The Design and Algorithms of a Verification Condition Generator

FreeBoogie

Radu Grigore

The thesis is submitted to University College Dublin for the degree of PhD in the College of Engineering, Mathematical & Physical Sciences.

March 2010

School of Computer Science and Informatics
Head of School: Prof. Joe Carthy
Supervisor: Assoc. Prof. Joseph Roland Kiniry
Second supervisor: Prof. Simon Dobson

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 History . . . 3
- 1.3 Related Work . . . 5
- 1.4 A Guided Tour . . . 9

2 The Core Boogie **11**

3 Design Overview **18**
- 3.1 An Example Run . . . 18
- 3.2 Pipeline . . . 20
- 3.3 The Abstract Syntax Tree and its Visitors . . . 22
- 3.4 Auxiliary Information . . . 30
- 3.5 Verification Condition Generation . . . 32
- 3.6 The Prover Backend . . . 33
- 3.7 Other Generated Code . . . 37
- 3.8 Related Work . . . 37

4 Optimal Passive Form **43**
- 4.1 Background . . . 43
- 4.2 The Definition of Passive Form . . . 50
- 4.3 The Version-Optimal Passive Form . . . 56
- 4.4 The Copy-Optimal Passive Form . . . 61
- 4.5 Conclusions . . . 65
- 4.6 Related Work . . . 66

5 Strongest Postcondition versus Weakest Precondition **68**
- 5.1 Hoare Logic for Core Boogie . . . 68
- 5.2 Predicate Transformers . . . 70
- 5.3 Replacing Assignments by Assumptions . . . 74
- 5.4 Verification Condition Size . . . 75

5.5 Experiments . . . 81
5.6 Conclusions . . . 82
5.7 Related Work . . . 83

6 Edit and Verify **84**
6.1 Motivation . . . 84
6.2 Overview . . . 85
6.3 Simplifying SMT Formulas . . . 87
6.4 Correspondence between Trees . . . 95
6.5 Example . . . 101
6.6 Conclusions . . . 102
6.7 Related Work . . . 102

7 Semantic Reachability Analysis **104**
7.1 Motivation . . . 104
7.2 Theory . . . 108
7.3 Algorithm . . . 111
7.4 Case Study . . . 119
7.5 Conclusions and Related Work . . . 120

8 Conclusions **123**

A Notation **125**

Abstract

This dissertation provides a comprehensive description of the design and of the algorithms of the verification condition generator FreeBoogie. The main contributions are:

- the design of the VC generator, including discussions of the main design decisions;
- a precise definition of passivation and a study of its algorithmic properties;
- a comparison between the weakest precondition and the strongest postcondition methods of generating VCs;
- various semantics for a subset of the Boogie language—operational semantics, Hoare logic, predicate transformers—and a discussion of the relations between them;
- an algorithm for unsharing expressions, a problem that is usually tangled with computing the weakest precondition efficiently;
- a proof technique for the correctness of algorithms that simplify a VC based on a known old VC;
- a heuristic for detecting common parts of two expression trees, such as two VCs;
- the semantic reachability analysis, in the context of program verifiers;
- an efficient heuristic for finding dead code, doomed code, inconsistent specifications, and soundness bugs.

Chapter 1

Introduction

"The purpose of your paper is not ... to describe the WizWoz system. Your reader does not have a WizWoz. She is primarily interested in re-usable brain-stuff, not executable artifacts."
— Simon Peyton-Jones [144]

1.1 Motivation

Ideal programs are *correct*, *efficient*, and easy to *evolve*. Tools can help with all three aspects: Type-checkers ensure that certain classes of errors do not occur, profilers identify performance hot-spots, and IDEs (**i**ntegrated **d**evelopment **e**nvironments) refactor programs. Automation allows humans to focus on the interesting issues. Knuth [112] put it differently: "Science is knowledge which we understand so well that we can teach it to a computer; and if we don't fully understand something, it is an art to deal with." For example, a bit string can represent both a text and an integer, we understand well how to check that a program does not mix the two interpretations, and we leave the task to type-checkers. If, on the other hand, we want to check that a program computes the transitive closure of a graph we usually do it by hand.

A *program verifier* automatically checks whether code agrees with specifications. Figure 1.1 shows an example. The code is an implementation of the Roy–Warshall algorithm [157, 175]. The programmer spent energy to specify *what* the algorithm does (**requires**, **ensures**) and *how* it works (**invariant**). In words, $path(G, i, j)$ means that there is a path $i \rightsquigarrow j$ in the graph G, and $pathK(G, i, j, k)$ means that there is such a path whose intermediate nodes come only from the set $0\,..\,k-1$. The example illustrates what program verifiers

```
1  void fw(boolean[][] G) // G is an adjacency matrix
2    requires square(G);
3    requires (forall i; G[i][i]);
4    ensures (forall i, j; G[i][j] == path(old(G), i, j));
5  {
6    final int n = G.length;
7    for (int k = 0; k < n; ++k) invariant (forall i, j; G[i][j] == pathK(old(G), i, j, k));
8      for (int i = 0; i < n; ++i)
9        for (int j = 0; j < n; ++j)
10         G[i][j] = G[i][j] || (G[i][k] && G[k][j]);
11 }
12
13 axiom (forall G, i, j; path(G, i, j) = G[i][j] || (exists q; G[i][q] && path(G, q, j)));
14 axiom (forall G, i, j, k; pathK(G, i, j, k) = G[i][j] || (exists q; q<k & G[i][q] && pathK(G, q, j, k)));
```

Figure 1.1: An example of what program verifiers can ideally do

ought to be able to do in the future.

Note that it is trivial to establish the invariant (because *pathK* reduces to G when $k = 0$) and to infer the postcondition from the invariant (because *pathK* reduces to *path* when $k = n$). Proving that the invariant is preserved is trickier than it might seem mainly because of the in-place update, but could conceivably be done automatically. This proof is sometimes omitted from informal explanations of the algorithm, but the information contained by the definitions of *path* and *pathK* is always communicated. The long term goal illustrated here can be rephrased as follows: *A program verifier should be able to automatically check a program even when its annotations contain no more than what we would say to a reasonably good programmer to explain what the code does and how it works.*

From an engineering perspective, program verifiers are similar to compilers. The input is a program written in a high-level language, and the output is a set of warnings (or errors) that indicate possible bugs. For a good program verifier, the lack of warnings should be a better indicator that the program is correct than any human-made argument. The architecture *often* consists of a front-end that translates a high-level language into a much simpler intermediate language, and a back-end that does the interesting work. The same back-end may be connected to different front-ends, each supporting some high-level language. The back-end is itself split into a VC (**v**erification **c**ondition) generator and an SMT (**s**atisfiability **m**odulo **t**heories) solver. A few alternatives to this architecture are discussed later (Section 1.3).

The VC generator is a trusted component of program verifiers. Therefore it is important to study it carefully, including its less interesting corners. This dissertation shows that even such corners come to life when analyzed in detail from the point of view of correctness and efficiency. The insights gained from such an analysis sometimes lead to simpler and cleaner implementations and sometimes lead to more efficient implementations.

1.2 History

In 1957, 'programming' was not a profession. At least that's what Dijkstra was told [69]. "And, believe it or not, but under the heading *profession* my marriage act shows the ridiculous entry *theoretical physicist*!" That is only one story showing that in those times programmers were second class citizens and did not have many rights. Their predicament was, however, well-deserved: They did not care about the correctness of programs, and they did not even grasp what it *means* for a program to be correct! A few bright people changed the situation. In 1961 McCarthy [132] published the first article concerned with the study of what programs mean. The article is focused on handling recursive functions without side effects, which corresponds to the style of programming used nowadays in pure functional languages like Haskell [106]. Six years later, in 1967, Floyd [80] showed how programs with side effects and arbitrary control flow can be handled formally. He acknowledges that some ideas were inspired by Alan Perlis. He emphasized the view of programs as flowgraphs (or, more precisely, flowcharts). His method is usually known under the name "inductive assertions." The method was popularized by Knuth [111]. In 1969, Hoare [95] introduced an axiomatic method of assigning meanings to programs that appealed more to logicians than to algorithmists, perhaps because no graph was involved. Although the presentation style is very different, Hoare states that "the formal treatment of program execution presented in this paper is clearly derived from Floyd." Based on Hoare's work, Dijkstra [70] introduced in 1975 yet another way of defining the meaning of programs based on predicate transformers such as the weakest precondition transformer. He did so in the context of a language (without **goto**) called "guarded commands," which provided the main inspiration for the Boogie language.

It is revealing that *all* the authors mentioned in the previous paragraph received the Turing Award, although not necessarily for closely related topics:

1966 Alan Perlis: "For his influence in the area of advanced programming techniques and compiler construction."

1971 John McCarthy: "For his major contributions to the field of artificial intelligence."

1972 Edsger W. Dijkstra: "[For] his approach to programming as a high, intellectual challenge; his eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness; and his illuminating perception of problems at the foundations of program design."

1974 Donald E. Knuth: "For his major contributions to the analysis of algorithms and the design of programming languages."

1978 Robert W. Floyd: "For having a clear influence on methodologies for the creation of efficient and reliable software, and for helping to found the following important subfields of computer science: the theory of parsing, the semantics of programming languages, automatic program verification, automatic program synthesis, and analysis of algorithms."

1980 C. Anthony R. Hoare: "For his fundamental contributions to the definition and design of programming languages."

Most researchers now prefer to define programming languages from an operational point of view. Such definitions (1) tend to be more intuitive for programmers and (2) correspond directly to how interpreters are implemented. Plotkin's lecture notes [145] constitute the first coherent and comprehensive account of this approach. Much later, in 2004, Plotkin put together a historical account [146] of how his ideas on structural operational semantics crystallized. He points to alternative ways of handling the **goto** statement. He cites McCarthy [132] as inspiring him to simplify existing work, and credits Smullyan [161] for the $\frac{\text{up}}{\text{down}}$ rules. He also relates operational semantics to denotational semantics [25], yet another formalism that is not in much use today.

All these developments are based on even older work. In the nineteenth century Hilbert advocated a rigorous approach to mathematics: It should be possible in principle to decompose any mathematical proof into a finite sequence of formulas $p_1, p_2, \ldots, p_n$ such that each formula is either an axiom or is obtained from previous formulas by the application of a simple transformation rule. (Such a sequence is a proof of all the formulas it contains.) The set of *axioms* is fixed in advance and is called *theory*. The set of transformation rules is also fixed in advance and is called *calculus*. Without looking at the language used to express formulas, there is not much more that can be said about this process. If the language is propositional logic ($\top$, $\bot$, and variables connected by $\neg$, $\vee$, and $\wedge$), then we can *evaluate* a formula to $\top$ or $\bot$ once a *model*—an assignment of values to variables—is fixed. A formula is *valid* when it evaluates to $\top$ for all models. A calculus is *sound* if it produces only valid formulas starting from valid axioms; a calculus is *complete* if it can produce all the valid formulas. Even if a sound calculus is used, anything can be derived if we start with an *inconsistent* theory. These observations generalize for other languages like fol, hol (**h**igher **o**rder **l**ogic), and lambda calculus. The notion of evaluating formulas is 'operational,' while the calculus feels more like the axiomatic approach to programming languages. The intimate connection between proofs and programs is explored in a tutorial style by Wadler [173].

1.3 Related Work

The previous section gave (intentionally) a very narrow view of modern research on program verification. It is now time to right that wrong, partly. Because the field is so vast, we still do not look at all important subfields. For example, no testing tool inspired by theory is mentioned.

The sharpest divide is perhaps between tools mainly informed by practice and tools mainly informed by theory. The authors of FindBugs [100], PMD [18], and FxCop [10] started by looking at patterns that appear in bad code and then built specific checkers for each of those patterns. Hence, those tools are collections of checkers plus some common functionality that makes it easy to implement and drive such checkers. Crystal [7], Soot [166], and NQuery [16] are stand-alone platforms that make it easy to implement specific checkers. Rutar et al. [159] compare a few static analysis tools for Java, including FindBugs and PMD, from a practical point of view.

Tools informed by theory lag behind in impact, but promise to offer better correctness guarantees in the future. These tools can be roughly classified by the main techniques used by their reasoning core, which determine the language used to describe their input.

Model checking [55] led to successful hardware verifiers like RuleBase [20, 39]. A model checker verifies whether certain (liveness and safety) properties hold for a certain state graph—the 'model.' The properties are written in a temporal logic, such as LTL or CTL; the model is a Kripke structure and is represented usually in some implicit form that tries to avoid state explosion. SPIN [22, 97] and NuSMV [17, 54] are generic model checkers and each has its input language. Hardware model checkers start by transforming a VHDL or Verilog description into a state graph, while software model checkers start by transforming a program written in a language like Java into a state graph. Software model checkers that are clearly under active development include the open source Java Pathfinder [15, 170] and CHESS [5, 136] (for Win32 object code and for CIL). Another noteworthy software model checker is BLAST [3, 42] (for C). Bogor [4, 154] is a framework for developing software model checkers.

The input of theorem provers [155, 156] is a logic formula. Usually, when the language is fol, the theorem prover tries to decide automatically if the input is valid; usually, when the language is hol, the theorem prover waits patiently to be guided through the proof. The former are proof finders, while the later are proof checkers. The distinction is not clear cut: Sometimes, the steps that an interactive prover "checks" are as complicated as some of the theorems that are "proved" automatically. Still, in practice the distinction is important: Automatic theorem provers tend to be fast, while interactive theorem provers tend to be

expressive.

The most widely used hol provers are Coq [6, 41], Isabelle/HOL [12, 13, 139], and PVS [19, 142]. The gist of such provers is that they rely on a very small trusted core. One way to use such theorem provers for program verification is to do everything in their input language. For example, Leroy [126] implemented a compiler for a subset of C in Coq's language, formulated theorems that capture desired properties of a C compiler, and proved them. Since Coq comes with a converter from Coq scripts to OCaml [151] programs, the compiler is executable. (The converter fails if non-constructive laws such as the excluded middle are used.) Another approach is to introduce notation that makes the Coq/HOL/Isabelle script look 'very much' like the program that is being run [130]. Yet another approach is to use hol as the target language of a VC generator (see [45, 35, 167]).

The first approach, that of interactively programming and proving in the same language, is also used with ACL2 [1, 108], whose input language is not higher-order.

Provers that handle only fol do not usually require programmers to interact directly with them. SMT provers [34], like Z3 [65] and CVC3 [33], are designed for program verification. The modules of an SMT solver—a SAT solver and decision procedures for various theories—communicate through equalities. This architecture dates back to Nelson and Oppen [138]. The theories are axiomatizations of things that occur frequently in programs, like arrays, bit vectors, and integers. A specialized decision procedure can handle integers a lot more efficiently than a generic procedure which relies only on function properties, such as the fact that $f(x) = f(y)$ whenever $x = y$. The extra speed comes at the cost of increasing considerably the size of the trusted code base. To have the best of both worlds, speed and reliability, SMT provers may produce proofs that can be later checked by a small program [135]. However, this poses the extra challenge of producing proof pieces from *each* decision procedure and then gluing them together. Simplify [67] is an old prover that has a similar architecture (being co-developed by Nelson), but understands a language slightly different than the standardized SMT language.

Throughout the dissertation, the term 'program verifier' is used usually in a very restricted sense. It refers to a tool that

1. uses an SMT prover as its reasoning core,
2. has a pipeline architecture and an intermediate language, and
3. can be used to generate warnings just like a compiler.

The pipeline architecture with an intermediate language is typical in translators

and compilers [141].

Many tools fit this narrow definition, including ESC/Java [76], the static verifier in Spec$^\sharp$ [31] (for C$^\sharp$), HAVOC [116] (for C), VCC [57] (for C), Krakatoa [75] (for Java), the Jessie plugin in Frama-C [8] (for C), and Jack [35] (for Java). ESC/Java and Jack support JML [117] (the **J**ava **m**odeling **l**anguage), an annotation language for Java that has wide support, a reference manual, and even (preliminary and partial) formal semantics [118]. Frama-C supports ACSL [36], another of the few annotation languages with an adequate reference manual.

The intermediate language, or at least the intermediate representation, used by these tools is much better specified. ESC/Java uses a variant of Dijsktra's guarded commands [70] that was exceptions. Spec$^\sharp$, HAVOC, and VCC, which are developed by Microsoft Research, use the Boogie language. Krakatoa and Frama-C, which are developed by INRIA, use the Why language [75]. The Boogie language (and the associated verifier from Microsoft Research) was used also as a high-level interface to SMT provers in order to verify algorithms and to explore encoding strategies for high-level languages [28, 163].

Many tools have more than one reasoning engine. Jack uses Simplify and Coq. The Why tool uses a wide range of theorem provers: Coq, PVS, Isabelle/HOL, HOL 4, HOL Light, Mizar; Ergo, Simplify, CVC Lite, haRVey, Zenon, Yices, CVC3. SLAM [27] is a Windows driver verifier that uses both a model checker and Z3. Note that, in principle, all SMT provers are interchangeable. The practical issue remaining is that even if the formula language is standardized, the command language is only now being standardized. That is, it is clear how one should write a formula φ, but it is not clear how to ask a prover "is formula φ valid?". The latter is still prover dependent.

The Boogie tool, FreeBoogie, and the Why tool are fairly big pieces of code that convert a program written in an intermediate language into a formula that should be valid if and only if the program is correct. Moore [134] argues that a better solution is to explicitly write the operational semantics, which leads to a much smaller VC generator. It is not clear what impact this has on speed. However, the technique is most intriguing and it would be interesting to pursue it in the context of Boogie and Why. (Moore uses ACL2.) Other tools, like KeY [37] (for Java, using dynamic logic [92]), jStar [71] (for Java, using separation logic [152]), and VeriFast [102] (for C, using separation logic [152]), avoid the VC generation step because they rely on symbolic execution. This means, very roughly, that instead of turning the program into a proof obligation into one giant step, they 'execute' the program and they keep track of the current possible states using formulas. At each execution step they may use the help of a reasoning engine. (Note that at this level of abstraction and hand-

waving there is not much difference between symbolic execution and abstract interpretation [62].)

With so many tools, it is surprising that they do not have a more serious impact in practice. On this subject one can only speculate. It is probably true that a closer collaboration between theoreticians and practitioners would ameliorate the situation. But it is also true that researchers still have much work to do on *known* problems. The expressivity of specification languages is a family of such problems. For example, it is still considered a research challenge to annotate particularly simple patterns, like the Iterator pattern [90] or the Composite pattern [163]. Speed is another problem. As any (non-Eclipse) computer user will tell you, the number of users of a program tends to decrease with its average response time. Since program verifiers are intended to be used similarly with compilers, their response times are naturally compared with the response times of compilers. Right now, program verifiers tend to be slower.

It is, of course, bothersome that in today's state of affairs it is hard to annotate the Iterator pattern properly in many verification methodologies. (Otherwise it would not have been the official challenge of SAVCBS 2006.) But we should not expect practitioners to annotate such things *at all*, just as we should not expect them to state the invariant $0 \leq i \leq n$ on the omnipresent **for** loop. In general, usable verification tools should embody knowledge that is common to people in a specific domain. There is some work on tackling typical exercises in an introductory programming course [123] and there is also work in encoding traditional mathematical knowledge in verifiers [49].

A choice that may have seemed unusual is the use of the name 'Roy–Warshall,' instead of the more standard 'Floyd–Warshall,' for the algorithm in Figure 1.1. There are many algorithms with the shape

$$\textbf{for } k \textbf{ do for } i, j \textbf{ do } a_{ij} \leftarrow a_{ij} \circ (a_{ik} \bullet a_{kj}) \tag{1.1}$$

and they are sometimes known under the name "the *kij* algorithm." Roy [157] and Warshall [175] noticed independently that $\circ = \vee$ and $\bullet = \wedge$ solves the transitive closure problem. Floyd [79] noticed that $\circ = \min$ and $\bullet = +$ solves the all pairs shortest paths problem. (He did this five years before assigning meanings to programs.) Although less clear, other earlier algorithms are instantiations of the *kij* schema. Kleene [110] proved that every finite automaton corresponds to a regular expression. His proof is constructive and is now known as Kleene's algorithm. The Gauss–Jordan method for solving systems of linear equations is another example. Pratt [148] discusses the *kij* algorithm in general terms.

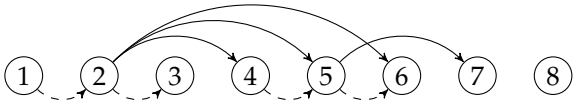


Figure 1.2: Dependencies between chapters

1.4 A Guided Tour

The content of this dissertation touches on issues related to software engineering, compilers, programming languages, algorithm analysis, and theorem provers. It is often abstract and theoretical, but sometimes descends into implementation details. Therefore, there are plenty of opportunities for a reader to find something enjoyable. Of course, there are even more opportunities for a reader to find something not enjoyable, especially if the reader has a strong preference for one sub-field of computer science. Fortunately, the dissertation does not stem from one big contribution, but rather from many smaller ones that are related and yet can be understood independently. The dependencies between chapters appear in Figure 1.2. A solid arrow signifies a strong dependency and a dashed one signifies a weak dependency. Chapter 2 is very important.

The scope of the dissertation reflects the broad interests of the author. Often, broad texts say nothing about a lot, and are quite boring. The focus on a very specific part of a very specific type of static analysis tools and the focus on a very small subset of the Boogie language are meant to ensure enough depth.

My hope is that the dissertation will bring together at least two people from different sub-fields of computer science to work together on a common problem related to program verifiers.

Chapter 2 presents the syntax and the operational semantics of a subset of the Boogie language, which is used throughout subsequent chapters.

Chapter 3 presents *what* FreeBoogie does. Its architecture is sketched, including the interfaces between major components. The advantages and disadvantages of each important design decision are discussed, so that others who endeavor in similar tasks avail of our experience and avoid repeating our mistakes.

Chapter 4 presents *how* FreeBoogie replaces assignments by equivalent assumptions. The algorithm's complexity is analyzed in detail and its goals are clearly defined. A natural variant of the problem is proved to be NP-hard.

Chapter 5 presents *how* FreeBoogie generates a prover query from a Boogie program, using either a weakest precondition transformer or a strongest postcondition transformer. In the process, we see how four types of assigning meanings to Boogie programs relate to each other: operational semantics, Hoare

triples, weakest preconditions, and strongest postconditions. (Chapters 4 and 5 are based on [87].)

