser, S. Park, and J. Penix. Using predicate abstraction to reduce object-oriented programs for model checking. In M. P. E. Heimdahl, editor, *Third Workshop on Formal Methods in Software Practic (FMSP)*, pages 3–182. ACM, 2000.
281. J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *IEEE/ACM Design Automation Conference (DAC)*, 2001.
282. R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *Compiler Construction (CC)*, volume 1781 of *LNCS*, pages 1–17. Springer, 2000.
283. H. P. Williams. Fourier-Motzkin elimination extension to integer programming problems. *Journal of Combinatorial Theory*, 21:118–123, July 1976.
284. R. Williams, C. P. Gomes, and B. Selman. Backdoors to typical case complexity. In *IJCAI*, pages 1173–1178. Morgan Kaufmann, 2003.
285. J. Wilson. Compact normal forms in propositional logic. *Computers and Operations Research*, 90:309–314, 1990.
286. S. A. Wolfman and D. S. Weld. The LPSAT engine & its application to resource planning. In T. Dean, editor, *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 310–317. Morgan Kaufmann, 1999.
287. P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *LNCS*, pages 1–19. Springer, 2000.
288. L. Wolsey. *Integer Programming.* Wiley-Interscience, 1998.

289. L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla-07: The design and analysis of an algorithm portfolio for SAT. In *Principles and Practice of Constraint Programming (CP)*, pages 712–727, 2007.

290. B. Yordanov, S.-J. Dunn, H. Kugler, A. Smith, G. Martello, and S. Emmott. A method to identify and analyze biological programs through automated reasoning. *Nature Systems Biology and Applications, In Press*, 2016.

291. B. Yordanov, C. Wintersteiger, Y. Hamadi, and H. Kugler. SMT-based analysis of biological computation. In *NFM*, volume 7871 of *LNCS*. Springer, 2013.

292. B. Yordanov, C. Wintersteiger, Y. Hamadi, and H. Kugler. Z34Bio: An SMT-based framework for analyzing biological computation. In *SMT*, 2013.

293. B. Yordanov, C. Wintersteiger, Y. Hamadi, A. Phillips, and H. Kugler. Functional analysis of large-scale DNA strand displacement circuits. In *Proc. 19th International Conference on DNA Computing and Molecular Programming (DNA 2013)*, volume 8141 of *LNCS*, pages 189–203, 2013.

294. H. Zantema and J. F. Groote. Transforming equality logic to propositional logic. *Electronic Notes in Computer Science*, 86(1), 2003.

295. L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *International Conference on Computer Aided Design (ICCAD)*, pages 279–285. IEEE, 2001.

296. L. Zhang and S. Malik. Conflict driven learning in a quantified Boolean satisfiability solver. In *International Conference on Computer Aided Design (ICCAD)*, 2002.

297. L. Zhang and S. Malik. Towards symmetric treatment of conflicts and satisfaction in quantified Boolean satisfiability solver. In P. V. Hentenryck, editor, *8th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 2470 of *LNCS*, pages 200–215. Springer, 2002.

298. L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In *Theory and Applications of Satisfiability Testing (SAT)*, 2003. Presentation only.

299. L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design and Test in Europe Conference (DATE)*, pages 10880–10885. IEEE, 2003.

Tools index

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0

Algorithms index

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0

Index

Note that there is a separate index page for tools on page 347, and a separate index page for algorithms on page 349.

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0