Chapter 6 presents *how* FreeBoogie exploits the incremental nature of writing code and annotations in order to improve its response times. The generic idea, of exploiting what is known from previous runs of the verifier, is refined and then proved correct. (This chapter is based on [88].)

Chapter 7 presents *what* FreeBoogie does to protect developers from making silly mistakes that render verification useless: Inconsistencies make everything provable. The algorithm explaining *how* inconsistencies are ruled out efficiently is analyzed in detail, and experimental results are given. (This chapter is based on [104].)

Chapter 8 concludes the dissertation. Appendix A summarizes the notation used throughout the dissertation. The reader is advised to browse that appendix before continuing with the next chapter.

Software engineers and practitioners are likely to enjoy most Chapter 3. People with a formal background, like logicians and type theorists, are likely to enjoy most Chapters 5 and 6. Algorithmists are likely to enjoy most Chapter 4, 6 and 7. Developers of program verification tools are likely to enjoy most Chapters 6 and 7.

Chapter 2

The Core Boogie

"The boogie bass is defined as a left-hand rhythmic pattern, developed from boogie-woogie piano styles, that is played on the bass string."
— Frederick M. Noad [140]

From now on all programs are written in the Boogie language, and almost all are written in a subset—the core Boogie—defined in this section. What is 'core' is relative to the topics discussed in this dissertation.

Figure 2.1 shows an implementation of sequential search. After the counter i is initialized in line 2 the control goes nondeterministically to both labels b and d. An execution gets *stuck* if it hits an assumption that does not hold. Since the conditions on the lines 3 and 5 are complementary, exactly one of the two executions will continue. (In general, the **if** statement of full Boogie may be desugared into a **goto** statement that targets two **assume** statements with complementary conditions.) The **return** statement is reached only if $i \geq vl$ or $v[i] = u$.

The high-level constructs of full Boogie (such as **if** statements, **while** statements, and **call** statements) can always be desugared into the core that is formalized here. The concrete grammar of core statements appears in Figure 2.2. The

```
1  procedure indexOf(u : int, v : [int] int, vl : int) returns (i : int) {
2    a: i := 0; goto b, d;
3    b: assume i < vl && v[i] != u;
4    c: i := i + 1; goto b, d;
5    d: assume !(i < vl && v[i] != u); return;
6  }
```

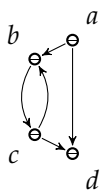


Figure 2.1: Boogie program with one loop and its flowgraph

statement → label? (assignment | assumption | assertion | jump) **;**
label → *id* **:**
assignment → id **:=** expression
assumption → **assume** expression
assertion → **assert** expression
jump → (**goto** id-tuple) | **return**
id-tuple → *id* (**,** *id*)*

Figure 2.2: Syntax of core Boogie statements

type → primitive-type | map-type
primitive-type → **int** | **bool**
map-type → **[** primitive-type **]** type
expression → **(** quantifier *id* **:** type **::** expression **)**
expression → unary-operator expression | expression binary-operator expression
expression → expression **[** expression **]**
expression → *id* | literal | **(** expression **)**
quantifier → **forall** | **exists**
unary-operator → **!** | **-**
binary-operator → **+** | **-** | < | **==** | **&&** | ||
literal → **true** | **false** | **0** | **1** | **2** | ...

Figure 2.3: Syntax of core Boogie types and expressions

statement **assert** p means "when you reach this point, check that p holds." The statement **assume** p means "continue to look only at executions that satisfy p."

The type system of full Boogie is rich, featuring polymorphic maps, bit vectors, and user-defined types, among others. Its expression language is similarly rich. Unlike in the case of statements, the VC generator implementation (henceforth known as FreeBoogie) does not desugar types and expressions into simpler ones. How they are treated, however, is not novel. For the sake of the presentation, only a few types and expressions are retained in the core, the ones in Figure 2.3.

The syntax for the overall structure of core Boogie programs appears in Figure 2.4. Statements are preceded by variable declarations, in which the variables representing the input and the output are singled out. The keyword **procedure** is retained from full Boogie, where a program may contain more than one procedure and where there is also a **call** statement. Procedures are not included in the core, because there is nothing novel related to procedure calls in this dissertation. The mandatory **return** statement at the end of each body reduces the number of special cases that need to be discussed later. FreeBoogie inserts such a statement automatically during parsing, so the user does not have to end all procedures with **return**.

Typechecking core Boogie is straightforward: Figure 2.5 shows a represen-

program → **procedure** *id* **(** arguments? **) returns (** results? **)** body
arguments, results → *id* **:** type (**,** *id* **:** type)*
body → variable-declaration* statement* **return ;**
variable → **var** *id* **:** type **;**

Figure 2.4: Structure of a (core) Boogie program

$$\frac{\forall\, 1 \le k \le n,\ \exists t,\ (\vdash v_k : t \wedge \vdash e_k : t)}{\vdash v_1, \ldots, v_n := e_1, \ldots, e_n}\ [\text{asgn}] \qquad \frac{\vdash e : \mathbf{bool}}{\vdash \mathbf{assert}\ e}\ [\text{asrt}]$$

$$\frac{\vdash e : \mathbf{bool} \quad \vdash f : \mathbf{bool}}{\vdash e\ \&\&\ f : \mathbf{bool}}\ [\text{bool}] \qquad \frac{\vdash e : \mathbf{int} \quad \vdash f : \mathbf{int}}{\vdash e + f : \mathbf{int}}\ [\text{arith}]$$

$$\frac{\vdash e : \mathbf{int} \quad \vdash f : \mathbf{int}}{\vdash e < f : \mathbf{bool}}\ [\text{comp}] \qquad \frac{\exists t, \vdash e : t \wedge \vdash f : t}{\vdash e == f : \mathbf{bool}}\ [\text{eq}]$$

$$\frac{}{\vdash \mathbf{true} : \mathbf{bool}}\ [\text{lit-bool}] \qquad \frac{}{\vdash \mathbf{0} : \mathbf{int}}\ [\text{lit-int}]$$

Figure 2.5: Typing rules for core Boogie

tative sample of the typing rules. The judgment $\vdash p$ means that the program fragment p is well-typed and the judgment $\vdash p : t$ means that the program fragment p is well-typed *and* has the type t. The customary environment is omitted because it is fixed—it consists of all the variable typings appearing at the beginning of the program. The rules that are missing are similar to the ones given: For example, there is a rule that says that the expression appearing in an assumption must have the type **bool** (analogous to rule [asrt]).

Operational Semantics Although it has not been done before, it is not hard to give an operational semantics for (core) Boogie. The set of all variables is denoted by *Variable*. It can be thought of as the set of all identifiers or as the set of all strings. The set of all values is denoted by *Value* and it contains the set of booleans $\mathbb{B} = \{\top, \bot\}$. *Stores* (usually denoted by σ) assign values to variables.

$$\mathit{Store} = \mathit{Variable} \to \mathit{Value} \tag{2.1}$$

$$\sigma \in \mathit{Store} \tag{2.2}$$

Expressions assign values to a stores. Boolean expressions, called *predicates* and usually denoted by p, q, r, define sets of stores.

$$\mathit{Expression} = \mathit{Store} \to \mathit{Value} \tag{2.3}$$

$$\mathit{Predicate} = \mathit{Store} \to \mathbb{B} \tag{2.4}$$

$$p, q, r \in \mathit{Predicate} \tag{2.5}$$

According to the syntax, the program is a list of statements, which means that we can assign counters 0, 1, 2 ... to them. To simplify the presentation, we will assume that all labels are counters (in the proper range). The *state* of a Boogie program is either the special *error* state or a pair $\langle \sigma, c \rangle$ of a store σ and a counter c of the statement about to be executed. The following rules define

a relation $\rightsquigarrow$ on states, thus giving an operational semantics for the core of the Boogie language.

$$\frac{p\ \sigma}{\langle \sigma, c : (\textbf{assume}/\textbf{assert}\ p)\rangle \rightsquigarrow \langle \sigma, c+1\rangle} \tag{2.6}$$

$$\frac{\neg(p\ \sigma)}{\langle \sigma, c : (\textbf{assert}\ p)\rangle \rightsquigarrow \mathit{error}} \tag{2.7}$$

$$\frac{}{\langle \sigma, c : (v{:=}e)\rangle \rightsquigarrow \langle (v \leftarrow e)\ \sigma, c+1\rangle} \tag{2.8}$$

$$\frac{c' \in \ell}{\langle \sigma, c : (\textbf{goto}\ \ell)\rangle \rightsquigarrow \langle \sigma, c'\rangle} \tag{2.9}$$

A rule $\frac{h}{\langle \sigma, c : \mathscr{P}\rangle \rightsquigarrow s}$ means that the program may evolve from state $\langle \sigma, c\rangle$ to the state s if the hypothesis h holds and if the counter c corresponds to a statement that matches the pattern $\mathscr{P}$. For example, rule (2.9) says that $\langle \sigma, c\rangle$ may evolve into $\langle \sigma, c'\rangle$ if the statement at counter c matches the pattern **goto** ℓ and $c' \in \ell$. The notation $p\ \sigma$ stands for the application of function p to the argument σ. Space (that is, the function application operator) is left associative, has the highest precedence, and, as is customary, it is omitted if there is only one parenthesized argument that is not followed by another function application. The notation $(v \leftarrow e)$ used in rule (2.8) stands for a store transformer, defined as follows.

$$(v \leftarrow e) : \mathit{Store} \rightarrow \mathit{Store} \tag{2.10}$$

$$(v \leftarrow e)\ \sigma\ w = \begin{cases} \sigma\ w & \text{if } v \neq w \\ e\ \sigma & \text{if } v = w \end{cases} \tag{2.11}$$

Definition 1. An *execution* of a core Boogie program is a sequence $s_0, s_1, \ldots, s_n$ of states such that $s_{k-1} \rightsquigarrow s_k$ for all $k \in 1..n$ and $s_0 = \langle \sigma_0, 0\rangle$ for some arbitrary *initial store* σ_0.

The sources of nondeterminism in core Boogie are (1) the initial store σ_0 and (2) the **goto** rule (2.9) which allows multiple successors.

We say that an execution $s_0, s_1, \ldots, s_n$ *goes wrong* if $s_n = \mathit{error}$. This can happen only if the last statement that was executed was an assertion. We say that that assertion was *violated*.

Definition 2. A core Boogie program is *correct* if none of its executions goes wrong.

Boogie does not facilitate reasoning about termination. Throughout the

dissertation the term "correct" will usually mean what is traditionally referred to as "partially correct." Because we are not interested in termination, we do not distinguish between executions that reach an assumption that is not satisfied and executions that reach a **return** statement: both get stuck.

Let us look at an example. If we turn the labels of the program in Figure 2.1 (on page 11) into counters we obtain

```
0: i := 0;
1: goto 2, 5;
2: assume i < vl && v[i] != u;
3: i := i + 1;
4: goto 2, 5;
5: assume !(i < vl && v[i] != u);
6: return;
```

for which one possible execution is

$$\langle \sigma, 0 \rangle \tag{2.12}$$
$$\langle (i \leftarrow 0)\ \sigma, 1 \rangle \tag{2.13}$$
$$\langle (i \leftarrow 0)\ \sigma, 2 \rangle \tag{2.14}$$
$$\big(i < vl \wedge v[i] \neq u\big)\ \big((i \leftarrow 0)\ \sigma\big) \qquad (*) \tag{2.15}$$
$$\langle (i \leftarrow 0)\ \sigma, 3 \rangle \tag{2.16}$$
$$\langle (i \leftarrow i+1)\ ((i \leftarrow 0)\ \sigma), 4 \rangle \tag{2.17}$$
$$\langle (i \leftarrow i+1)\ ((i \leftarrow 0)\ \sigma), 5 \rangle \tag{2.18}$$
$$\neg\big(i < vl \wedge v[i] \neq u\big)\ \big((i \leftarrow i+1)\ ((i \leftarrow 0)\ \sigma)\big) \qquad (*) \tag{2.19}$$
$$\langle (i \leftarrow i+1)\ ((i \leftarrow 0)\ \sigma), 6 \rangle \tag{2.20}$$

(The store σ is arbitrary but fixed.) The lines marked with $(*)$ are not states but rather conditions that are assumed to hold. In order to evaluate those conditions we need to look inside the predicates. Every $n$ary function $f : (\mathit{Value} \rightarrow)^n \mathit{Value}$ has a corresponding function f' on expressions:

$$\begin{aligned} f' &: (\mathit{Expression} \rightarrow)^n \mathit{Expression} \\ f'\ e_1 \ldots e_n\ \sigma &= f\ (e_1\ \sigma) \ldots (e_n\ \sigma) \end{aligned} \tag{2.21}$$

In particular, boolean functions have corresponding predicate combinators. By notation abuse, we write $\wedge$, $\vee$, ... between booleans as well as between predicates. Also, the boolean constants $\top$ and $\bot$ are boolean functions with arity 0, so we shall abuse them too and write $\top$ and $\bot$ for constant predicates. Similarly, we will lift the other operators. An example evaluation of a predicate

follows.

$$\begin{aligned} & \big(i < vl \wedge v[i] \neq u\big)\ \big((i \leftarrow 0)\ \sigma\big) && (2.22)\\ = \quad & (i < vl)\ \big((i \leftarrow 0)\ \sigma\big) \wedge (v[i] \neq u)\ \big((i \leftarrow 0)\ \sigma\big) && (2.23)\\ = \quad & i\ \big((i \leftarrow 0)\ \sigma\big) < vl\ \big((i \leftarrow 0)\ \sigma\big) \wedge \ldots && (2.24) \end{aligned}$$

A predicate that consists of a variable v is evaluated by reading the variable's value from the store; a predicate that consists of a constant c evaluates to that (lifted) constant. (For example, the predicate 0 is lifted to the integer 0.)

$$v\ \sigma = \sigma\ v \tag{2.25}$$

$$c\ \sigma = c \tag{2.26}$$

The evaluation continues as follows.

$$\begin{aligned} & i\ \big((i \leftarrow 0)\ \sigma\big) < vl\ \big((i \leftarrow 0)\ \sigma\big) && (2.27)\\ = \quad & (i \leftarrow 0)\ \sigma\ i < (i \leftarrow 0)\ \sigma\ vl && (2.28)\\ = \quad & 0 < \sigma\ vl && (2.29) \end{aligned}$$

And we conclude that for the previous example execution the initial store σ must satisfy $0 < (\sigma\ vl)$.

A special case of (2.21) is equality. Equality between values is a function $= : \mathit{Value} \to \mathit{Value} \to \mathbb{B}$ that has a corresponding expression combinator $= : \mathit{Expression} \to \mathit{Expression} \to \mathit{Predicate}$. For example, we write $v = e$ for a predicate that defines the set of stores in which the variable v and the expression e evaluate to the same value.

We say that predicate p is *valid* and we write $|p|$ when it holds for all stores.

$$|p| = (\forall \sigma,\ p\ \sigma) \tag{2.30}$$

We will often need to say that two predicates delimit exactly the same set of stores, so we introduce a shorthand notation for it, which will also be useful when defining (syntactically) new predicates.

$$(p \equiv q) = |p = q| \tag{2.31}$$

Finally, we shall abuse notation and write $(v \leftarrow e)\ p$ for predicates p, based on the fact that every store transformer $f : \mathit{Store} \to \mathit{Store}$ has a corresponding

expression transformer f':

$$f' : \mathit{Expression} \to \mathit{Expression} \tag{2.32}$$

$$f'\ e\ \sigma = e\ (f\ \sigma) \tag{2.33}$$

This concludes the presentation of the core of Boogie that is used often in the next chapters. The full Boogie language is more high-level and more user-friendly [119, 124]. For example, the full Boogie language allows the use of *uninterpreted* mathematical functions. (They are called uninterpreted to distinguish them from interpreted functions like integer addition, which have a special meaning from the point of view of Boogie's semantics. All that is known about an uninterpreted function f is that $f(x) = f(y)$ follows from $x = y$, and other properties that are given explicitly in the Boogie program.)

Chapter 3

Design Overview

"One of the best indications that a program is the result of the activity of design is the existence of a document describing that design."
— Jim Waldo [174]

This chapter is rather dense. A cursory read will show how the later, more theoretical chapters fit together in the context of a real program; a careful read will serve as a guide to FreeBoogie's source code for new developers.

3.1 An Example Run

The best way to understand how FreeBoogie works is to run it on a few examples and ask it to dump its data structures at intermediate stages. Figure 3.1 shows a Boogie program suitable for a first run. Notice that the language is not restricted to the core defined in Chapter 2. To peek at FreeBoogie's internals use the command

```
fb --dump-intermediate-stages=log example.bpl
```

assuming that you wrote the content of Figure 2.1 in the file `example.bpl` and that FreeBoogie is correctly installed on your system. This will create a directory `log`. The output of each processing phase of FreeBoogie appears in a

```
1  type T;
2  procedure indexOf(x : T, a : [int] T, al : int) returns (i : int) {
3    i := 0;
4    while (i < al && a[i] != x) { i := i + 1; }
5  }
```

Figure 3.1: A high-level Boogie version of Figure 2.1

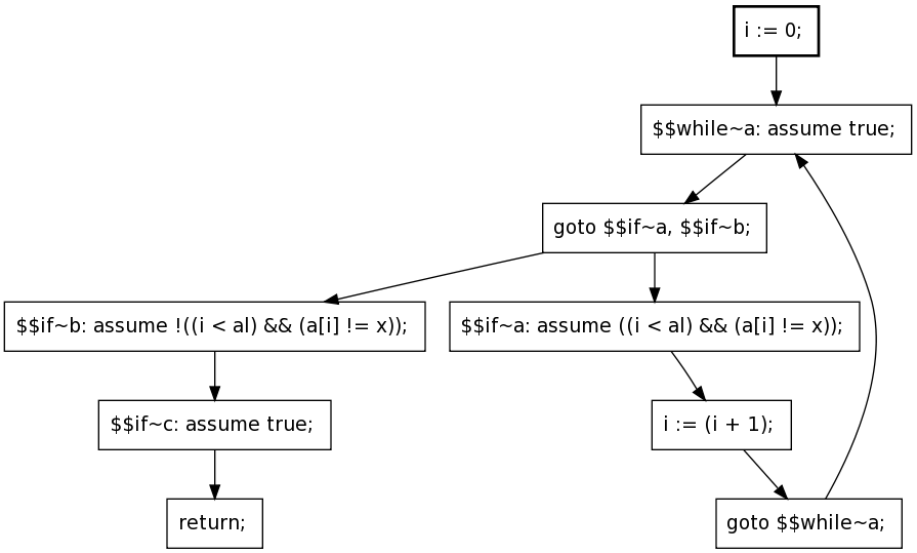


Figure 3.2: Flowgraph of a desugared version of Figure 2.1

subdirectory of `log`. The name of the subdirectory is the name of the phase.

Such transformation phases include desugaring the **while** statement, desugaring the **if** statement, cutting cycles, eliminating assignments (Chapter 4), computing the VC (Chapter 5). Let us look briefly at the state of FreeBoogie after **while** and **if** statements are desugared. To do so we could look at the pretty-printed Boogie code but we can also look at the flowgraph that is dumped by FreeBoogie in the GraphViz [72] format. The flowgraph is rendered in Figure 3.2. Such representations of the internal data structures are very helpful in understanding FreeBoogie and in debugging it.

The flowgraph is an example of *auxiliary* information that FreeBoogie computes after each transformation. The other pieces of auxiliary information are the symbol table and the types. The *symbol table* is a one-to-many bidirectional map between identifier definitions and identifier uses. The *types* are associated with expressions (and subexpressions).

To see the query that is sent to the theorem prover you must run a different command, this time shown in abbreviated form:

```
fb -lf=example.log -ll=info -lc=prover example.bpl
```

The log file `example.log` will contain everything sent to the prover. The result of the whole run is

```
OK: indexOf at example.bpl:2:11
```

indicating that the program is correct.

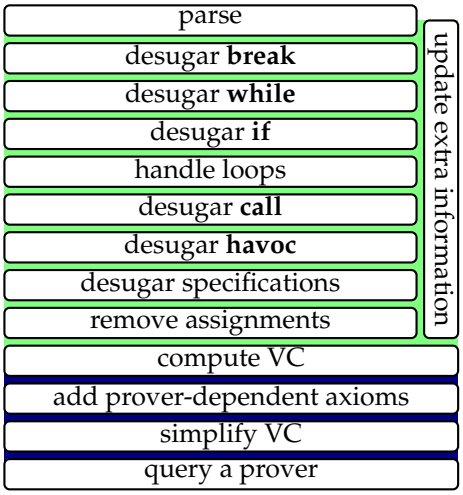


Figure 3.3: FreeBoogie architecture.

3.2 Pipeline

Figure 3.3 shows that FreeBoogie has a pipeline architecture. The light green color stands for the Boogie AST (**a**bstract **s**yntax **t**ree); the dark blue color stands for the SMT AST.

Horizontal boxes, except for the first one (parse) and the last one (query a prover), represent *transformations*. Depending on the type of input and on the type of output there are three types of transformations: Boogie to Boogie, Boogie to SMT, and SMT to SMT. For brevity, we say 'Boogie transformations' instead of 'Boogie to Boogie transformations', and 'SMT transformations' instead of 'SMT to SMT transformations'. All these transformations are designed not to miss bugs, at the cost of possible false positives.

Definition 3. A Boogie transformation is *sound* when it produces only incorrect Boogie programs from incorrect Boogie programs. A Boogie to SMT transformation is *sound* when it produces only invalid formulas from incorrect Boogie programs. An SMT transformation is *sound* when it produces only invalid formulas from invalid formulas.

Remark 1. This definition is in a way formal, but in a way it is not. It makes use of the concept of 'correct Boogie program' and we only have semantics for *core* Boogie programs (Chapter 2). For example, we can say precisely what it means for the assignment removal transformation to be sound, because both the input and the output of that transformation are core Boogie programs; however, we can only informally describe the preceding transformations as sound.

The symmetric notion is that of completeness.

Definition 4. A Boogie transformation is *complete* when it produces only correct Boogie programs from correct Boogie programs. A Boogie to SMT transformation is *complete* when it produces only valid formulas from correct Boogie programs. An SMT transformation is *complete* when it produces only valid formulas from valid formulas.

All transformations in FreeBoogie are sound; all transformations in FreeBoogie, except loop handling,

Full Boogie would not be user friendly without high-level constructs like **while** statements and **break** statements. Many phases in FreeBoogie perform syntactic desugarings of these constructs. The desugaring is sometimes local, in the sense that it can be done without keeping track of an environment, and sometimes it is not. For example, to desugar the **break** statement we must keep track of the enclosing **while** and **if** statements; but the desugaring of a **havoc** statement does not depend on any surrounding code.

The most important transformation in FreeBoogie is the transition from Boogie to SMT. The role of preceding Boogie transformations is to simplify the program to a form on which the VC is easily computed; the role of subsequent SMT transformations is to bring the VC to a form that is easily handled by a prover.

The order of Boogie transformations depends on constraints such as the following. The Boogie to SMT transformation ('compute VC' in Figure 3.3) only handles the **assert**, **assume**, and **goto** statements. The Boogie transformation that removes assignments only handles acyclic flowgraphs. Hence, the flowgraph must first be transformed into an acyclic one ('handle loops' in Figure 3.3).

The VC uses concepts such as arrays, which may or may not be known to the prover. In the latter case, axioms that describe the concept must be added to the VC. Finally, the VC is simplified so that the communication with the prover is more efficient.

The source code of FreeBoogie, written in Java 6, contains four packages, which are in one-to-one correspondence with the following sections.

- `freeboogie.ast`: data structures for representing Boogie programs and facilities for processing such data structures
- `freeboogie.tc`: code that computes auxiliary information from a Boogie AST
- `freeboogie.vcgen`: the Boogie transformations and the Boogie to SMT transformation

```
Program = Signature! sig, Body! body;
Signature = String! name, [list] VariableDecl args, [list] VariableDecl results;
Body = [list] VariableDecl vars, Block! block;
VariableDecl = String! name, Type! type;
Block = [list] Statement statements;
Type = enum(Ptype: BOOL, INT) ptype,
Statement :> AssertAssumeStmt, AssignmentStmt, GotoStmt;
AssertAssumeStmt = enum(StmtType: ASSERT, ASSUME) type,
  [list] Identifier typeArgs, Expr! expr;
AssignmentStmt = [list] OneAssignment assignments;
GotoStmt = [list] String successors;
Identifier = String! id,
OneAssignment = Identifier lhs, Expr rhs;
```

Figure 3.4: The abstract grammar of core Boogie

- `freeboogie.backend`: SMT transformations, data structures for representing SMT, and code for interfacing with the prover

3.3 The Abstract Syntax Tree and its Visitors

The Boogie AST data structures are described using a compact notation. A subset, corresponding to core Boogie, appears in Figure 3.4. AstGen (a helper tool) reads this description and a code template to produce Java classes. The approach has advantages and disadvantages. The generated classes are very similar to each other because they come from the same template. This means that it is easy to learn their interface. It also means that it is easier to change all the classes in a consistent way by changing the template. The compact description in Figure 3.4 is easier to read than the corresponding 12 Java classes. The overall structure of the AST is easier to grasp. It is also easier to modify, since it takes far less time to change one or two lines than one or two Java classes. However, the programmer needs to learn a new language (the one used in Figure 3.4) and IDEs are usually confused by code generators.

Another consequence of this approach, which might be seen as a disadvantage, is that there is no way to add specific code to specific classes: We are forced to implement operations over the AST using the visitor pattern [82].

3.3.1 The Abstract Grammar Language

AstGen reads a description of an abstract grammar and a template. Therefore it understands two languages—the AstGen abstract grammar language and the AstGen template language. This section describes the AstGen abstract grammar language, which was already used in Figure 3.4. The syntax of the AstGen abstract grammar language appears in Figure 3.5. (Note that a formal language

```
grammar →     rule*
rule →        composition | inheritance | specification
composition →  class = members? ;
inheritance →  class :> classes? ;
specification → class : text ¶
member →      tags type !? name
tag →         [ id ]
type →        id | enum ( id : ids? )
members →     member (, members)*
tags →        tag (, tags)*
name, class → id
```

Figure 3.5: The syntax of the abstract grammar language

is used to describe the concrete syntax of a language that is used to describe the abstract grammar of a language whose concrete syntax was studied in Chapter 2 using the same formal language that we use here in Figure 3.5: There is some opportunity for confusion.)

The abstract grammar is described by a list of rules. Each rule starts with the name of the class to which it pertains. A composition rule continues with an equal sign (=) and a list of members. An inheritance rule continues with a supertype sign (:>) and a list of subclasses. A specification rule continues with a colon (:) and some arbitrary text. The end of composition and inheritance rules is marked by a semicolon (;) and the end of a specification rule is marked by the end of the line (depicted as ¶ in Figure 3.5). That is (almost) all.

To illustrate why this notation is beneficial, suppose that initially the data structures for the Boogie AST contained only public fields.

```
1 public class Program {
2   public Signature sig;
3   public Body body;
4 }
```

This Java code is obviously not much longer and indeed very similar to the first line in Figure 3.4. But it has a number or problems. First, we probably want the *Program* class to be final. Without using AstGen we must go and add the keyword **final** in each class: *Program*, *Signature*, *Body*, *VariableDecl*, ... With AstGen, we only need to add that keyword in the template. Another problem is that there is no constructor. Again, adding constructors is a repetitive job if we must do it in each and every class. Finally, FreeBoogie's data structures are immutable. More precisely, the members are private and final, they are set by the constructor, and accessor methods only allow them to be read. Again, making these changes in all classes is a repetitive job. In summary, the main advantage of using AstGen is that we separate the concern of defining the shape of the abstract grammar from lower-level concerns such as whether we allow subclassing or not, whether we allow mutations or not.

Specification rules, which refer to classes, and tags, which refer to members (see Figure 3.5), allow for a little non-uniformity between generated classes.

The bang sign (!) is a shorthand for the tag [nonnull]. As we will see, the code template may contain parts that are used or not by AstGen depending on whether a tag is present. In particular, FreeBoogie's code template asserts nonnullness only for members with the tag [nonnull]. The only other tag used in Figure 3.4 is [list], which will be discussed briefly in Section 3.3.4.

A specification rule associates some arbitrary text to a class. Templates then instruct AstGen where in the output to insert the arbitrary text. For example, in FreeBoogie the arbitrary text is always a side-effect free Java boolean expression. FreeBoogie's code template inserts these boolean expressions in Java **assert** statements within constructors. In general, the intended use of specification rules is to give object or class invariants. However, there is nothing in AstGen to enforce this use. Hence, specification rules could be abused to insert, say, custom comments in the header of generated classes.

3.3.2 AstGen Templates

Figure 3.6 illustrates the main characteristics of an AstGen template and, at the same time, gives some details on how the Boogie AST data structures are implemented. The language for templates is influenced by TEX [114]. Macros start with a backslash (\) and may take arguments. Some macros are primitive and some are defined using `\def`. Before cataloging primitive macros, let us analyze the high-level structure of the template in Figure 3.6.

The first four lines define macros that are used later. AstGen then sees the (primitive) `\classes` macro and processes its argument once for each class in the abstract grammar. Terminal classes, those that have no subclass declared in the abstract grammar, have private fields, a private constructor that initializes those fields, a static factory method *mk*, a method *checkInvariant*, accessors for getting the values of the fields, and a method *eval*, which is typically called *accept* in most presentations of the visitor pattern. Non-terminal classes only have abstract accessors for getting the values of the fields.

The primitive macros can be grouped in four categories: (1) output selection, (2) data, (3) test, and (4) iteration.

The macro `\file`$\{f\}$ globally directs the output from now on to the file *f*.

The data macros do not take any parameter. They expand to the name of the current class (`\className`), the name of the base class of the current class (`\baseName`), the type of the current member (`\memberType`), the name of the current member (`\memberName`), the name of the current enumeration (`\enumName`), the current enumeration value (`\valueName`), the current invariant (`\inv`). The 'current' class/member/enumeration/value/invariant is determined by the enclosing iteration macros. All data macros except `\inv` are made of two words

```
\def{smt}{\if_primitive{\Membertype}{\MemberType}}
\def{mt}{\if_tagged{list}{ImmutableList<}{}\smt\if_tagged{list}{>}{}}
\def{mtn}{\mt \memberName}
\def{mtn_list}{\members[,]{\mtn}}

\classes{\file{\ClassName.java}
/* ... package specification and some imports ... */
public \if_terminal{final}{abstract} class \ClassName extends \BaseName {
\if_terminal{
  \members{private final \mtn;}
  private \ClassName(\mtn_list) {
    \members{this.\memberName = \memberName;}
    checkInvariant();
  }
  public static \ClassName mk(\mtn_list) {
    return new \ClassName(\members[,]{\memberName});
  }
  public void checkInvariant() {
    assert location != null;
    \members{\if_tagged{nonnull|list}{assert \memberName != null;}{}}
    \invariants{assert \inv;}
  }
  \members{public \mtn() { return \memberName; }}
  @Override public <R> R eval(Evaluator<R> evaluator) {
    return evaluator.eval(this);
  }
}{
  \selfmembers{public abstract \mtn();}
}} }
```

Figure 3.6: Excerpt from the AstGen template for Boogie AST classes

and they come in four case conventions (camelCase, PascalCase, lower_case, and UPPER_CASE): The output is formatted accordingly.

The test macros have the shape `\if`*condition*{*yes*}{*no*}. If the condition holds then the *yes* part is processed and the *no* part is ignored; if the condition does not hold then the *no* part is processed and the *yes* part is ignored. The braces in the *yes* and *no* parts must be balanced. The condition `_primitive` holds when the type of the current member does not appear on the left hand side of a composition rule. (In particular, it holds for members whose type is an enumeration.) The condition `_enum` holds when the type of the current member is an enumeration. The condition `_terminal` holds when the current class has no subclass. The condition `_tagged`{*tag_expression*} is more interesting. A tag expression may contain tag names, logical-and (`&`), logical-or (`|`), and parentheses. A tag name evaluates to $\top$ when the current member has that tag.

The iteration macros have the shape *macro*[*separator*]{*argument*}. The argument is processed repeatedly and the optional separator is copied to the output between two passes over the argument. The macro `\classes` processes its argument for each class (and hence each pass has a 'current class'). The macro `\members` processes its argument for each member of the current class, including inherited members (and hence each pass has a 'current member'). The macro `\selfmembers` processes its argument for each member of the current class, excluding inherited members (and hence each pass has a 'current member'). The macro `\invariants` processes its argument for each invariant of the current class, which appear in specification rules (and hence each pass has a 'current invariant'). The macro `\enums` processes its argument for each enumeration used as a type in the current class (and hence each pass has a 'current enumeration'). The macro `\values` processes its argument for each value of the current enumeration (and hence each pass has a 'current enumeration value').

It is an error for a macro x to appear in a context where there should be a current y, but there is none. For example, it is an error for the macro `\enumName` to appear in a context where there is no current enumeration. In other words, the macro `\enumName` cannot appear outside of the argument of `\enums`, which is the only macro that sets a current enumeration.

3.3.3 Visitors

The visitor pattern is widely used to implement compilers. It can be seen as a workaround to a limitation of most object-oriented languages. A reference u has the static type C_u when the declaration of variable u is C_u u; a reference u has the dynamic type C'_u when it points to an object whose type is C'_u; an object has the type C'_u when it was created by the statement **new** $C'_u(\cdots)$. The method call

$u.m(v)$ is resolved based on the dynamic type C'_u and on the static type C_v. In other words, the code that will be executed is in a method named m that takes an argument of type C_v (or a supertype of C_v) and is defined in the class C'_u (or a supertype of C'_u). There is no way to do the dispatch based on the dynamic type of two (or more) references.

However, it is possible to do the dispatch based on the dynamic types of n references *one by one*, at the cost of writing extra code. Say the references $u_1, u_2, \ldots, u_n$ have static types $C_1, C_2, \ldots, C_n$ and dynamic types $C'_1, C'_2, \ldots, C'_n$. The initial call $u_1.u_1(u_2,\ldots,u_n)$ will execute a method $m_1(C_2, \ldots, C_n)$ from the class C'_1, because all the (proper) subclasses of C_1 implement such a method. The body of all these methods will be identical: It will contain the call $u_2.m_2(\textbf{this}, u_3, \ldots, u_n)$. Each subclass of C_2, including C'_2, is expected to have a *set* of methods $m_2(\mathscr{C}_1, C_3, C_4, \ldots, C_n)$ for all possible (proper) subclasses $\mathscr{C}_1$ of C_1. The static type of **this** in the call $u_2.m_2(\textbf{this}, u_3, \ldots, u_n)$ was C'_1, so the method with $\mathscr{C}_1 = C'_1$ will be chosen out of the whole set. In general, all subclasses of C_k must implement a set of methods $m_k.(\mathscr{C}_1, \ldots, \mathscr{C}_{k-1}, C_{k+1}, \ldots, C_n)$, for all subclasses $\mathscr{C}_1$ of C_1, all subclasses $\mathscr{C}_2$ of C_2, and so on. The methods m_n will do the actual work. Let us estimate the number of methods that only forward calls and were referred to in the beginning of the paragraph as 'extra code'. If the number of possible types for $u_1, u_2, \ldots, u_n$ is, respectively, $w_1, w_2, \ldots, w_n$ then the number of methods m_k is $\prod_{i \leq k} w_i$. There are therefore $\sum_{k<n} \prod_{i \leq k} w_i$ methods whose only purpose is forwarding and $\prod_{i \leq n} w_i$ methods that do something interesting.

As is traditionally presented, the visitor pattern is the case $n = 2$ with C_1 being the root of the AST class hierarchy and C_2 being the root of the visitors class hierarchy. There is exactly one forwarding method per AST class (and their total number is w_1 with the previous notation). In this guise, the visitor pattern can be seen as a way of grouping together the code that achieves one conceptual operation. For example, pretty printing an AST can be done by implementing a method *prettyPrint* in each AST class, but can also be done by putting all the pretty printing code into one visitor called *PrettyPrinter*. (Note that AstGen makes it hard to use the former approach, with a specific *prettyPrint* method in each class.)

FreeBoogie uses the traditional visitor pattern and the root of the visitors' class hierarchy is the class *Evaluator<R>*. The root of the Boogie AST class hierarchy is the class *Ast*. A subclass of *Evaluator<R>* is like a function of type $Ast \rightarrow R$, in the sense that it associates a value of type R (possibly **null**) to an AST node. For example, the type checker is a subclass of *Evaluator<Type>*. The base class *Evaluator* declares one *eval*($\mathscr{A}$) method for each AST class $\mathscr{A}$. These are the methods called m_2 in the previous discussion of the general visitor

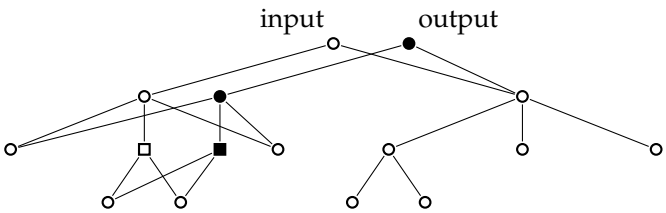


Figure 3.7: Path copying

pattern. These methods are not only declared, but they are also implemented, so that subclasses explicitly handle only the relevant types of AST nodes. For all the other AST node types, the default behavior implemented in *Evaluator* is to recursively evaluate all children and to cache the results. Because the *eval* methods of *Evaluator* are so similar, they are generated from an AstGen template.

An important type of evaluator is a transformer: The class *Transformer* extends *Evaluator*<*Ast*>. The main functionality implemented in *Transformer*, path copying, is illustrated in Figure 3.7. Empty nodes (○ and □) represent AST nodes that exist on the heap before a transformer T acts; filled nodes (● and ■) represent AST nodes created by the transformer T. Because the transformer T is interested only in rectangle nodes, it overrides only the *eval* method that takes rectangles as parameters. That overriden method is responsible for creating the filled rectangle (■). All the other filled nodes (●) are created by *Transformer*, and need not be of any concern to the particular transformer T.

The input and the output of a transformer usually share a large number of nodes. Since *Evaluator* caches the information that various evaluators associate with AST nodes, there is no need to repeat the computation of that auxiliary information for the shared parts. For example, most of the type information is already in the cache of the type checker.

Sometimes a transformer wants to 'see' AST nodes of type A even if it computes no value for them. A typical example is a pretty printer. In such cases a transformer may override *eval*(A) and return **null**. A nicer solution is to override *see*(A), whose return type is **void**. If both *eval*(A) and *see*(A) are overriden, then the former will be called by the traversal code in *Transformer*.

3.3.4 Immutability

In Java programming, it is unusual to constrain data structures to be immutable. Since the resulting code may look awkward to many programmers, there better be some good reasons for this design decision. In fact, awkward code, such as copying all but one of the fields in a new object instead of doing a simple

```
1  public class Renamer extends Transformer {
2    @Override public Identifier eval(Identifier identifier) {
3      if (!identifier.id().equals("u")) return identifier;
4      else return Identifier.mk("v");
5    }
6  }
```

Figure 3.8: Changing all occurrences of variable v into variable y

assignment, is only one of the apparent problems.

Immutability implies path copying, which is a potential performance problem. Consider the task of changing all occurrences of the variable u into variable v, which is achieved by the transformer in Figure 3.8. Suppose an AST with height h and n nodes contains exactly one occurrence of variable u. If the class *Identifier* would be mutable, one assignment would be enough to achieve the substitution; since the class *Identifier* is immutable, about h new nodes must be created and initialized. However, if there are two occurrences of variable u, they share some ancestors, meaning that less than about $2h$ new AST nodes must be created and initialized. Even more, if we take into account the tree traversal, then both implementations, with a mutable AST and with an immutable AST, take $\Theta(n)$ time. In other words, there is no asymptotic slowdown.

A Boogie block contains a *list* of statements (see Figure 3.4). Such lists should be immutable, but there are no immutable lists in the Java API (**a**pplication **p**rogramming **i**nterface), only immutable *views* of lists. Immutable collections can be implemented such that immutability is enforced statically by the compiler or such that immutability is enforced by runtime checks. Unfortunately, the former is incompatible with implementing Java API interfaces [14]. For example, in order to use the iteration statement **for** (T x : xs), one must implement the interface *Iterable* that contains the method *remove*. Obviously, calls to the *remove* method are not prevented statically by the compiler. FreeBoogie uses the *ImmutableList* class from the Google Collections [11] library, which follows the approach with runtime checks. (Figure 3.6 shows that the *ImmutableList* is used whenever the [list] tag appears in the abstract grammar.)

However, the advantages of immutability outweigh its disadvantages.

First, immutability enables *Evaluator* to cache the results of previous computations, because only immutable data structures can be used as keys in maps. A particular evaluator, such as the type-checker, need not mention anywhere in its implementation that caching is used. Yet, if the type-checker is invoked twice on the same AST fragment, then the second call will return immediately. This leads to cleaner code also because AST transformers need not bother with updating the auxiliary information—recomputing it is cheap. These advantages are discussed further in Chapter 6.

Second, immutability makes the code easier to understand, because it frees the programmer from thinking about aliasing of AST nodes. In Java, any mutation of $u.f$ must be done only after thinking how it will affect code that uses possible aliases of u. Because the AST is a central data structure in FreeBoogie, there is a lot of potential aliasing that must be considered whenever a mutation is done. It is much simpler to forbid mutations altogether.

Still, there are situations when the programmer must think about aliasing of AST data structures. It is natural to think of an AST reference as *being* a piece of a Boogie program, even if, strictly speaking, it only *represents* a piece of a Boogie program. To maintain this useful illusion the programmer must ensure that no sharing occurs within one version of the AST. More precisely, there should never be more than one reference-paths between two AST nodes. (There is a *reference-edge* $u \to v$ from the object referred by u to the object referred by v when $u.f{==}v$ for some field f.) For example, if the expression $x + y$ appears multiple times in a Boogie program, then the corresponding AST also appears multiple times, instead of being shared. In practice, this means that the programmer must occasionally clone pieces of the AST when implementing transformers. (The *clone* method is implemented in the code template for AST classes.)

3.4 Auxiliary Information

The package *freeboogie.tc* derives extra information from a Boogie AST—types, a symbol table, and a flowgraph.

The AST constructed by the parser is type-checked in order to catch simple mistakes in the input. As a safeguard against bugs, the AST is type-checked after each transformation. A side-effect of type-checking is that the type of each expression is known.

The symbol table helps in navigating the AST. It consists of one-to-many bidirectional maps that link identifier declarations to places where the identifiers are used. The only such map that is relevant to core Boogie is the one that links variable declarations, including those in arguments, to uses of variables. The other maps, relevant to full Boogie, link procedure declarations to procedure calls, type declarations to uses of user defined types, function declarations to uses of (uninterpreted) functions, and type variables to uses of type variables. (Type variables are similar to generics in Java.) All these maps are in the class *SymbolTable*.

Another bidirectional map is built by *ImplementationChecker*: In full Boogie a *procedure* may have zero, one, or multiple *implementations*. (In core Boogie,

the whole program is one implementation.)

Finally, it is sometimes convenient to view one implementation as a flowgraph whose nodes are statements. Such a flowgraph is built by *FlowGraphMaker*. Formally, a flowgraph is defined as follows.

Definition 5. A *flowgraph* is a directed graph with a distinguished *initial node* from which all nodes are reachable.

It seems natural that a flowgraph has an initial node, because there is usually one entry point to a program. It seems less natural that all nodes must be reachable, which means that there is no obviously dead code. The reason for this standard restriction is rather technical: It simplifies the study of flowgraph properties. However, it does complicate slightly the definition of what it means for a flowgraph to correspond to a core Boogie program. A few terminology conventions will help. In Chapter 2 we noticed that we can attach counters to statements because they are in a list. For concreteness, let us use the counters $1, 2 \ldots, n$, in this order, when the list of statements has length n. Each counter x in the range $0..n$ has an associated statement, named statement x. Statement 0 is the sentinel statement **assume true**, which is prepended for convenience. Label x is the label that precedes statement x, if there is one.

Remark 2. The sentinel statement 0 is not introduced by the FreeBoogie implementation. It is merely a device that will simplify the subsequent presentation, especially some proofs.

The flowgraph of a Boogie program is constructed, conceptually, in two phases.

Definition 6. The *pseudo-flowgraph of a core Boogie program* with n statements has as nodes statement 1 up to statement n and the sentinel statement 0. It has an edge $0 \to 1$ (from statement 0 to statement 1) and has edges $x \to y$ when (a) statement x is **goto** label y, or (b) statement x is *not* **goto** and label y is the successor of label x.

Remark 3. Compare with (2.6)–(2.9). This definition roughly says that there is an edge where the operational semantics rules would allow a transition *if we ignore the upper parts of the rules*.

Definition 7. The *flowgraph of a core Boogie program* is a graph that has node 0 as its initial node. Its nodes V are those nodes of the pseudo-flowgraph that are reachable from node 0 and that are not **goto** statements. It has an edge $m \to n$ when there is a path $m \rightsquigarrow n$ in the pseudo-flowgraph that is disjoint from V, except at endpoints.

```
1  procedure dead(x : int) returns () {
2    L1: assume x > 0;
3    L2: goto L4, L6;
4    L3: assume true;
5    L4: assume x < 0;
6    L5: return;
7    L6: assume true;
8    L7: return;
9  }
```

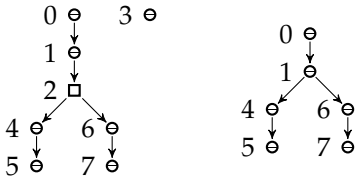


Figure 3.9: Flowgraph of a core Boogie program

Example 1. Figure 3.9 shows a core Boogie program, its pseudo-flowgraph, and its flowgraph. (Label k is L_k.) Chapter 7 discusses *semantically* unreachable nodes of the flowgraph, such as node 4 in this example.

Proposition 1. *The only nodes that have no outgoing edges in a flowgraph of a core Boogie program are those that correspond to* ***return*** *statements.*

All auxiliary information is available through *TcInterface*, which is an implementation of the Facade pattern.

3.5 Verification Condition Generation

The package *freeboogie.vcgen* consists of Boogie transformers and Boogie to SMT transformers. The facade of this package is the class *VcGenerator*.

Most Boogie transformers are responsible for small AST modifications such as desugaring an **if** statement into **assume** and **goto** statements. For speed, it would be better to cluster many such simple transformers into one, but the code is easier to maintain if they are kept separate. A few helper classes are used by multiple Boogie transformers: *CommandDesugarer* is used as a base class by transformers that change statements into lists of statements; *ReadWriteSetFinder* is an evaluator that associates with each statement two sets—the set of variables that are read and the set of variables that are written.

Boogie transformations do not update the auxiliary information while they are building new AST nodes. Instead, at the very end, they recompute all auxiliary information, and caches ensure that no computation is repeated. This way, bugs that produce untypable Boogie programs get caught at run-time. (Type information is auxiliary information, so type-checking is repeated.)

The Boogie to SMT transformation is done by the class *WeakestPrecondition* or by the class *StrongestPostcondition*, depending on the command line options. The theory behind these two classes is presented in Chapter 5.

3.6 The Prover Backend

The package *freeboogie.backend* contains (1) SMT data structures and (2) code to communicate with provers. The design is inspired by the sorted multi-prover backend in ESC/Java.

3.6.1 Data Structures and Sort-Checking

The main data structure is a rooted ordered tree whose nodes are labeled by strings. Each node has a *sort*, and there are sort-checking rules, which say what combinations of sorts and labels are valid. In effect, sorts are types—the only reason a different name is used is to distinguish SMT sorts from Boogie types. In ESC/Java it is impossible to construct a tree that has sort errors: Programs that try to construct invalid terms fail Java type-checking. Such a strong static guarantee is appealing, but has a significant disadvantage: The size of the backend is big. For example, instead of a single factory method *SmtTree mk*(*String label*, *ImmutableList*<*SmtTree*> *children*) there is a plethora of methods with various argument and return types, such as the method *SmtFormula mkEq*(*SmtTerm left*, *SmtTerm right*), where both *SmtFormula* and *SmtTerm* are subtypes of *SmtTree*. Because of the size, the backend is hard to adapt to changes. FreeBoogie opts for dynamic checks instead of static guarantees so that the backend is smaller and easier to maintain.

Before calling *mk*(*label*, *children*) the label must have been defined. For example, after the call *def*(`"eq"`, **new** *Sort*[]{*Sort.TERM*, *Sort.TERM*}, *Sort.FORMULA*) it is possible to call *mk*(`"eq"`, *children*). This second call will check (using Java assertions) that there are two children and both are terms, and will mark the constructed SMT tree as being a formula. All defined labels are grouped in stack frames, such that the call *popDef*() discards all definitions done after the corresponding call *pushDef*(). Such grouping is useful because some labels refer to constructs built into SMT solvers and other labels refer to uninterpreted functions that are defined by the Boogie program. When FreeBoogie moves from one input file to another it forgets about labels corresponding to functions while not forgetting about labels corresponding to solver built-ins by using the stack mechanism.

The methods *def*, *mk*, *pushDef*, and *popDef* are all defined in the class *TreeBuilder*. For convenience, the functions *def* and *mk* are overloaded.

Let us first look at the method *mk*. It comes in three varieties:

$$mk(\texttt{"and"}, children) \tag{3.1}$$

$$mk(\texttt{"eq"}, t_1, t_2) \tag{3.2}$$

$$mk(\texttt{"literal\_int"}, \textbf{new}\ FbInteger(3)) \tag{3.3}$$

The first form takes a list of children as the second argument. When the number of children is fixed, as is the case for the label `eq`, it is convenient to hide the building of the list behind a helper overload. The second form can be used when the number of children is one, two, or three. The third form is special. Strictly speaking, the constants 1, 2, 3 ... are distinct uninterpreted functions that take no argument. This suggests that they should each be defined separately, which is clearly a very bad idea from the point of view of performance. So, instead of defining labels `1`, `2`, `3`, ..., we define the meta-label `literal_int`. A *meta-label* has an associated Java type (in this case *FbInteger*) and it is equivalent to multiple labels, one for each value of the associated Java type. In other words, the meta-label `literal_int` and the value **new** *FbInteger*(3) determine the label, and there is no child.

Now let us look at the method *def*. It comes in three varieties:

$$def(\texttt{"and"}, Sort.FORMULA, Sort.FORMULA) \tag{3.4}$$

$$def(\texttt{"eq"}, \textbf{new}\ Sort[]\{Sort.TERM, Sort.TERM\}, Sort.FORMULA) \tag{3.5}$$

$$def(\texttt{"literal\_int"}, FbInteger.\textbf{class}, Sort.INT) \tag{3.6}$$

The order of the arguments is: label, sort of arguments, sort of result. The example for the first form says that tree nodes labeled with `and` may have any number of children, all of whom must be formulas, and the tree itself is a formula. The example for the second form says that tree nodes labeled `eq` have two children, the first one is a term, the second one is a term, and the tree itself is a formula. The example for the third form says that the meta-label `literal_int` together with a value of type *FbInteger* constitutes a label, and trees labeled in this way are integers.

(As a side note, *FbInteger* is used because Boogie allows arbitrary large integers and has bit vector operations. No class in the standard Java library supports both.)

Any SMT trees s and t have the property that $s.equals(t)$ implies $s==t$. This is implemented by maintaining a global set of all SMT trees that were created, a technique sometimes known by the name *hash-consing*.

3.6.2 The Translation of Boogie Expressions

The methods *mk* provide one way of building trees; the method *of* provides another way of building trees. For example, the class *StrongestPostcondition* uses the methods *mk* to connect formulas (using the labels `and`, `implies`) and uses the method *of* to obtain the formulas corresponding to individual assertions and assumptions.

The method *of* converts from Boogie expressions to SMT formulas. The actual work is done in two classes—*TermOfExpr* and *FormulaOfExpr*. The translation is almost one-to-one. Each Boogie operator has a corresponding interpreted symbol in the SMT language; each function declared in a (full) Boogie program behaves similarly to an uninterpreted function symbol in the SMT language. There is, however, an important deviation from the one-to-one correspondence. As we have seen, the SMT language distinguishes between terms and formulas. Roughly speaking these correspond, respectively, to non-boolean Boogie expression and to boolean Boogie expressions. For example, in the SMT language the operands of logical-and must be formulas and in Boogie the operands of logical-and must be booleans. On the other hand, an SMT uninterpreted symbol is always a term, while in Boogie a function may return a boolean. Also, in SMT the arguments of an uninterpreted symbol must be terms, while in Boogie a function might take booleans as arguments. Because of these reasons, a one-to-one translation may produce ill-formed SMT trees, which fail sort-checking.

In SMT the constants **true** and **false** are a formulas. If we introduce two corresponding uninterpreted terms, *trueTerm* and *falseTerm*, we can then try to fix the ill-formed SMT tree using the following two rules.

1. If a term τ appears where a formula is expected then we replace the term by (= τ *trueTerm*). This compares for equality τ and *trueTerm*.

2. If a formula φ appears where a term is expected then we replace the formula by (**ite** φ *trueTerm falseTerm*). This expression evaluates to *trueTerm* for all interpretations in which φ evaluates to $\top$; it evaluates to *falseTerm* for all interpretations in which φ evaluates to $\bot$.

This, however, is not exactly what FreeBoogie does. Simplify is an old but still competitive prover whose language is similar to the SMT language. One difference is that in Simplify a term never contains a formula. In particular, there is no **ite**, so the rule 2 from above cannot be used.

To clarify these ideas, let us turn to an example.

```
1 function f(x : bool) returns (bool);
2 axiom (forall x : bool :: x == f(x));
3 procedure p() returns () { assert f(true) != f(false); }
```

The assertion should hold.

The following is what FreeBoogie sends to Simplify.

```
1 (BG_PUSH (NEQ trueTerm falseTerm))
2 (BG_PUSH (FORALL (xTerm) (EQ xTerm (f xTerm))))
3 (NOT (IFF (EQ trueTerm (f trueTerm)) (EQ trueTerm (f falseTerm)))))
```

In Simplify's language, interpreted symbols are written in CAPITAL letters and their names are usually self-explanatory. The command **BG_PUSH** communicates a hypothesis to the prover. Line 3 is a query. The Boogie constants **true** and **false** that appear as arguments of the function f were translated into terms directly, without an intermediate application of rule 2. The comparison between booleans • != •, which appears in the assertion, was translated to (**NOT** (**IFF** • •)). Because **IFF** expects formulas as arguments and because (f ...) is a term, rule 1 was applied, which is why **EQ** appears in the query.

The hypothesis on line 1 is necessary. In fact, it is equivalent to the query.

(**NOT** (**IFF** (**EQ** *trueTerm* (*f trueTerm*) (**EQ** *trueTerm* (*f falseTerm*)))))
= { instantiate hypothesis 2 with $xTerm = trueTerm$ }
(**NOT** (**IFF TRUE** (**EQ** *trueTerm* (*f falseTerm*)))))
= { boolean algebra }
(**NEQ** *trueTerm* (*f falseTerm*))
= { (*f falseTerm*) is the same as *falseTerm*, again from hypothesis 2 }
(**NEQ** *trueTerm falseTerm*)

The example illustrates that it is sometimes necessary to introduce new hypotheses. If not enough hypotheses are introduced, then FreeBoogie is incomplete, but should still be sound. Whenever *TermOfExpr* or *FormulaOfExpr* produce an SMT tree, they may attach to it extra hypotheses. These are SMT trees themselves, and may have further hypotheses attached. All hypotheses are collected and sent to the prover before the query.

3.6.3 Talking to the Prover

The class *Prover* defines the interface that is used by the package *freeboogie.vcgen* to talk to the prover. It is a thin interface, consisting of the methods *assume*, *retract*, *push*, *pop*, and *isValid*.

The real prover does not have to have the notion of an assumption (also known as hypothesis), but a class that extends *Prover* should take advantage of all facilities of a real prover. For example, if Simplify is used as a prover, then a sequence of calls *assume*(h), *isValid*(q_1), *isValid*(q_2) may result in one of

the following two strings being sent to the prover:

$$(\textbf{IMPLIES}\ h\ q_1)\ (\textbf{IMPLIES}\ h\ q_2) \tag{3.7}$$

$$(\textbf{BG\_PUSH}\ h)\ q_1\ q_2 \tag{3.8}$$

Both are OK, but the second is better, if only because h is communicated once.

Similarly, a class that extends *Prover* may choose to treat certain SMT tree labels specially to take advantage of other facilities of the real prover.

3.7 Other Generated Code

The Boogie parser resides in the package *freeboogie.parser* and is generated by ANTLR (**an**other **t**ool for **l**anguage **r**ecognition); the command line parser resides in the package *freeboogie.cli* and is generated by CLOPS (**c**ommand **l**ine **op**tion**s**).

3.8 Related Work

The main goal of the previous sections is to anchor the subsequent theoretical chapters in a concrete program, FreeBoogie. A secondary goal is to serve as a guide to the code and to make explicit the early design choices. This section is for the reader who wants to understand the design in detail, but feels that the previous sections are too shallow.

3.8.1 Similar Tools

FreeBoogie is a Java clone of the Boogie tool [29] from Microsoft Research. The internals differ but the input and the output interfaces are the same. The input is a program written in the Boogie language [119, 124]; the output is a formula written in the SMT language, or in a similar language.

The Boogie language is imperative. The Why language [75] is functional and, like Boogie, was designed to be an intermediate language for program verifiers. The Why language is used by the Why tool.

Since both the Boogie tool and the Why tool accomplish tasks similar to FreeBoogie, it is interesting to compare their design decisions. For example, it is easy to see that they were written in different programming languages. Both Boogie and FreeBoogie can be used to verify themselves. The Boogie tool is written in Spec$^\sharp$, a superset of C$^\sharp$, and is part of a verification system [31] for Spec$^\sharp$ programs. FreeBoogie is written in Java and can be used to verify Java

programs if it is combined with the converter B2BPL from Java bytecode to Boogie that was developed at ETH Zürich. (The source code of B2BPL is included in the FreeBoogie code repository.) In the case of the Why tool, however, the choice of language was not governed by such recursive considerations: It is written in OCaml, a language well suited for implementing compilers, but it is not used to verify OCaml code. The main use of the Why tool is in the Jessie plugin of the Frama-C framework [8] for the verification of C programs. Similarly, it is interesting to compare approaches to representing ASTs, interfacing with provers (Why being especially interesting), and so on.

3.8.2 Design, Pipeline, and Correctness

Because the input and the output are programs written in well-defined languages, FreeBoogie is a compiler. The standard text on compiler design is the Dragon Book [24]. The pipeline architecture is common to most compilers.

One of the oldest problems studied by the formal methods community is the correctness of compilers. The early approaches were focused on proving that a compiler is correct (see, for example, [133]). The idea is to show that the semantics of the input program are in a certain relationship with the semantics of the output program. The relationship usually ensures that the two programs have the same observable behavior. Although there is recent research in the same vein [126], there are also attempts to go around the problem and avoid the full verification of the compiler. One of these alternative approaches is *translation validation*, proposed in 1998 by Amir Pnueli and others [147]. The idea is to check the result of each particular compilation, instead of proving that the compiler works for all possible inputs. To make it even easier, instead of checking the whole compilation, the idea can be applied to each compilation phase. The technique was used to find bugs in GCC (the **G**NU **C** compiler) [137]. A related technique, *credible compilation* [153], consists of modifying the compiler to output a proof for each compilation. It if then possible to check the proof of equivalence using a small trusted proof checker. In 2004, Benton [40] introduced a way of doing equivalence proofs that seems to fit well with the Boogie language.

The problem of correctness of a program verifier has certain peculiarities. The 'observable behavior' of a Boogie program, which is not executable, is whether it is correct or not, according to Definition 2. It follows that two Boogie programs are equivalent if they are both correct or both incorrect. As with normal compilers, a transformation of a Boogie program is correct if the output is equivalent to the input. Section 3.2 says that some transformations in FreeBoogie are incorrect and it identifies soundness and completeness as

weaker guarantees.

The Dragon Book analyzes the problem of correctness from an algorithmic point of view, but it does not address the problem of correctness of an implementation; the Dragon Book discusses the overall pipeline architecture of a compiler but does not go into details of code organization. The Design Patterns book [82] partly covers this area. For example, the visitor pattern, the facade pattern, and the factory method pattern, which were used to explain FreeBoogie's design, are all presented in that book. Of these three patterns, only the visitor pattern was briefly analyzed in Section 3.3.3.

"[A] *facade* provide[s] a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use." In this dissertation a more specific meaning is used—a class (or interface) that provides almost all the services of a package. The non-facade classes are still visible from outside, in case they are needed, but their use is discouraged.

A *factory method* creates an object and returns it. The Design Patterns book emphasizes that the method call may be dynamically dispatched to subclasses. While this aspect is used by the backend, the factory methods for Boogie AST are static. Such methods lead to code that is easier to read because of their descriptive names and because Java's type inference for generics works better for methods than for constructors, at least in version 6. Such methods also make possible the implementation of more sophisticated creation patterns, such as singleton and hash-consing.

3.8.3 The Expression Problem

The visitor pattern is a way of achieving multiple dynamic dispatch, but it is also a partial solution to the expression problem. The term was coined by Philip Wadler in an email [172] from 1998: "The *Expression Problem* is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining type safety." Since most languages have modular compilation, the restriction on recompilation usually means that one is allowed to add new modules but not to modify existing ones. Wadler continues: "One can think of cases as rows and functions as columns in a table. In a functional language the rows are fixed [...] but it is easy to add new columns. In an object-oriented language, the columns are fixed [...] but it is easy to add new rows."

The visitor pattern is a way of easily adding new columns in an object-oriented language. In compilers, functionality tends to evolve more than the AST. This is one reason why functional languages are well-suited for implement-

ing compilers and it is why most compilers written in object oriented languages use the visitor pattern. However, once the visitor pattern is used, it becomes hard to modify the AST.

FreeBoogie is written in an object oriented language and uses the visitor pattern. To add new functionality, one implements a new evaluator or a new transformer, both being visitors. The code implementing the new functionality goes in a new class so, in Wadler's terminology, it is easy to add columns. If a new type of node must be added to the AST then one new class must be added to the AST data structures. So far, the existing code was not touched and no recompilation is necessary. The next step, however, is to add a methods to *Evaluator*, the root of the hierarchy of visitors. In Wadler's terminology, it is hard to add rows. What is problematic in practice is not the recompilation time, but the fact that the AST data structures and the base classes for visitors must be kept in sync. Because the AST data structures, the class *Evaluator*, and the class *Transformer* are generated by AstGen from the same description of the abstract grammar they are *automatically* in sync. In other words, AstGen can be seen as a patch that brings the visitor pattern closer to the ideal solution for the expression problem. (But recompilation is still necessary when new AST nodes are added.)

Another approach to the expression problem is to modify the programming language to support multiple dispatch [53, 56].

3.8.4 Provers

Many languages understood by theorem provers (such as the SMT language, the Simplify language, and the PVS prover language) are based on S-expressions. Like XML [46] (the e**x**tensible **m**arkup **l**anguage), S-expressions provide a syntax that is easy to parse. Briefly, they are a fully parenthesized preorder print of an AST. Where XML says `<tag>` $\cdots$ `</tag>`, S-expressions say `(tag ...)`. (There is no equivalent for XML attributes.) One year before writing about the semantics of programs, John McCarthy presented [131] in 1960 how S-expressions are used in the programming system LISP (**lis**t **p**rocessing). "S stands for symbolic."

Internally, FreeBoogie uses a tree of strings to represent symbolic expressions. To save memory and to speed up otherwise expensive structural comparisons FreeBoogie uses hash-consing. The idea was described by Andrei Ershov [73] in 1957, and one year later an English translation was available. Before creating a tree node, a global hash table is searched to see if a structurally similar node already exists. The creation of nodes takes a constant amount of time on average, structural comparison of tree nodes is done then by a simple pointer comparison, and much memory is saved because duplication of information is avoided. A

simple implementation of hash-consing is easy, but offering proper library support is not trivial—a solution [74] for OCaml was published in 2006.

The tree of strings was chosen as the main data structure of *freeboogie.backend* because it is a natural representation of the SMT language. The SMT community [32] produced a language [150], a command language, theories, benchmarks; it also organizes an annual competition [34] between SMT provers. The language defines, on top of S-expressions, the meaning of about twenty keywords and about a dozen 'attributes'. There are more than a dozen actively developed provers and more than a dozen other provers that support the SMT language. The SMT language itself describes the syntax and semantics of formulas and of terms, but it does not provide any other way of communicating with the prover. The SMT command language specifies how to interact with a running prover—how to ask "is formula φ valid," how to say "from now on please assume φ' holds," what format the prover should use to answer, and so on. The theories are the 'standard library' of the SMT language. Each defines the meaning of a set of symbols. For example, the theory Ints, defines the semantics of the symbols `0`, `1`, `~`, `-`, `+`, `*`, `<=`, `<`, `>=`, and `>`. The benchmarks include handcrafted queries, random queries, queries produced from hardware verification tasks, and queries produced from software verification tasks. A random sample of the benchmarks is used in the SMT competition. Since the SMT language is supported by many provers that compete each year on software verification tasks, it makes a natural target for FreeBoogie.

According to the results of the most recent SMT competition the best provers are Barcelogic [44], Boolector [47], MathSAT [48], SatEEn [21], Yices [23], and Z3 [65]. Each of these won at least one category in 2009. (Although Z3 did not officially participate, its version from 2008 was still tested and it unofficially won a few categories.) FreeBoogie was used so far with FX7 [9], Simplify [67], and Z3, which support the slightly different language of Simplify. Unfortunately, the source code of all the best provers is not open. FX7 and Simplify are open source but not actively maintained, and are written in esoteric languages whose compilers are hard to get (Nemerle and Modula 3, respectively).

A better way to interact with provers is through an API. Boolector, MathSAT, Yices, and Z3 provide a C API. Z3 provides also a .NET API and an OCaml API. Unfortunately, they are different, since SMT does not define an uniform API.

3.8.5 Code Generators

The Boogie AST is generated by AstGen; the Boogie parser is generated by ANTLR; the command-line parser is generated by CLOPS.

ANTLR [143] is probably the most widely used parser generator for Java.

The grammar is LL(k), but backtracking can be optionally activated. FreeBoogie sticks to LL(k) so that the generated parser is as fast as possible. ANTLR can construct AST trees. This facility is not used. It is not easy to convince ANTLR to generate different data structures for the AST. For example, it is not easy to convince ANTLR to generate immutable ASTs. Also, it is not easy to convince ANTLR to generate the root of the visitors' hierarchy in sync with the AST data structures.

CLOPS [105] is a Java parser generator specialized for command lines. FreeBoogie is one of its first users.

There are, of course, other parser generators for Java. ANTLR was chosen because it is widely used and because a C++ version of AstGen combined with Bison [2] already showed that the approach is viable. CLOPS was chosen because its authors were nearby and easy to convince to add new features and fix bugs, if needed.

Chapter 4

Optimal Passive Form

"Active Evil is better than Passive Good."
— William Blake

Flanagan and Saxe [78] explain how passivation avoids the exponential explosion of VCs. This chapter (1) gives a precise definition for passivation and (2) proceeds to examine its algorithmic difficulty.

4.1 Background

4.1.1 VC Generation with Assignments

The best way to understand why passivation is such a good idea, especially from the point of view of performance, is to compare it to a VC generation method that skips passivation. This means that we have to look ahead at the next stage of the pipeline and see how it would have to work if passivation is not performed. For this reason, the methods presented in this *subsection* will only be fully justified in the next chapter.

Weakest Precondition for Core Boogie

The weakest precondition transformer for the three statements that can appear in a flowgraph (see Definition 7 on page 31) is

$$wp\,(\textbf{assume}\ p)\,b \equiv p \Rightarrow b \tag{4.1}$$

$$wp\,(\textbf{assert}\ p)\,b \equiv p \wedge b \tag{4.2}$$

$$wp\,(u := e)\,b \equiv (u \leftarrow e)\,b \tag{4.3}$$

Here b and p are predicates as defined in Chapter 2, where the predicate transformer $(u \leftarrow e)$ is also introduced. Symbolically, the effect of $(u \leftarrow e)$ is to substitute the expression e for each occurrence of the variable u.

Each node x of the flowgraph gets a precondition a_x and a postcondition b_x according to the rules

$$b_x \equiv \bigwedge_{x \to y} a_y \tag{4.4}$$

$$a_x \equiv wp\ x\ b_x \tag{4.5}$$

The query sent to the prover is the predicate a_0.

Figure 4.1 shows a program that leads to a very big query. Let us denote by $q_k(u)$ the postcondition of the statement **assert** $p_k(u)$. Then $q_n(u) \equiv \top$ and $q_{n-1}(u) \equiv p_n(e_n(u)) \wedge p_n(f_n(u))$. In general, all q_ks will be conjunctions of function compositions, so let us use a simplified notation just here, while we evaluate the memory usage of the weakest precondition method.

$$q_{n-1} = \{p_n e_n, p_n f_n\} \tag{4.6}$$

Here pe stands for the function composition $p \circ e$, and $\{p, q, r\}$ stands for the conjunction $p \wedge q \wedge r$. In general, the rule is

$$q_{k-1} = \{p_k e_k, p_k f_k, q_k e_k, q_k f_k\}. \tag{4.7}$$

For example,

$$q_{n-2} = \{p_{n-1}e_{n-1}, p_{n-1}f_{n-1}, p_n e_n e_{n-1}, p_n f_n e_{n-1}, p_n e_n f_{n-1}, p_n f_n f_{n-1}\} \tag{4.8}$$

All these expressions are represented using SMT trees, which are really dags because of hash-consing. In evaluating the memory space necessary to represent a predicate we must take into account subexpressions that appear more than once. One such subexpression is u, which appears quite often. Is there any other sharing in, say, the predicate q_{n-2}? Well, it contains $p_{n-1}e_{n-1}$ and $p_n e_n e_{n-1}$, which both end in e_{n-1}, so q_{n-2} contains e_{n-1} at least twice.

In general, if we regard e_k, f_k, and p_k as (distinct) letters from some alphabet and q_k as a set of strings, then we are interested in the size of the trie that represents the reversed strings of q_k. Figure 4.2 shows the trie for q_{k-1}. (This trie is, roughly, an upside-down depiction of the SMT tree, except that SMT tree nodes correspond to edges in the trie.) If we denote by $|q_k|$ the number of edges

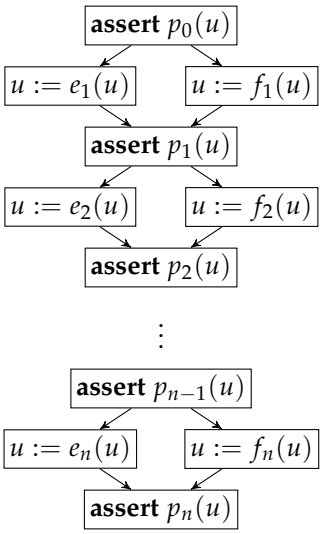


Figure 4.1: Exploding diamonds

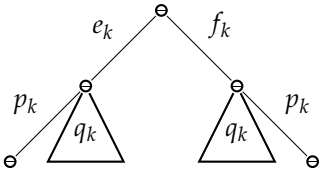


Figure 4.2: The trie constructed by (4.7)

in the trie necessary to represent q_k, then

$$|q_{n-1}| = 4, \tag{4.9}$$

$$|q_{k-1}| = 2|q_k| + 4. \tag{4.10}$$

So $|q_0| = 2^{n+2} - 4$. Of course, the SMT tree for the whole query will also contain p_0, u, and a node for $\wedge$. In any case, the size of the query is $\Theta(2^n)$. We will see later that if passivation is applied first, then the size of the query is $\Theta(n)$. This is one reason why we passivate programs.

Flanagan and Saxe [78, Section 4] identify sequences of assignments like $x_{k+1} := x_k + x_k$ for $k = 0, 1, \ldots, n-1$ as another possible source of exponential explosion. Such programs are problematic only in the absence of hash-consing.

Strongest Postcondition of Core Boogie

What about strongest postcondition? It is computed for the three types of statements using the following rules.

$$sp\,(\textbf{assert/assume}\ p)\,a \equiv a \wedge p \tag{4.11}$$

$$sp\,(u := e)\,a \equiv \exists v, \big((u \leftarrow v)\,a\big) \wedge \big(u = (u \leftarrow v)\,e\big) \tag{4.12}$$

A serious problem is already apparent: The strongest postcondition of an assignment contains an existential quantifier, which means that it is *very* probable that no SMT prover will be able to answer the query. For example, Z3, one of the best SMT solvers, does not even prove that $(\exists x,\ x = 0)$.

For completeness, the other equations used by the strongest postcondition method are

$$a_y \equiv \begin{cases} \top & \text{if the node } y \text{ is initial} \\ \bigvee_{x \to y} b_x & \text{otherwise} \end{cases}, \tag{4.13}$$

$$b_n \equiv sp\ n\ a_n. \tag{4.14}$$

4.1.2 A Few Concepts from Computational Complexity

One of the main results of this chapter is that a certain variation of passivation is NP-complete. This section is a brief reminder of what NP-complete means.

Very roughly, The RAM (**r**andom **a**ccess **m**emory) model of computation consists of a processor and a memory whose words each have a fixed number of bits and are indexed by integers that fit in a word. The processor can execute binary operations between two memory words and store the result in a (not necessarily distinct) third word in one time step. Operations include modular arithmetic and bitwise logic. The processor can also (1) request one word of input and (2) produce one word of output.

Definition 8. A *computational problem* is a relation $P \subseteq L \times L$, where L is the set of finite sequences of words $w_1, w_2, \ldots, w_n$, for some n.

We say that y is a *valid solution* (or *valid output*) for the *problem instance* (or *input*) x when xPy. An *algorithm* is the finite set of instructions that the processor is hardwired to carry out. We say that an algorithm is a *solution* to a problem when it produces valid output for every input. (For simplicity, let us assume that there is a valid output for every input.) We write $|w|$ for the length n of a sequence $w_1, w_2, \ldots, w_n$.

Definition 9. The *time complexity* of an algorithm is a function $t : L \to \mathbb{Z}_+$ that associates to each input the number of steps executed by the algorithm

before producing the output. The *space complexity* of an algorithm is a function $s : L \to \mathbb{Z}_+$ that associates to each input the number of memory words touched by the algorithm before producing the output.

The *worst case* complexity $T(n)$ is the maximum complexity $t(x)$ over inputs x of size n. Similarly, one can define an *average complexity*, if probabilities are associated to inputs.

Definition 10. A computational problem is in the complexity class P when it has a solution whose worst case time complexity has a polynomial upper bound.

Definition 11. A *decision problem* is a set $D \subseteq L$.

Equivalently, a decision problem, is a computational problem whose answer is 0 or 1, that is, a problem with at most two valid outputs.

Definition 12. A decision problem D is in the complexity class NP when there exists a subset $R \subseteq L \times L$ with the properties:

1. There is a polynomial p such that $|y| \leq p(|x|)$ whenever xRy.
2. For all $x \in D$ there is an y such that xRy. There is no such y for $x \notin D$.
3. The problem of deciding whether xRy is in P.

The element y is a *proof* that $x \in D$; a solution to problem 3 in the above definition is a *verification procedure*.

This definition applies to decision problems, but we will later use it with respect to optimization problems.

Definition 13. An *optimization problem* asks for the minimum cost feasible solution. It is specified by a function $f : L \to L \to \mathbb{B}$ that associates a set of feasible solutions to each input and, for each input x, a function $(c\ x) : (f\ x) \to \mathbb{R}_+$ that gives a nonnegative cost to each feasible solution. A valid output for the input x is a feasible solution $y \in (f\ x)$ such that all other feasible solutions $y' \in (f\ x)$ have at least the same cost ($c\ x\ y' \geq c\ x\ y$).

Each optimization problem has an associated decision problem that asks whether there exists a feasible solution of cost $\leq k$, for some constant k. We say that an optimization problem is in NP when its associated decision problem is in NP. Note that if we have a solution to the associated decision problem, then we can find the cost of an optimal solution with a logarithmic overhead by using binary search. This is an example of reducing a problem to another.

Definition 14. An *oracle* for problem P solves any instance of P in constant time and constant space.

Definition 15. A decision problem P *reduces* to a decision problem Q when there is a polynomial time solution for P that uses an oracle for Q as a subroutine. A decision problem P *transforms* to a decision problem Q when there is a function $f : L \to L$ computable in polynomial time such that $x \in P$ if and only if $f(x) \in Q$, for all inputs x.

Remark 4. Karp defined transformability and Cook defined reducibility. Alas, both used the term 'reduction'. There are still authors who use the term 'reduces' to mean what was defined above as 'transforms'.

Definition 16. A problem is *NP-complete* when it is in NP and all other problems in NP transform to it. A problem is *NP-hard* when all problems in NP reduce to it.

As before, we say that an optimization problem is NP-complete/NP-hard when its associated decision problem is NP-complete/NP-hard.

No polynomial solution is known for any NP-complete problem. The only way to handle large instances of NP-complete problems in practice is to settle for approximations or heuristics.

4.1.3 Algorithms and Data Structures Reminder

One of the proofs given later is by transformation from the MINS problem (**m**aximum **i**ndependent **n**ode **s**et). Then a conjecture is supported by a heuristic argument that makes use of the LIS problem (**l**ongest **i**ncreasing **s**ubsequence) and the maximum bipartite matching problem. This section serves as a brief reminder of what these problems are and what are the best algorithms known for them.

Maximum Independent Node Set

Definition 17. Two nodes of a graph are *adjacent* when they are the endpoints of some edge. A set of nodes is *independent* when its elements are pairwise non-adjacent.

Problem 1 (maximum independent node set)**.** *Given is a graph. Find a largest independent set of nodes.*

The associated decision problem asks whether there is an independent set of size $\geq k$.

Example 2. The graph ○-○-○ has one independent set of size 0 (○-○-○) three independent sets of size 1 (●-○-○ , ○-●-○ , ○-○-●) and one of size 2 (●-○-●). The latter is a solution to this instance of the MINS problem. There are also three sets of nodes that are not independent (●-●-○ , ○-●-● , ●-●-●).

```
LIS(w_1, w_2, ..., w_n)
1  r := 0
2  for i from 1 up to n
3       l := 1 + pred(w_i − 1)
4       r := max(r, l)
5       update(w_i, l)
6  return r
```

Figure 4.3: A solution for LIS

This problem is known to be NP-complete.

Longest Increasing Subsequence

Definition 18. A *subsequence* is obtained from a sequence $w_1, w_2, \ldots, w_n$ by removing some of its elements and maintaining the relative order of the others.

Problem 2 (LIS)**.** *Given is a sequence of integers. Find a subsequence of it that has maximum length.*

The best known solution for this problem works in $\Theta(n \lg \lg n)$ time. It uses a data structure that maps integers to integers and supports the operations $update(k, v)$ than binds the key k to the value v and $predecessor(k)$ that returns the value last bound to the largest key that is $\leq k$. For example, after $update(2, 1)$, $update(1, 2)$, $update(2, 3)$, the query $predecessor(1)$ returns 2 and the queries $predecessor(2)$ and $predecessor(3)$ return 3. For simplicity, let us assume $v \in \mathbb{R}_+$, which means that $predecessor(k)$ returns 0 if the value of k filters out all previous updates. The *update* and *predecessor* operations are supported by binary search trees in $\Theta(\lg n)$ time and by van Emde Boas trees in $\Theta(\lg \lg n)$ time. Figure 4.3 shows an algorithm that uses such a data structure to solve the LIS problem in $\Theta(n \lg \lg n)$ time. Strictly speaking, the algorithm only returns the length of a longest subsequence, but it can be easily modified to return the subsequence.

Maximum Bipartite Matching

Definition 19. A graph is *bipartite* when its nodes can be partitioned into two subsets, the left nodes and the right nodes, such that all edges are between a left and a right node.

Definition 20. Given a graph, a *matching* is a subset of pairwise non-adjacent edges. Two edges are *adjacent* if they share (at least) a node.

Problem 3 (maximum bipartite matching)**.** *Given is a bipartite graph. Find a largest matching.*

```
1  procedure makeEven(u : int) returns (v : int) {
2    L0: v := u; goto L1, L2;
3    L1: assume !even(v); goto L3;
4    L2: assume even(v); goto L4;
5    L3: v := v + 1;
6    L4: assert even(v); return;
7  }
```

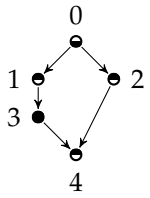


Figure 4.4: An example of a Boogie program

The old Hungarian algorithm solves the problem in $O(n^3)$ time. Other algorithms solve the problem in $O\big(\min(m\sqrt{n}, n^{2.38})\big)$ time. Here, m is the number of edges and n is the number of nodes.

4.1.4 Examples, Terminology, and Notations

The running example in Figure 4.4 is the simplest interesting case for passivation. Variable v is the variable of interest. With respect to it, the nodes are *read-only* (◓), *write-only* (◒), or *read-write* (●). A statement that does not involve the variable of interest will be drawn as a white circle (⊖). There is an easy mnemonic rule for these notations: Since execution flows downward and reading precedes writing, the upper half corresponds to reading and the lower half to writing. The terms *read node* (for ◓ or ●) and *write node* (for ◒ or ●) will also be used. In Figure 4.4, nodes 1, 2, 3, and 4 are read nodes, while nodes 0 and 3 are write nodes.

Figure 4.5(a) shows a flowgraph with a slightly different drawing convention: This time the arrows are missing and edges are understood to go downward. This convention is not limiting because all inputs to the passivation phase are dags. All nodes of this flowgraph read and write the variable of interest.

Figure 4.5(b) shows another flowgraph, using an even more complicated drawing convention: A dotted edge x ◓┈◓ y stands for a read-only node (◓) that has (normal) incoming edges from node x and from node y. So Figure 4.5(b) depicts a flowgraph with 7 nodes and 8 edges. Similarly, the third example in Figure 4.5(c) represents a flowgraph with 14 nodes (9 write-only and 5 read-only) and 18 edges. Without the dotted edge convention, Figure 4.5(c) would be much bigger and harder to grasp.

4.2 The Definition of Passive Form

The main purpose of this section is to give a precise formulation for the problem of finding a good passive form.

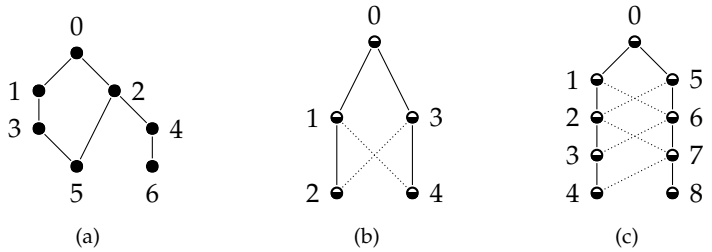


Figure 4.5: Various interesting special cases

```
1  procedure makeEven(u : int) returns (v : int) {
2    L0: v0 := u; goto L1, L2;
3    L1: assume !even(v0); goto L3;
4    L2: assume even(v0); goto L5;
5    L3: v1 := v0 + 1;
6    L4: assert even(v1); return;
7    L5: v1 := v0; goto L4;
8  }
```

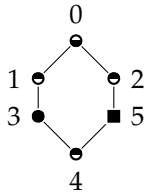


Figure 4.6: A passive form for the program in Figure 4.4

Example 3. A passive form of the program in Figure 4.4 appears in Figure 4.6. It is obtained by introducing *versions* 0 and 1 of the variable v and by inserting the copy statement 5.

In general, a *copy statement* (or *copy node*) has the shape $v_i := v_j$ and is drawn as a filled square (■). A copy node is a read-write node.

Example 3 contains only one variable and this is the case we shall analyze in detail. Multiple variables do not introduce any new complications: A program can be made passive with respect to each of its variables in turn, while considering the others to be constants.

Definition 21. A program is *passive* when on all execution paths each variable is written at most once.

Remark 5. Obtaining a passive form is similar to automatically deriving a functional equivalent of a loop-less imperative program. The remaining assignments, which are eventually transformed into assumptions, can be seen as **let** bindings.

The passive form of a program G' is an equivalent program G that is passive. If there are execution paths in program G' that write to variable v multiple times then those writes must be changed to write to distinct variables in program G. We denote the variables of program G by v_i where i is some integer and say that v_i is version i (of variable v). To maintain the semantics, each read from variable v must be replaced by a read from the latest written version. As Example 3 illustrates, it is necessary sometimes to alter the structure of the

program, and in current approaches [78, 30] this is always done by inserting copy statements of the form $v_i := v_j$. The following definition makes these notions precise.

Definition 22. A program G whose flowgraph has nodes V is a *passive form* of the program G' whose flowgraph has nodes V' when

1. program G is passive and
2. there exists a mapping $c : V' \to V$, a *write-version* function $w : V \to \mathbb{Z}$, and a *read-version* function $r : V \to \mathbb{Z}$ such that
   (a) *statement structure is preserved*: the statement $c(x)$ is obtained from statement x by replacing each occurrence of variable v by some variable v_i,
   (b) *the new nodes are copy statements*: all statements in $C = V - c(V')$ have the form $v_i := v_j$,
   (c) *the flow structure is preserved*: there is an edge $x \to y$ in program G' if and only if there is a path $c(x) \overset{Q}{\rightsquigarrow} c(y)$ in program G that uses only copy nodes as intermediate nodes, that is, $\big(Q - \{x, y\}\big) \subseteq C$; also, all copy nodes have at least one outgoing edge,
   (d) *the initial node is preserved*: $c(0) = 0$,
   (e) *writes and reads are confined*: each statement x in program G may only read version $r(x)$ and may only write version $w(x)$,
   (f) *the read version is always the latest written version*: $w(x) = r(y)$ if x is a write node, y is a read node, and there is a path $x \rightsquigarrow y$ in G that contains no intermediate write nodes.

Remark 6. The definition is fairly straightforward but it is important to have it written down. The definition naturally leads to two notions of optimality (which have been previously missed) and to a straightforward algorithm that is better than some previous solutions.

Example 4. If a program starts with **goto** $l_1, \ldots, l_n$ and continues with l_k : S; **return** for all k, then S is a passive form of it. Thus, the passive form may be smaller than the original program, although we will usually keep the mapping c injective.

It is easy to see that the passive form of a program G si correct if and only if program G is correct. The reason is that every read of a version by the passive form reads the same value as the corresponding read of the variable in program G, and every write of a version by the passive form writes the same value as the corresponding write to the variable in program G.

4.2.1 Types of Passive Forms

The requirement that program G is passive can be expressed as a constraint on the write-version function w.

Proposition 2. *If the write-version function w is a witness that program G is a passive form, then the existence of a path $x \rightsquigarrow y$ where both x and y are write nodes implies that $w(x) \neq w(y)$.*

A write node x writes to version $w(x)$. According to Definition 21, the same version must not be written by any other node on an execution path that includes node x.

Proposition 2 suggests that a "passive form" may be called more explicitly "distinct-version passive form." Definition 22 is very general. The passive forms obtained by previous approaches all satisfy a stronger definition that corresponds to the intuition that versions increase in time.

Definition 23. An *increasing-version passive form* is a passive form witnessed by a write-version function w with the property that $w(x) < w(y)$ whenever there is a path $x \rightsquigarrow y$ from a write node x to another write node y.

Programs have multiple passive forms, some better than others. There are two natural notions of optimality.

Definition 24. A passive form is *version-optimal* when it uses as few variable versions as possible.

Definition 25. A passive form is *copy-optimal* when it uses as few copy nodes as possible.

We say that *a version i is used by a passive form* when there is a write node whose write version is i. This convention simplifies the later analysis, but it requires some explanation. Why is it OK to ignore the read version of read nodes? The read version either is written by the program, in which case it was already counted as the write version of another node, or it is not written by the program. All versions that are not written are essentially equivalent. In other words, by ignoring the read versions of read nodes we undercount by one, if the uninitialized variable is read by the original program. That is a small price to pay for a reduction in the number of special cases that we must consider in the following analysis.

Let us summarize the information in Definitions 22, 23, 24, and 25. Each program G determines a set of (distinct-version) passive forms according to Definition 22; each program determines a set of increasing-version passive forms according to Definition 23. Each increasing-version passive form is a

distinct-version passive form. Each passive form has two associated costs—the number of versions and the number of copy nodes. Therefore, for each program there are four (not necessarily distinct) interesting classes of passive forms:

1. the version-optimal increasing-version passive forms,
2. the copy-optimal increasing-version passive forms,
3. the version-optimal distinct-version passive forms, and
4. the copy-optimal distinct-version passive forms.

One interesting problem is whether these four classes always overlap. In other words, is there always a passive form that is optimal by all measures and also follows the intuitive rule that versions do not decrease in time? But, before addressing this question, let us first look at how good a version-optimal passive form can be.

Lemma 1. *The number of versions of any passive form is greater or equal to the number of write nodes on any execution path in the original program.*

Proof. Consider an execution path P in the original program and its subset of write nodes $Q \subseteq P$. For two nodes $x, y \in Q$, we can assume that there is a path $x \rightsquigarrow y$. According to Proposition 2, their write versions are distinct. Hence, $|w(Q)| = |Q|$. □

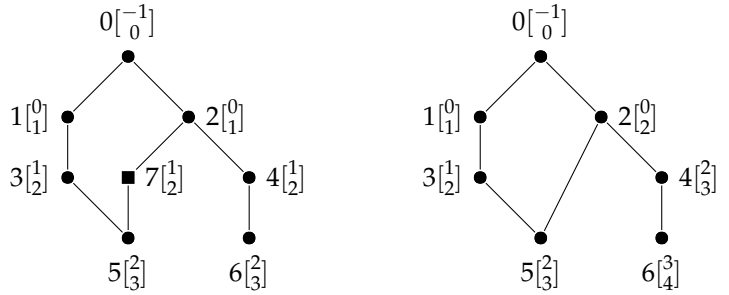
We can now show two facts about how different types of passive forms relate to each other—Proposition 3 and Proposition 4.

Proposition 3. *There are programs for which an increasing-version passive form cannot be both version-optimal and copy-optimal.*

Proof. One such program has the flowgraph depicted in Figure 4.5(a) on page 51. A version-optimal passive form appears in Figure 4.7(a) and a copy-optimal passive form appears in Figure 4.7(b). In these drawings, each node x is labeled with $x[{}^{r(x)}_{w(x)}]$. For example, the label $0[{}^{-1}_{0}]$ is put on node 0, which has read version -1 and write version 0.

It is easy to see that these solutions are optimal. According to Lemma 1 the number of versions in any passive form is ≥ 4 and Figure 4.7(a) uses 4 versions. Figure 4.7(b) uses 0 copy nodes.

It remains to show that there is no optimal increasing-version passive form for this program that uses 4 versions and 0 copy nodes. We do this by showing that all increasing-version passive forms that have no copy node must use ≥ 5 versions. According to Definition 23, $w(0) < w(1) < w(3)$ and $w(2) < w(4) < w(6)$. Because there are no copy operations, the Definition 22 gives us



(a) version-optimal (b) copy-optimal

Figure 4.7: Optimal increasing-version passive forms for Figure 4.5(a)

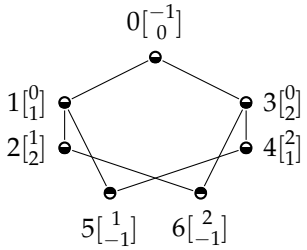


Figure 4.8: A copy-optimal passive form for Figure 4.5(b)

$w(3) = r(5) = w(2)$. Hence, the 5 used versions $w(0)$, $w(1)$, $w(3) = w(2)$, $w(4)$, and $w(6)$ are distinct. □

The second fact justifies the *unintuitive* choice to allow versions to decrease in time.

Proposition 4. *There are programs for which copy-optimal distinct-version passive forms use strictly fewer copy nodes than copy-optimal increasing-version passive forms.*

Proof. One such program has the flowgraph depicted in Figure 4.5(b) on page 51. A distinct-version passive form with no copy nodes appears in Figure 4.8. (Note that the two nodes represented earlier with dotted lines are now explicit.)

It remains to show that there is no increasing-version passive form than uses no copy node. Suppose that there is one. Then, $w(1) = w(4)$ and $w(2) = w(3)$. But Definition 23 tells us that $w(1) < w(2)$ and $w(3) < w(4)$. Contradiction. □

```
READ(y)        // memoized
1  r := −1
2  for each predecessor x of y
3        r := max(r, Write(x))
4  return r

WRITE(y)       // memoized
1  r := Read(y)
2  if y is a write node
3        r := r + 1
4  return r

VERSIONOPTIMALPASSIVEFORM(G)
1  for each node x of G
2        substitute reads of v by reads from version Read(x)
3        and writes of v by writes to version Write(x)
4  for each edge x → y of G
5        if Write(x) ≠ Read(y)
6              create a new node z := (v_Read(y) := v_Write(x))
7              remove the edge x → y
8              add edges x → z and z → y
```

Figure 4.9: A practical algorithm for finding a passive form

4.3 The Version-Optimal Passive Form

4.3.1 The Algorithm

Proposition 4 says that it is beneficial to allow versions to decrease in time if we are looking for a copy-optimal passive form. This section shows that this is not the case if the objective is a version-optimal passive form. The proof is constructive—an algorithm that finds a version-optimal increasing-version passive form that is as good as any passive form can be.

The algorithm from Figure 4.9 is *the best practical solution for the passivation problem*. It is very similar to the algorithm of Flanagan and Saxe [78], being changed so that it always finds a version-optimal passive form. The method *VersionOptimalPassiveForm*(G) modifies the program G into one of its passive forms. (The real implementation in FreeBoogie transforms the program without mutating it, as was explained in Section 3.3.3.)

Let us see, slowly, why this algorithm is correct. The mapping $c : V' \to V$ from nodes in the original program to nodes in its passive form is $c(x) = x$. The method $Read(y)$ computes the read-version $r(y)$, but only for nodes y that exist in the original graph; similarly, the method $Write(y)$ computes the write-version $w(y)$, but only for nodes y that exist in the original graph. With a

different notation, the read-version function and the write-version function are defined to be

$$r(y) = \begin{cases} \max_{x \to y} w(x) & \text{if } y \text{ has predecessors} \\ -1 & \text{otherwise} \end{cases}, \tag{4.15}$$

$$w(y) = r(y) + [\text{node } y \text{ is a write node}]. \tag{4.16}$$

If we unfold r,

$$w(y) = [y \text{ writes}] + \begin{cases} \max_{x \to y} w(x) & \text{if } y \neq 0 \\ -1 & \text{if } y = 0 \end{cases}, \tag{4.17}$$

it becomes clear that $w(y) + 1$ is the maximum number of write nodes on a path $0 \rightsquigarrow y$. It follows that $r(y)$ and $w(y)$ computed by (4.15) and (4.16) are in the range $-1 \,..\, n-1$, where n is the maximum number of write nodes on an execution path. Moreover, $w(y)$ can be -1 only if y is not a write node. So far we know that for write nodes y in the original graph, $w(y)$ is in the range $0 \,..\, n-1$.

The output graph also has copy nodes z created on line 6 of *VersionOptimalPassiveForm*. Each such node corresponds to an edge $x \to y$ in the original graph and has $w(z) = r(y)$. Hence, the version written by copy nodes is also in the range $0 \,..\, n-1$. (If $r(y) = -1$ then $w(x') = -1$ for all predecessors x' of y, including x, so there would be no copy node created.)

It follows immediately from Lemma 1 that no passive form has a number of versions $< n$, so we proved that if the output of *VersionOptimalPassiveForm* is indeed a passive form then it must be version-optimal. Most of the conditions imposed by Definition 22 on page 52 are easy to check.

1. *Passive:* The write version increases at every write node, so it cannot be the same as the write version of any preceding node.

2. (a) *Statement structure is preserved*: The statement structure is only modified by lines 2–3.

   (b) *The new nodes are copy statements*: New nodes are only created on line 6.

   (c) *The flow structure is preserved*: Edges are modified only on lines 7–8. Replacing $x \to y$ by $x \to z \to y$ maintains paths, because node z has no other adjacent edges.

   (d) *The initial node is preserved*: We have $c(x) = x$ for all nodes x in the original graph.

   (e) *Writes and reads are confined*: For existing nodes, see lines 2–3. For

each copy node we can simply choose the read version and the write version appropriately.

(f) *The read version is always the latest written version*: For each edge $x \to y$ we have $w(x) = r(y)$. This is true even for edges adjacent to copy nodes. Also $w(x) = w(y)$ if node y is not a write node. If there is a path $x_0 \to x_1 \to \cdots \to x_n \to x_{n+1}$, and $x_1, \ldots, x_n$ are not write nodes then $w(x_0) = w(x_1) = \cdots = w(x_n) = r(y)$.

We proved the following theorem.

Theorem 1. *The algorithm in Figure 4.9 constructs a version-optimal (distinct-version) passive form, which is also an increasing-version passive form.*

The algorithm is fast. The first loop of *VersionOptimalPassiveForm* (lines 2–3) is executed once for every node x in V; the second loop (line 5) is executed $|E|$ times. The substitutions performed on lines 2 and 3 can be done in time proportional to the AST size $|x|$ of the statement x. Because of memoization, the bodies of *Read* and *Write* are executed at most $|V|$ times each. One way to memoize the two methods (that is, to cache their results) is to have two integer fields in the data-structure representing a flowgraph node. A sentinel value, say -2, would represent 'not yet computed', while any other value would mean that the computation was already done. These two memory cells per node are the only significant memory used by the algorithm, apart from the memory used to represent the input and the output.

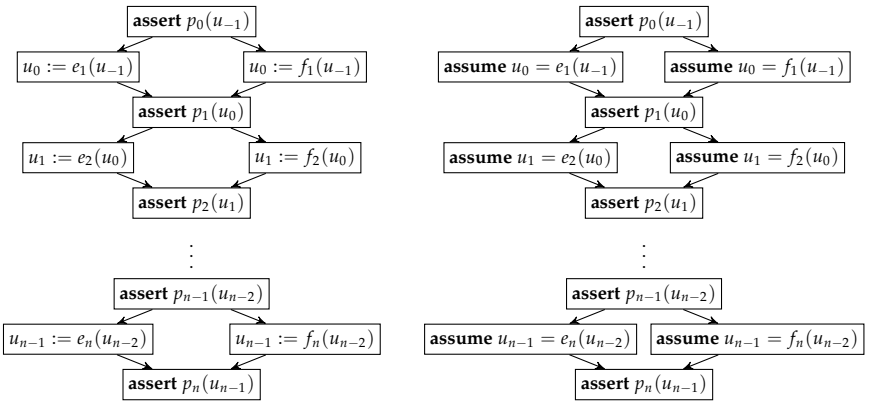
Proposition 5. *The algorithm in Figure 4.9 on page 56 uses $\Theta(|E| + \sum_{x \in V} |x|)$ time and $\Theta(|V|)$ temporary space.*

If the data-structure for representing a flowgraph node is immutable, then memoization can be implemented using a hashtable, in which case the time bound in Proposition 5 is true with high probability, but not always. Also, the constant hidden by the space bound is bigger.

The input occupies $\Theta(|E| + \sum_{x \in V} |x|)$ space and only $O(|E| + \sum_{x \in V} |x|)$ extra space can be allocated in $\Theta(|E| + \sum_{x \in V} |x|)$ time, so the output must occupy, asymptotically, the same amount of space as the input. In fact, the difference in size is accounted entirely by the copy nodes.

Proposition 6. *The algorithm in Figure 4.9 creates $O(|E|)$ copy nodes.*

Example 5. If we apply this algorithm to the program in Figure 4.1 we obtain the result in Figure 4.10(a), which is both copy-optimal (since it has no copy nodes) and version-optimal (by Theorem 1). It is now trivial to get rid of assignments: Just replace each assignment $u := e$ by the assumption **assume** $u = e$ (see



(a) passive form (b) no assignments form

Figure 4.10: Diamonds that do not explode

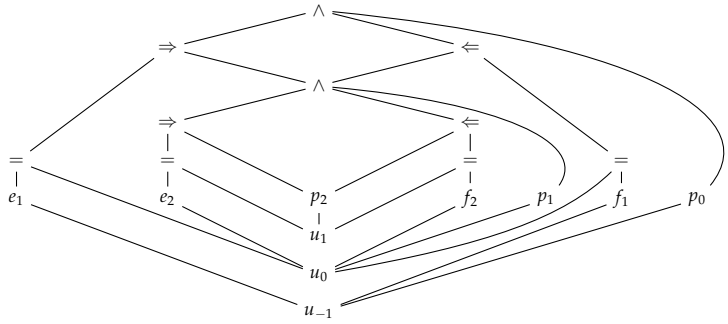


Figure 4.11: The weakest precondition of two diamonds (see Figure 4.1)

Figure 4.10(b)). The size of the weakest precondition of the initial node is now linear in the size of the program. The weakest precondition consists of (1) the expressions present in the program, (2) at most one new $\wedge$ for each assertion, (3) at most one new $\Rightarrow$ for each assumption, and (4) at most one new $\wedge$ for each flowgraph node with multiple successors. Figure 4.11 shows the weakest precondition SMT tree for the case $n = 2$.

The passive form computed for Example 5 happened to be copy-optimal. In general, this is not the case. In fact, sometimes the algorithm introduces clearly redundant copy nodes. For example, suppose that a read node x has three predecessors with the write-versions 0, 0, and 1, respectively. The algorithm presented so far will insert two copy nodes $v_1 := v_0$ between the predecessors with write-version 0 and node x, but only one is enough—copy nodes can be fused as long as the flowgraph structure is preserved (condition (c) of Definition 22 on page 52). It is easy to modify the algorithm such that copy nodes inserted before a given node x do not repeat. For example, one could keep a hashtable

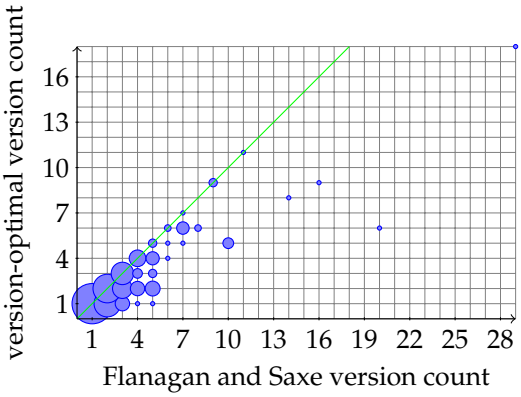


Figure 4.12: Version count experimental comparison

that maps the triple (node x, version i, version j) to a copy node $v_i := v_j$ that gets reused if it is already in the set. (The technique is similar to hash-consing.)

4.3.2 Experiments

Flanagan and Saxe [78] did not try to produce an optimal passive form. Still, their algorithm is very similar to the one presented here, so it is interesting to see how it compares.

Microsoft Research released a set of Boogie programs [165] to serve as a basis for comparing program verifiers. The algorithm of Flanagan and Saxe, however, does not handle the **goto** statement, which is used extensively in the Boogie benchmark. Luckily, it turns out that the **goto** statement is used in an interesting way only in 16% of the implementations in the Boogie benchmark. In the other 1070 implementations, the flowgraphs are series–parallel, which means that they correspond directly to programs that use only sequential composition and nondeterministic choice for flow control. The FreeBoogie implementation of the algorithm of Flanagan and Saxe simply refuses to give an answer if the flowgraph is not series–parallel.

Figure 4.12 compares the results of the two algorithms. A variable for which Flanagan and Saxe introduce m versions when n versions are enough contributes to the disc centered at point (m, n). The more variables are in this situation, the bigger the disc. (Its radius varies logarithmically.) The plot summarizes the result of passivating 1177 variables.

For 70% of the variables in the Boogie benchmark both algorithms say that one version is enough. For the other 30% variables the algorithm of Flanagan and Saxe introduces on average 46% more versions than needed.

It is interesting to note that for randomly generated flowgraphs the difference

between the two algorithms is much bigger: Both algorithms say that only one version is needed in less that 1% of situations, while for the others the algorithm of Flanagan and Saxe introduces on average 160% more versions than needed. The generator of random Boogie programs is part of FreeBoogie. It works, roughly, by generating random series–parallel flowgraphs by choosing graph grammar productions randomly and then deciding for each node independently with some probability whether it is a write node and whether it is a read node.

4.4 The Copy-Optimal Passive Form

We saw that finding a version-optimal passive form is rather easy. In contrast, finding a copy-optimal passive form seems rather hard. This is more evidence that the algorithm in Section 4.3 should be the algorithm of choice in practice.

4.4.1 The Increasing-Version Case

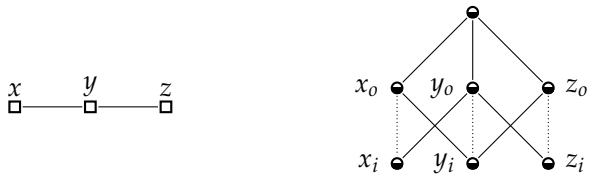
This section proves that finding a copy-optimal increasing-version passive form is NP-complete. The proof is by transformation from the MINS problem (which is defined in Section 4.1.3). Let us see, on the example in Figure 4.13, how to find a maximum independent set:

1. *Transform the MINS instance into a flowgraph.* A node x is transformed into the write only nodes x_i and x_o, the read only node x_r, and edges $0 \to x_o$ and $x_o \to x_r$ and $x_i \to x_r$. (The two incoming edges of node x_r are drawn in Figure 4.13 as x_o ◒┈◒ x_i .) An edge x □—□ y is transformed into edges $x_o \to y_i$ and $y_o \to x_i$.

2. *Find a copy-optimal increasing-version passive form of the flowgraph.* We do this by invoking an oracle.

3. *Transform the passive form into a set I of nodes in the original graph.* The set I contains nodes x for which the corresponding node $c(x_r)$ is *not* preceded by a copy node in the passive form.

To understand why this procedure indeed finds a maximal independent set let us focus on step 3. Clearly, it defines a function from passive forms to sets of nodes in the original graph. Since the definition of a passive form (Definition 22 on page 52) does not prevent redundant copy nodes, all nodes $c(x_r)$ might be preceded by copy nodes and the corresponding set I needs not be independent. The optimal passive form, however, will be among those passive forms without redundant copy nodes. We say that a passive form witnessed by a write version function w is *non-redundant* or that it *has no redundant copy nodes* when



(a) MINS instance (b) flowgraph

Figure 4.13: Transformation from MINS

it contains exactly one copy node before each node $c(x_r)$ with $w(c(x_i)) \neq w(c(x_o))$. It is easy to see that this implies $r(c(x_r)) = \max(w(c(x_i)), w(c(x_o)))$, and that the copy nodes are necessary and sufficient.

The image of a non-redundant passive form is an independent node set I. Assume, by contradiction, that the set I contains the adjacent nodes x and y. Then $w(c(x_i)) = w(c(x_o))$ and $w(c(y_i)) = w(c(y_o))$. The edges $x_o \to y_i$ and $y_o \to x_i$ imply paths $c(x_o) \rightsquigarrow c(y_i)$ and $c(y_o) \rightsquigarrow c(x_i)$ which in turn imply that $w(c(x_o)) < w(c(y_i))$ and $w(c(y_o)) < w(c(x_i))$. We reached a contradiction, as desired.

Conversely, every independent node set I is the image of some increasing-version non-redundant passive form. We construct such a non-redundant passive form as follows. Use the mapping $c(x) = x$. For $x \in I$ set $w(x_o) = w(x_i) = 2$; for $x \notin I$ set $w(x_o) = 1$ and $w(x_i) = 3$. If two nodes x and y are adjacent then we must have $w(x_o) < w(y_i)$ and $w(y_o) < w(x_i)$, which are true because at least one of the nodes x and y is not in the set I.

Moreover, a non-redundant passive form with $n - k$ copy nodes corresponds to an independent set of size k, where n is the number of nodes in the original graph. Hence, the copy-optimal increasing-version passive form corresponds to the maximum independent node set, which suggests the following theorem.

Theorem 2. *The problem of finding a copy-optimal increasing-version passive form is NP-complete.*

To complete the proof of this theorem we must formulate the transformation in terms of the corresponding decision problems and we must show that the decision problem corresponding to finding copy-optimal increasing-version passive forms is in NP.

The decision problem corresponding to the MINS problem asks whether there is an independent set of $\geq k$ nodes. The decision problem corresponding to the problem of finding a copy-optimal increasing-version passive form for the restricted family of flowgraphs illustrated in Figure 4.13(b) asks whether there is a copy-optimal increasing-version passive form with $\leq n - k$ copy nodes. To

transform one into the other we simply make sure we use the same k.

Also, the problem is in NP because we can check quickly if a flowgraph is a passive form of another, provided we are also given the mapping c, the write function w and the read function r.

4.4.2 The Distinct-Version Case

The problem seems difficult even if we drop the increasing-version restriction.

Conjecture 1. *The problem of finding a copy-optimal distinct-version passive form is NP-complete.*

The conjecture is that if we enlarge the search space (from increasing-version passive forms to distinct-version passive forms) the problem remains just as hard. In general, enlarging the set of feasible solutions can make an optimization problem easier or harder. As a trivial example, consider the problem of finding the maximum of each of these sets:

$$S_1(n) = \{\, 2 \,\} \tag{4.18}$$

$$S_2(n) = \{\, k \mid 0 < k < n \text{ and } k \text{ is prime} \,\} \tag{4.19}$$

$$S_3(n) = \{\, k \mid 0 < k < n \,\} \tag{4.20}$$

It is clear that $S_1(n) \subset S_2(n) \subset S_3(n)$ when $n > 2$, yet finding the maximum of the set $S_2(n)$ is hardest. On the other hand, if the input is restricted, then the problem cannot become harder.

Figure 4.14 illustrates a family of flowgraphs for which finding a copy-optimal passive form is equivalent to certain known classic problems. It is the example in Figure 4.5(c) plus decreasing adjacency lists added on the right. The flowgraphs in this family consist of two chains of write only nodes plus some read only nodes (depicted as dotted edges) that have one parent from the left chain and one parent from the right chain. The adjacency lists represent the dotted edges. For simplicity, let us also assume that all the statements have distinct structures so that the mapping c must be injective. Since c is injective we can be lazy and write x instead of $c(x)$ when $c(x)$ is not a copy node. We shall refer to flowgraphs in this restricted class as *two-chain flowgraphs*.

Finding a copy-optimal distinct-version passive form in such a flowgraph is equivalent to the bipartite matching problem. A dotted edge is selected if and only if the write versions of its endpoints are equal. As before, we focus on non-redundant passive forms, whose copy nodes are in one-to-one correspondence with non-selected dotted edges. Hence, selecting a maximum of edges corresponds to inserting a minimum of copy nodes.

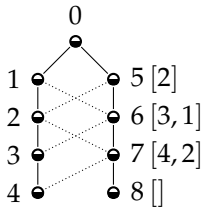


Figure 4.14:

In the bipartite matching problem, two edges can be selected simultaneously if and only if they do not share an endpoint. This is equivalent to the condition that write versions for the nodes in one chain are distinct. If, by way of contradiction, two edges $x \cdots z$ and $y \cdots z$, which share an endpoint, are both selected, then $w(x)$ must equal $w(y)$, which cannot happen since nodes x and y are in the same chain. Conversely, if two edges $x \cdots x'$ and $y \cdots y'$ do not share any endpoint then nodes x and x' can be assigned one write version and nodes y and y' a different version. So they can be both selected.

According to Section 4.1.3, the best known algorithms for the bipartite matching problem work in $O(\min(m\sqrt{n}, n^{2.38}))$ time, where m is the number of dotted edges and n is the number of nodes. A lower bound would be more useful towards justifying Conjecture 1. Unfortunately, none is known. The equivalence to the bipartite matching problem does tell us, however, that without finding a better algorithm for this classic problem we cannot hope to find a copy-optimal distinct-version passive form as fast as we can find a version optimal one.

It is also interesting to note that solving two-chain flowgraphs under the increasing-version restriction is equivalent to another classic problem, that of finding the longest increasing subsequence. Again, a dotted edge is selected if and only if its endpoints have the same write version. This time, the write versions in the left chain have to be increasing, and so do those in the right chain. This is equivalent to saying that two dotted edges can be selected if and only if they do not intersect, endpoints included.

Now take a look at Figure 4.14. The nodes in the left chain are renumbered so they are also increasing (like their write versions). The nodes in the right chain are annotated with the list of nodes that they can reach through one dotted edge. Each number in those lists represents a dotted edge. For example, the number 3 in the list next to node 6 represents the edge $3 \cdots 6$. The task is now to select at most one number from each adjacency list such that their sequence increases. Two numbers from the same list cannot be simultaneously selected because they correspond to dotted edges that intersect at their right endpoints.

Similarly, non-increasing numbers from different adjacency lists correspond to intersecting dotted edges. See for example the edges represented by number 3 in the adjacency list of node 6 and by number 2 in the adjacency list of node 7.

If we now concatenate the lists and then extract a longest increasing subsequence we risk selecting more than one number from an adjacency list. To make sure this does not happen, before concatenation, we sort each adjacency list in decreasing order.

Note that all operations used to transform the two-chain flowgraph into a sequence of numbers (renumbering nodes, sorting adjacency lists, concatenating them) can be done in linear time. Note also that computing the inverse transformation can also be done in linear time. We conclude that the problem of finding a copy-optimal increasing-version passive form is equivalent to LIS, in the strong sense that asymptotic bounds on the running time apply to both.

We managed to solve the two-chain family of flowgraphs faster in the increasing-version case than in the distinct version case. This is a small piece of evidence supporting Conjecture 1. However, keep in mind that (1) restricting the input may make an optimization problem harder *or* easier and (2) in a proof we should be looking at lower bounds instead of best known upper bounds.

4.5 Conclusions

Passivation follows loop cutting and precedes the computation of the VC. It increases the size of the program but its output will never lead to exponentially large VCs.

The precise definition of passivation (in Section 4.2) enables a formal study of its properties. The definition leads to four classes of passive forms that are good, each in its own way (Section 4.2.1). The subsequent sections are concerned with the complexity of finding a passive form: Version-optimal passive forms are easy to find, copy-optimal increasing-version passive forms are hard to find. A few open problems remain.

Problem 4. *What is the complexity of finding a copy-optimal distinct-version passive form?*

In other words, settle Conjecture 1.

Problem 5. *Find an approximation algorithm for the problem of finding a copy-optimal increasing-version passive form. It should run in $o(n^2)$ time to be practical.*

Problem 6. *Is it always possible to find a distinct-version passive form that is both version-optimal and copy-optimal? If yes, then how? If no, then what kind*

of trade-off can be achieved?

These problems are clearly inspired by the practice of developing a program verifier, yet they are very algorithmic and puzzle-like in nature.

4.6 Related Work

Flanagan and Saxe [78] observed that programs without assignments never lead to exponential VCs. They proposed passivation to avoid exponential explosion altogether and implemented it in ESC/Java. The intermediate language of ESC/Java, inspired by Dijkstra's guarded commands [70], does not have a **goto** statement, which makes the task of finding a passive form easier. Their method for VC generation was proved correct in Coq by Vogels et al. [171].

The passive form used for program verification resembles the DSA (**d**ynamic **s**ingle **a**ssignment) form used for a long time in the compiler community to ease optimizations. Sadly, the literature on compiler optimizations is largely disconnected from the literature on program verification. For example, in 2007 Vanbroekhoven et al. [169] identified copy operations as the key to reducing the size of the DSA. Flanagan and Saxe used copy operations six years earlier. Another example is the treatment of arrays. In the program verification area it is standard to model them using a pair of uninterpreted functions, usually named *update* and *select*. Once this is done, arrays are handled like all other variables during passivation.

Barnett and Leino [30] describe the VC generation done by the Boogie tool. The passivation stage is described informally. Since Boogie has **goto** statements, arbitrary flowgraphs are handled.

Flanagan and Saxe [78], Vanbroekhoven et al. [169] as well as Barnett and Leino [30] do not give a formal definition of what it means for a program to be a passive form of another; Vogels at al. [171] does give such a definition, which was developed independently from the one used in this dissertation [87]. Flanagan and Saxe [78], Barnett and Leino [30], and Vogels et al. [171] do not define what it means for a passive form to be optimal. Barnett and Leino [30] do discuss optimality, briefly. Vanbroekhoven et al. [169] employs a postprocessing heuristic that reduces the number of copy operations introduced by their passivation algorithm.

The algorithm presented in Section 4.3 should be compared with a textbook algorithm for coloring comparability graphs [85]. Tries, used in Section 4.1.1 when analyzing the size of the VC, were introduced by Fredkin [81]. Section 4.1.2 briefly enumerates a few basic definitions from computational complexity, adapted mostly from Goldreich [84]. The importance of the class of

NP-complete problems was established by Cook [60], who proved that there exist an NP-complete problem (boolean satisfiability), and by Karp [107], who proved that 21 other problems are NP-complete. Section 4.1.2 uses the terms 'transform' and 'reduce' like Knuth [113].

The solution to the longest increasing subsequence was presented in 1977 by Hunt and Szymanski [101] and it exploits a fast data structure presented by van Emde Boas et al. [168] the same year. An $O(m\sqrt{n})$ algorithm for bipartite matching was given by Hopcroft and Karp [99] in 1973. Faster algorithms for dense graphs are much newer (see, for example, Harvey [93]).

Chapter 5

Strongest Postcondition versus Weakest Precondition

"Given a propositional theory T and a proposition q, a sufficient condition of q is one that will make q true under T, and a necessary condition of q is one that has to be true for q to be true under T. In this paper, we propose a notion of strongest necessary and weakest sufficient conditions."
— Fangzhen Lin [128]

The main contributions of this chapter are (1) a comparison of the weakest precondition method to the strongest postcondition method, (2) links between different ways to define the semantics of subsets of Boogie (operational semantics, Hoare logic, weakest precondition, strongest postcondition), and (3) an algorithm for unsharing expressions.

5.1 Hoare Logic for Core Boogie

After passivation, FreeBoogie generates a VC using either a method based on the weakest precondition predicate transformer or on the strongest postcondition predicate transformer. The relation between these two methods is illuminated by their relation to a Hoare logic for Boogie.

5.1.1 Relation to Operational Semantics

The Hoare triple $\{p\}\ x\ \{r\}$ means that "if the store before executing statement x satisfies predicate p then (1) the execution of statement x does not go wrong and (2) if statement x is executed then the resulting store satisfies predicate r."

$$\begin{aligned}\{p\}\ x\ \{r\} = \forall \sigma,\ p\ \sigma \Rightarrow \\ \left(\forall \frac{h}{\langle \sigma, x\rangle \rightsquigarrow \mathit{error}},\ \neg h\right) \wedge \left(\forall \frac{h}{\langle \sigma, x\rangle \rightsquigarrow \langle \sigma', \_\rangle},\ h \Rightarrow r\ \sigma'\right)\end{aligned} \tag{5.1}$$

The quantifications over rules are perhaps a little unusual. The first one $\forall \frac{h}{\langle \sigma, x\rangle \rightsquigarrow \mathit{error}}$ goes over all rules that apply to the state $\langle \sigma, x\rangle$ and go to the *error* state. The hypothesis h is bound by the quantifier. The second quantification $\forall \frac{h}{\langle \sigma, x\rangle \rightsquigarrow \langle \sigma', \_\rangle}$ is similar, except that it ranges only over rules that go to non-*error* states. For example, if statement x is **assert** q, then the first quantification is instantiated only for rule (2.7) (on page 14) and the second quantification is instantiated only for rule (2.6).

$$\begin{aligned}\{p\}\ \textbf{assert}\ q\ \{r\} &= \forall \sigma,\ p\ \sigma \Rightarrow (q\ \sigma \wedge (q\ \sigma \Rightarrow r\ \sigma)) \\ &= \forall \sigma,\ p\ \sigma \Rightarrow (q\ \sigma \wedge r\ \sigma) \\ &= \forall \sigma,\ (p \Rightarrow (q \wedge r))\ \sigma \\ &= |p \Rightarrow (q \wedge r)|\end{aligned} \tag{5.2}$$

For the **assume** statement the first quantifier over rules evaluates to $\top$ since no rule for **assume** statements goes to the *error* state.

$$\{p\}\ \textbf{assume}\ q\ \{r\} = |(p \wedge q) \Rightarrow r| \tag{5.3}$$

The Hoare triple $\{p\}$ **assert** $q\ \{r\}$ holds when the predicate $p \Rightarrow (q \wedge r)$ is valid; the Hoare triple $\{p\}$ **assume** $q\ \{r\}$ holds when the predicate $(p \wedge q) \Rightarrow r$ is valid; and the Hoare triple $\{p\}\ v := e\ \{r\}$ holds when the predicate $p \Rightarrow (v \leftarrow e)\ r$ is valid. There is an appealing similarity between the predicates corresponding to **assert** and, respectively, **assume**.

5.1.2 Correctness of a Flowgraph

Before executing the initial statement of a program any store is possible. In other words, the set of statements possible before executing the initial statement is described by the predicate $\top$. The set of possible stores immediately after the initial statement may be restrained. For example, if the initial statement is **assume** q, then the set of possible stores after executing it is described by

the predicate q, because for other stores the execution does not proceed. In general, the set of all normal executions determines a set of possible stores before and after each statement. More precisely, for each statement x we define two predicates a_x and b_x:

$$a_x\ \sigma = \text{the state } \langle \sigma, x \rangle \text{ belongs to an execution} \tag{5.4}$$

$$b_x\ \sigma = \text{the states } \langle \_, x \rangle, \langle \sigma, \_ \rangle \text{ belong to an execution} \tag{5.5}$$

It follows immediately from this definition that $b_x \Rightarrow a_y$ is valid if there is an edge $x \to y$. In a correct program $\{a_x\}\ x\ \{b_x\}$ holds for all statements x.

Theorem 3. *A core Boogie program is correct when it is possible to attach to each statement x a precondition a_x and a postcondition b_x such that*

1. *the precondition a_0 of the initial node is valid,*
2. *$\{a_x\}\ x\ \{b_x\}$ holds, for all statements x, and*
3. *$b_x \Rightarrow a_y$ is valid, for all edges $x \to y$ in the flowgraph.*

Proof. We already saw that if a program is correct then we can find predicates a_x and b_x such that all the three conditions in this theorem hold. The converse, which is not yet clear, is that for incorrect programs it is impossible to find predicates a_x and b_x for all statements that satisfy the above conditions. Let $\langle \sigma_0, 0 \rangle, \langle \sigma_1, 1 \rangle, \ldots, \langle \sigma_n, n \rangle, \mathit{error}$ be an execution that goes wrong. Let us assume, by way of contradiction, that we found predicates satisfying the conditions above. Then we can prove by induction that $a_k\ \sigma_k$ holds for $k \in 0..n$. Initially $a_0\ \sigma_0$ holds because of condition 1. Condition 2 and $a_k\ \sigma_k$ imply $b_k\ \sigma_{k+1}$ and because of condition 3 we have $a_{k+1}\ \sigma_{k+1}$. We have now reached a contradiction because $a_n\ \sigma_n$ and condition 2 imply that the next state is not *error*. □

5.2 Predicate Transformers

We discuss now two methods of finding the preconditions a_x and postconditions b_x for each statement x. Both methods are complete, in the sense that if there exist predicates a_x and b_x that witness the correctness of a program according to Theorem 3 then they will be found.

The strongest postcondition method proceeds forwards: The precondition a_x is found before the postcondition b_x and, when there is an edge $x \to y$, the postcondition b_x is found before the precondition a_y. The weakest precondition

method proceeds backwards: The postcondition b_x is found before the precondition a_x and, when there is an edge $x \to y$, the precondition a_y is found before the postcondition b_x.

The strongest postcondition method always finds the strongest predicate that could possibly satisfy the conditions of Theorem 3; the weakest precondition method always finds the weakest predicate that could possibly satisfy the conditions of Theorem 3. We say that predicate p is stronger than predicate q (and that predicate q is weaker than predicate p) when the predicate $p \Rightarrow q$ is valid.

For the rest of this section it helps to think of predicates as sets of stores. The predicate $p \wedge q$ denotes the intersection set $p \cap q$; the predicate $p \vee q$ denotes the union set $p \cup q$; the predicate $\neg p$ denotes the complement set $\bar{p}$; the predicate $p \Rightarrow q$ denotes the set $\bar{p} \cup q$. The predicate $p \Rightarrow q$ is valid when $p \subseteq q$. Weakest means largest; strongest means smallest.

5.2.1 Dealing with Edges

In the strongest postcondition method, the precondition a_y is calculated after all postconditions b_x of nodes x with edges $x \to y$. For all nodes x we must have $b_x \Rightarrow a_y$. The strongest predicate a_y that is implied by all predicates b_x is their disjunction.

$$a_y \equiv \bigvee_{x \to y} b_x \tag{5.6}$$

The precondition of the initial node is not constrained by condition 3 (on flow-graph edges) but by condition 1.

$$a_0 \equiv \top \tag{5.7}$$

Note that (5.7) is not a special case of (5.6).

In the weakest precondition method, the postcondition b_x is calculated after all preconditions a_y of nodes y with edges $x \to y$. For all nodes y we must have $b_x \Rightarrow a_y$. The weakest predicate b_x that that implies all predicates a_y is their conjunction.

$$b_x \equiv \bigwedge_{x \to y} a_y \tag{5.8}$$

The postconditions of nodes with no outgoing edge, which correspond to **return** statements, are not constrained by condition 3. In fact they are not constrained by any of the conditions of Theorem 3 so for them $b_x \equiv \top$, the weakest possible predicate. Fortunately, that is what (5.8) reduces to for **return** nodes.

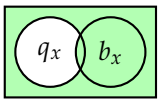


Figure 5.1: Weakest precondition of **assume** statements

5.2.2 Dealing with Statements

Assumptions If the statement x is **assume** q_x, then according to condition 2 of Theorem 3 the predicate

$$(a_x \wedge q_x) \Rightarrow b_x \tag{5.9}$$

must be valid.

In the strongest postcondition method we must find the strongest predicate b_x, given the predicates a_x and q_x. Clearly,

$$b_x \equiv (a_x \wedge q_x). \tag{5.10}$$

In the weakest precondition method we must find the weakest predicate a_x, given the predicates q_x and b_x. In terms of sets, we must find the biggest set a_x such that its intersection with the set q_x is included in the set b_x.

$$a_x \equiv (q_x \Rightarrow b_x) \tag{5.11}$$

The situation is depicted in Figure 5.1.

Assertions If the statement x is **assert** q_x, then according to condition 2 of Theorem 3 the predicate

$$a_x \Rightarrow (q_x \wedge b_x) \tag{5.12}$$

must be valid.

In the strongest postcondition method we must find the strongest predicate b_x, given the predicates a_x and q_x. No matter what predicate b_x we choose, the predicate in (5.12) is invalid unless

$$a_x \Rightarrow q_x \tag{5.13}$$

is valid. If this condition holds, then the strongest predicate b_x that satisfies (5.12) is

$$b_x \equiv (a_x \wedge q_x). \tag{5.14}$$

In the weakest precondition method we must find the weakest predicate a_x,

given predicates q_x and b_x.

$$a_x \equiv (q_x \wedge b_x) \tag{5.15}$$

Assignment If statement x is $v := e$, then according to condition 2 of Theorem 3 the predicate

$$a_x \Rightarrow (v \leftarrow e)\ b_x \tag{5.16}$$

must be valid.

In the strongest postcondition method we must find the strongest predicate b_x, given the predicate a_x, the variable v, and the expression e. We rewrite condition (5.16).

$$\forall \sigma,\ a_x\ \sigma \Rightarrow b_x\ ((v \leftarrow e)\ \sigma) \tag{5.17}$$

Note that $(v \leftarrow e)\ \sigma = (v \leftarrow e)\ \sigma'$ does not imply $\sigma = \sigma'$. That is, there might be multiple stores σ corresponding to the same right hand side in (5.17). Because we want the set b_x to be as small as possible we want $b_x\ \sigma$ to hold only if it must hold, that is, only if there is some corresponding left hand side in (5.17) that holds.

$$b_x\ \sigma = \exists \sigma',\ a\ \sigma' \wedge \left(\sigma = (v \leftarrow e)\ \sigma'\right) \tag{5.18}$$

It is possible to rewrite this with a quantification over values instead of the quantification over stores (see (4.12)), but since any expression using the existential quantifier is useless in practice (because of the limitations of the theorem prover) it makes little sense to spend more time refining this equation.

In the weakest precondition method we must find the weakest predicate a_x, given the variable v, the expression e, and the predicate b_x. This is trivial.

$$a_x \equiv (v \leftarrow e)\ b_x \tag{5.19}$$

5.2.3 Summary

In the strongest postcondition method we compute

$$a_y \equiv \begin{cases} \top & \text{if } y \text{ is the initial statement} \\ \bigvee_{x \to y} & \text{otherwise} \end{cases} \tag{5.20}$$

$$b_x \equiv \begin{cases} a_x \wedge q_x & \text{if } x \text{ is } \textbf{assert}/\textbf{assume}\ q_x \\ \lambda\sigma.\ \exists \sigma',\ a\ \sigma' \wedge \left(\sigma = (v \leftarrow e)\ \sigma'\right) & \text{if } x \text{ is } v := e \end{cases} \tag{5.21}$$

and we check the validity of

$$vc_{sp} \equiv \bigwedge_{\substack{x \text{ is an} \\ textassertion}} (a_x \Rightarrow b_x). \tag{5.22}$$

In the weakest precondition method we compute

$$b_x \equiv \bigwedge_{x \to y} a_y \tag{5.23}$$

$$a_x \equiv \begin{cases} q_x \Rightarrow b_x & \text{if } x \text{ is } \textbf{assume } q_x \\ q_x \wedge b_x & \text{if } x \text{ is } \textbf{assert } q_x \\ (v \leftarrow e)\ b_x & \text{if } x \text{ is } v := e \end{cases} \tag{5.24}$$

and we check the validity of

$$vc_{wp} \equiv a_0. \tag{5.25}$$

Equations (5.21) and (5.24) can be seen as defining the predicate transformer *sp* and, respectively, the predicate transformer *wp*. Both take two arguments, a statement and a predicate, and return a predicate.

Example 6. Let us look again at the program in Figure 4.6 on page 51. The two methods described above yield equivalent but structurally different VCs:

$$\begin{aligned} vc_{sp} \equiv & \big((\top \wedge (v_0 = u) \wedge \neg even(v_0) \wedge (v_1 = v_0 + 1)) \\ & \vee (\top \wedge (v_0 = u) \wedge even(v_0) \wedge (v_1 = v_0))\big) \\ & \Rightarrow even(v_1) \end{aligned} \tag{5.26}$$

$$\begin{aligned} vc_{wp} \equiv & (v_0 = u) \Rightarrow \\ & \big((\neg even(v_0) \Rightarrow (v_1 = v_0 + 1) \Rightarrow (even(v_1) \wedge \top)) \\ & \wedge (even(v_0) \Rightarrow (v_1 = v_0) \Rightarrow (even(v_1) \wedge \top))\big) \end{aligned} \tag{5.27}$$

5.3 Replacing Assignments by Assumptions

After passivation all assignments $v := e$ are transformed into assumptions **assume** $v = e$. This section argues briefly why this transformation is sound and complete. The key idea is that there is a one-to-one correspondence between the executions of the program without assignments and executions of the program with assignments where the initial store is chosen so that all writes are no-operations.

It is complete: If the program without assignments has an execution that goes

wrong then the program with assignments has an execution that goes wrong. Say $\langle \sigma, x_0 \rangle$, $\langle \sigma, x_1 \rangle$, ..., $\langle \sigma, x_n \rangle$, *error* is an execution of the program without assignments. If statement x_k is **assert**/**assume** q_k then $q_k\ \sigma$ holds. In particular, if the statement corresponds to an assignment $v := e$ then it is **assume** $v = e$, which means that $\sigma\ v = e\ \sigma$. It is easy to see now that a similar execution that goes wrong exists in the program with assignments, because $\sigma = (v \leftarrow e)\ \sigma$.

The proof of the converse is very similar and we omit it. The idea is to take an execution that goes wrong in the program with assignments and show that the final store in this execution can serve as the (unique) store in an execution of the assignment-free program.

The proofs above essentially rely on the existence of executions in which no assignment changes the store. Such executions exist because the program is passive. The advantages of the assignment-free form are presented in Section 4.1.1.

5.4 Verification Condition Size

5.4.1 VCs with Sharing

Figure 5.2 shows (5.20), (5.21), and (5.22) as an algorithm. It is assumed that the program contains only **assert** and **assume** statements. The call $Predicate(x)$ returns the predicate in statement x. The calls $Or(ps)$, $And(ps)$, and $Implies(p, q)$ build predicates out of simpler ones. They are builders of SMT terms, which implement hash-consing as described in Section 3.6. The set *bs* defined on line 3 of method *Pre* uses reference equality, which implies structural equality due to hash-consing.

Example 7. The VC in (5.26) is represented by the SMT term in Figure 5.3. Notice the sharing of a subtree. Enclosed in rectangles are expressions that appear in the Boogie program (from Figure 4.6 on page 51). These are translated from the Boogie AST representation to the SMT representation as explained in Section 3.6. The circled nodes are created by the algorithm in Figure 5.2 by calling the builders *True*, *And*, *Or*, and *Implies*.

Remark 7. FreeBoogie builds a slightly simpler SMT tree because builders carry out simplifications. For example, the call $And(\{p, \bot\})$ returns p.

Proposition 7. *The algorithm in Figure 5.2 computes the VC in $O(|V| + |E|)$ time. If the program contains no assertions then the lower bound $\Omega(|V|)$ is attained.*

Proof. Because of memoization the methods *Pre* and *Post* are called at most once for each node. Inserting a term in the set *bs* takes $O(1)$ time if the set is

```
PRE(y)        // memoized
1  if y is initial
2        return True()
3  bs := ∅        // collects all b_x
4  for each predecessor x of y
5        insert Post(x) in the set bs
6  return Or(bs)

POST(x)       // memoized
1  return And(Pre(x), Predicate(x))

VC()
1  vs := ∅        // collects local VCs
2  for each statement x that is an assertion
3        insert Implies(Pre(x), Predicate(x)) in the set vs
4  return And(vs)
```

Figure 5.2: VC computation using the strongest postcondition method

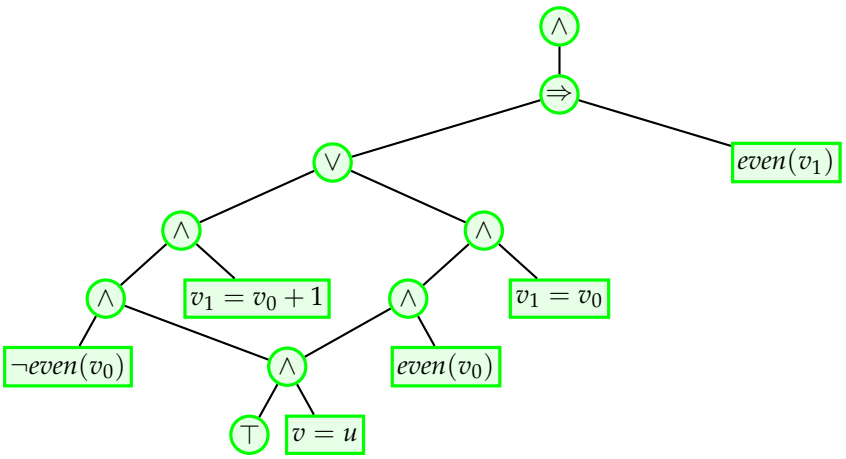


Figure 5.3: Data structure for the VC in (5.26)

implemented as a hashtable. The method *Pre* analyzes all the incoming edges of the node it was called for, so in the worst case it looks once at each edge in the flowgraph. If there are no assertions then the runtime is dominated by the loop in *Vc*, which checks there are no assertions. □

Note that the upper bound on the execution time would be slightly worse if randomized algorithms (such as hashtables) are forbidden.

Proposition 8. *The algorithm in Figure 5.2 computes a VC with size* $O(|V| + |E| + n)$*, where* n *is the space needed to represent all the expressions that appear in the program. If the program contains no assertions then the lower bound* $\Omega(1)$ *is attained.*

Proof. Each execution of the method *Pre* creates one new node that contains as many links to children as there are predecessors in the flowgraph. Each execution of the method *Post* creates one new node with two children, one of which is an expression from the program. If there are no assertions then $vc_{sp} \equiv \bot$. □

Remark 8. Results similar to Proposition 7 and Proposition 8 hold for the analogous implementation of the weakest precondition method.

5.4.2 VCs without Sharing

The next step is to send the VC to a theorem prover. One way is to implement the builder methods (like *True*, *Or*, ...) in terms of calls to the prover API. Another way is to communicate with the prover through the SMT language. In the second approach, a data structure is built inside the VC generator, then it is serialized as a string, and then the prover parses the string and builds its own data structures. Hence, the second approach is slower and uses more memory. Its main advantage is that multiple provers understand the SMT language. If the VC is printed in the SMT language, then it could be used as a benchmark for multiple provers. Of course, another advantage is that the backend of the VC generator does not have to deal with multiple APIs, some of which might not be for Java.

SMT terms cannot be printed recursively in the naive way, for this may result in strings that are exponentially large in the size of the data structure. Extra work is needed to identify the shared parts and to eliminate the sharing. Finding which parts are shared is useful also for deciding how to split the VC into multiple queries, in case it is too big.

Figure 5.4 shows a simplified version of the algorithm used in FreeBoogie. The method *EliminateSharing* transforms an SMT term, which represents a

predicate, into an equivalent one that has less sharing. The algorithm plucks sub-predicates repeatedly. Figure 5.5 illustrates the plucking process: It replaces the predicate $(v \leftarrow q)\ p$ by the predicate $(v = q) \Rightarrow p$. Each plucking preserves validity.

$$|(v \leftarrow q)\ p| = |(v = q) \Rightarrow p| \tag{5.28}$$

Suppose $(v \leftarrow q)\ p$ is valid and pick a store σ that satisfies the predicate $v = q$.

$$\begin{aligned}
&(v \leftarrow q)\ \sigma\ u && && (5.29)\\
&= \begin{cases} \sigma\ u & \text{if } u \text{ is } v \\ q\ \sigma & \text{if } u \text{ is not } v \end{cases} && \text{definition of } (v \leftarrow q) && (5.30)\\
&= \sigma\ u && \text{we assumed } \sigma\ v = q\ \sigma && (5.31)
\end{aligned}$$

So $(v \leftarrow q)\ \sigma = \sigma$.

$$\begin{aligned}
&(v \leftarrow q)\ p\ \sigma && \text{we assume } |(v \leftarrow q)\ p| && (5.32)\\
&= p\ ((v \leftarrow q)\ \sigma) && \text{definition of } (v \leftarrow q) && (5.33)\\
&= p\ \sigma && \text{because } (v \leftarrow q)\ \sigma = \sigma && (5.34)
\end{aligned}$$

We have proved

$$|(v \leftarrow q)\ p| \Rightarrow |(v = q) \Rightarrow p| \tag{5.35}$$

The key to prove the other implication is to note that $(v \leftarrow q)\ (v = q)$ is valid. Therefore, plucking preserves validity, even though the predicates $(v \leftarrow q)\ p$ and $(v = q) \Rightarrow p$ do not always describe the same set of stores. Note also that there is no need to constrain the predicate q to be independent of variable v, although this is the case in the unsharing algorithm.

Example 8. If $p \equiv (v \Rightarrow p_2)$ and $q \equiv p_1$ then the claim is

$$|p_1 \Rightarrow p_2| = |(v = p_1) \Rightarrow (v \Rightarrow p_2)|. \tag{5.36}$$

The method *Unshare* collects the definitions for all fresh variables and the method *EliminateSharing* uses them to form the result. In other words we need to prove

$$\big|(v_1 \leftarrow q_1)\ ((v_2 \leftarrow q_2)\ p)\big| = \big|\big((v_1 = q_1) \wedge (v_2 = q_2)\big) \Rightarrow p\big| \tag{5.37}$$

which generalizes easily to more fresh variables. Here is a proof.

$$
\begin{aligned}
& \left|(v_1 \leftarrow q_1)\ ((v_2 \leftarrow q_2)\ p)\right| && && (5.38)\\
= & \left|(v_1 = q_1) \Rightarrow ((v_2 \leftarrow q_2)\ p)\right| && \text{by (5.28)} && (5.39)\\
= & \left|(v_2 \leftarrow q_2)\ ((v_1 = q_1) \Rightarrow p)\right| && \text{see below} && (5.40)\\
= & \left|(v_2 = q_2) \Rightarrow (v_1 = q_1) \Rightarrow p\right| && \text{by (5.28)} && (5.41)\\
= & \left|\big((v_1 = q_1) \wedge (v_2 = q_2)\big) \Rightarrow p\right| && \text{boolean algebra} && (5.42)
\end{aligned}
$$

The second step of this proof holds only if the predicate q_1 does not depend on the variable v_2, which is true if the variable v_2 does not syntactically appear within the predicate q_1. With n variables, the condition is that predicate q_i does not contain variable v_j if $i < j$. It is always possible to find such an ordering since variables, which are created on line 10 of the method *Unshare*, may only syntactically contain in their definition variables that were created earlier.

Apart from being correct, the algorithm in Figure 5.4 is also good because it does not use much time or space.

The method *CountParents* is executed once for each node in the input (except for line 1, which is executed once for each edge in the input, and line 2, for which a similar bound holds). Because it is memoized, the method *Unshare* is also executed once for each node in the input. For each execution, it creates at most three new nodes (lines 6, 10, and 11). The method *PrintSize* is executed at most once for each new node. Therefore the algorithm runs in linear time and creates at most three times as many nodes as there are in the input. (Some of the new nodes 'created' on line 6 of the method *Unshare* are taken from the hash-consing cache.)

The constant k on line 9 of the method *Unshare* controls how much plucking is done. A big value reduces the number of plucks; a small value increases the number of plucks. The value $ps \times pc$ estimates the length of printing without plucking, and $ps + pc$ estimates the length of printing with plucking. In particular, if we pick $k = -1$, then plucking is done if $ps > 1$ and $pc > 1$. In this case, only nodes without children (leafs) may have multiple parents in the result. The print size of such a dag is linear in its size, which in turn is linear in the size of the input. In other words, if $k = -1$ then the printing size of *EliminateSharing*(t) is linear in the size of t.

We have shown the following.

Theorem 4. *The call EliminateSharing(t) returns in $O(n)$ time and uses $O(n)$ space, where n is the size of the dag t. The resulting dag represents a predicate that is valid if and only if the predicate represented by the dag t is valid. Moreover, when $k = -1$, the print size of the result is $O(n)$.*

```
1  global newDefinitions      // set of (v ⟺ t)
2  global parentCount         // term t has parentCount[t] parents
3  global seen
```

```
ELIMINATESHARING(t)
1  clear globals
2  CountParents(t)
3  t := Unshare(t)
4  return Implies(And(newDefinitions), t)
```

```
COUNTPARENTS(t)
1  if t ∈ seen
2       return
3  insert t in the set seen
4  for each child c of t
5       increment parentCount[c]
6       CountParents(c)
```

```
PRINTSIZE(t)       // memoized
1  s := 1
2  for each child c of t
3       s := s + PrintSize(c)
4  return s
```

```
UNSHARE(t)       // memoized
1  if t or one of its children is not a formula
2       return t
3  newChildren := empty list
4  for each child c of t
5       append Unshare(c) to newChildren
6  t' := mk(type(t), newChildren)
7  ps := PrintSize(t')
8  pc := parentCount[t]
9  if ps × pc − (ps + pc) > k       // k is a constant
10      v := fresh variable
11      insert Iff(v, t') in the set newDefinitions
12      t' := v
13 return t'
```

Figure 5.4: (Almost) eliminating sharing

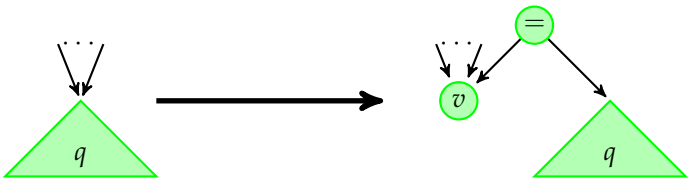


Figure 5.5: Plucking

The implementation in FreeBoogie is slightly more complicated, but produces queries that the theorem prover Simplify answers using 30% less time than it needs for the queries produced by the algorithm described so far. This reduction in time was measured on the benchmark used in Chapter 7.

Look again at Example 8 on page 78. Equation (5.36) is a special case of the "proof by indirect inequality law:"

$$|p_1 \Rightarrow p_2| = |(v \Rightarrow p_1) \Rightarrow (v \Rightarrow p_2)|. \tag{5.43}$$

It turns out that

$$|(v \leftarrow q)\ p| = |(v \Rightarrow q) \Rightarrow p| \tag{5.44}$$

holds when the predicate p is monotonic in the variable v

$$\big|(q_1 \Rightarrow q_2) \Rightarrow \big((v \leftarrow q_2)\ p \Rightarrow (v \leftarrow q_1)\ p\big)\big| \tag{5.45}$$

and also

$$|(v \leftarrow q)\ p| = |(q \Rightarrow v) \Rightarrow p| \tag{5.46}$$

holds when

$$\big|(q_1 \Rightarrow q_2) \Rightarrow \big((v \leftarrow q_1)\ p \Rightarrow (v \leftarrow q_2)\ p\big)\big|. \tag{5.47}$$

FreeBoogie detects monotonicity by examining a syntactic condition. A predicate p satisfies (5.47) if variable v syntactically occurs only in positive positions. Roughly speaking, a position is positive if it is under an even number of negations.

5.5 Experiments

Figure 5.6 shows the ratios between the proving times required for weakest precondition and strongest postcondition. The tests were ran using the Z3 v1.2 prover on a single core of a dual-core AMD Opteron 2218 (2×2.6 GHz) with 16 GiB of DDR2-667 RAM. Both methods perform similarly, as can be seen by the relative symmetry of the graph. Weakest precondition does mildly better however, as evidenced by the slight shift to the right. It is interesting to note that in over 40% of the test cases the time ratio between the worse method and the better method is > 2. These are the cases outside of the shaded area.

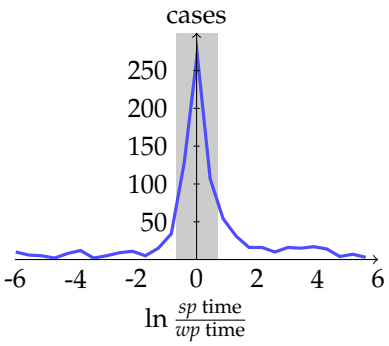


Figure 5.6: Experimental comparison of proving times

5.6 Conclusions

To reason about core Boogie programs we can use operational semantics (Chapter 2), Hoare logic (Section 5.1), and predicate transformers (Section 5.2). Hoare logic plays a central role, as it is easy to relate both to operational semantics and to predicate transformers. Hoare logic is also useful to characterize correctness of Boogie programs in a very intuitive way (Theorem 3).

The weakest precondition predicate transformer and the strongest postcondition predicate transformer power two different methods for generating VCs. The resulting VCs are equivalent (in the sense that one is valid if and only if the other is), but may have very different structures. The structure influences the theorem proving time in unexpected ways and it turns out to be worth trying both approaches in parallel if two processing units are available (Section 5.5).

The generated VC must be sent to a theorem prover. There is a simple and efficient algorithm for translating the in-memory representation of the VC into a standard language like SMT (Section 5.4). However, the in-memory representation is sometimes already too large.

Problem 7. *Exploit the structure of vc_{wp} and vc_{sp} in order to split big VCs into smaller ones.*

The VC produced by the strongest postcondition method seems to be especially amenable to splitting since it already has the proper structure—a big conjunction. The main problem is how to best handle the shared parts of the conjuncts.

5.7 Related Work

ESC/Java can use both the strongest postcondition [78] and the weakest precondition method [120], but not for arbitrary acyclic flowgraphs. Note that [78] describes a method analogous to our strongest postcondition method but only uses the term "outcome predicate". Boogie uses only the weakest precondition method and can treat unstructured programs [30].

Strongest postcondition was discussed before in the context of a simple language similar to ours in relation to predicate abstraction [77] (but only for structured programs) and in relation to proof reuse [88] (but not shown sound).

Chapter 6

Edit and Verify

"No battle plan survives contact with the enemy."
— Helmuth Karl Bernhard Graf von Moltke

Previous chapters cover various stages of FreeBoogie's pipeline (Section 3.2). The problem addressed by this chapter—speeding up the verification condition generation by exploiting the incremental nature of software development and verification—requires us to look again at the pipeline as a whole. The main contributions are (1) a proof technique for algorithms that simplify a VC by using an old VC as a reference, (2) a heuristic for detecting common parts of two expression trees, such as two VCs, and (3) a prototype implementation, which is part of the SMT solver Fχ7.

6.1 Motivation

Humans prefer swift tools: A response time $\gtrsim 0.1$ seconds is noticeable, and a response time $\gtrsim 8$ seconds entices context switches [98]. For example, during a long compilation a programmer might start browsing the Internet. The context switch from programming to browsing and back induces a significant loss of productivity.

Three techniques are especially useful in improving the performance of compilers. The first technique, background compilation, makes the perceived response time smaller than the processing time. The second technique, modular compilation, makes sure that time is spent mostly on the modified modules. A module is typically a file or a class. Modules still need to be linked, which is done by a second (faster) step for languages like C and by the virtual machine at run-time for languages like Java. The third technique, incremental compilation, is similar to modular compilation, but with a finer granularity. Typically

individual methods are recompiled separately.

Eclipse uses all three techniques. Background compilation is implemented in most IDEs; modular compilation is implemented in most compilers; incremental compilation is not yet implemented in some widely used compilers, such as GCC.

A core Boogie program corresponds to a procedure implementation in full Boogie. The correctness of a Boogie procedure implementation depends only on its preconditions and postconditions, and on those of the called procedures. It is easy and natural to verify a Boogie program one procedure implementation at a time. If we would mirror the compiler terminology, then we would say that Boogie was designed for incremental verification. However, the term 'modular verification' is more often used.

Java was designed to be compiled one class at a time; Boogie was designed to be verified one procedure implementation at a time. Java now has incremental compilers, which compile parts of a class separately. Is it possible to verify only parts of a Boogie procedure implementation? Could this improve the performance of a program verifier? These are the questions that motivate the investigations of this chapter.

6.2 Overview

Figure 6.1 shows 16 annotated programs, one for each possible choice on whether to include each of the four shaded lines. The program that includes no shaded line is the original program. Inserting one shaded line represents an edit operation. In general, an *edit operation* is any change from a type-correct Boogie program to another type-correct Boogie program. Programmers and IDEs invoke program verifiers periodically. For each invocation, the frontend transforms Java into Boogie, the backend transforms Boogie into a VC, and the theorem prover tries to determine if the VC is valid. Between two subsequent invocations (on type-correct programs) there is usually not much change, especially when the program verifier is invoked continuously in the background by an IDE. Moreover, many transformations are local. Therefore, much work is duplicated. As an example, consider the following scenario: A program verifier is run on the original program of Figure 6.1, then line 1 is added, and then the program verifier is run again. By adding line 1 nothing essential changes, so the first and the second run of the program verifier take the same amount of time. It should be possible to do much better.

FreeBoogie consists almost entirely of evaluators, some of which are transformers (see Section 3.3). The input of an evaluator is a Boogie AST. Because

```
1  // comment
2  abstract class Day {
3    public abstract int getMonth();
4      ensures 1 <= result && result <= 12;
5
6    public abstract int getYear();
7      ensures 1970 <= result;
8      ensures result <= 2038;
9
10   public abstract int getDay();
11     ensures 1 <= result && result <= 31;
12
13   public int dayOfYear()
14     ensures 1 <= result;
15     ensures result <= 366;
16   {
17     int offset = 0;
18     if (getMonth() > 1) offset += 31;
19     if (getMonth() > 2) offset += 28;
20     if (getMonth() > 3) offset += 31;
21     if (getMonth() > 4) offset += 30;
22     if (getMonth() > 5) offset += 31;
23     if (getMonth() > 6) offset += 30;
24     if (getMonth() > 7) offset += 31;
25     if (getMonth() > 8) offset += 31;
26     if (getMonth() > 9) offset += 30;
27     if (getMonth() > 10) offset += 31;
28     if (getMonth() > 11) offset += 30;
29     boolean isLeap = getYear() % 4 == 0 &&
30                 (getYear() % 100 != 0 || getYear() % 400 == 0);
31     assert offset <= 334;
32     if (isLeap && getMonth() > 2) offset++;
33     return offset + getDay();
34   }
35 }
```

Figure 6.1: Typical evolution of annotated Java code

Boogie ASTs are immutable, they can be used as keys in a map (dictionary) data structure. That is why the class *Evaluator*, the base class of all evaluators, can cache the results of all evaluations done in FreeBoogie. For example, if the type-checker is invoked twice on the AST of a procedure implementation, then the second invocation returns quickly after one hashtable lookup.

Two problems remain. First, if the input file is parsed twice, then two distinct ASTs are created by the parser, thus rendering the caches unusable. Second, even if the VC is obtained faster, the component that takes too much time is usually the prover. Let us consider these problems in turn.

To not produce the same AST twice when line 1 is inserted we must not parse the input file, in its entirety, twice. We could either re-parse only the parts that change or we could not parse at all. Both solutions work. The easiest and fastest way to detect what parts changed is to integrate FreeBoogie's parser in a text editor, such as the text editor of Eclipse. Parsing is unnecessary when the frontend uses FreeBoogie's API.

To reduce the prover time without re-engineering it, we must simplify the VC. For example, if the VC turns out to be the same as a previous one, for which the prover was called, then there is nothing more to do. This is the case for inserting line 1. In particular, if the VC before change is valid, then the VC after change simplifies to the predicate $\top$. Any prover should return quickly when the query is the predicate $\top$.

6.3 Simplifying SMT Formulas

When SMT solvers like Z3 and FX7 are asked if predicate p is valid, they search for a counterexample store σ, one for which $(\neg p)\ \sigma$ holds. Consider the following Boogie program.

```
1 var y : int;
2 assume (forall x : int :: even(x) || odd(x));
3 assume !odd(y);
4 assert even(y);
```

The VC produced by the strongest postcondition method is

$$vc_{sp} \equiv \big((\forall x,\ even(x) \vee odd(x)) \wedge \neg odd(y)\big) \Rightarrow even(y) \tag{6.1}$$

and its negation is

$$\neg vc_{sp} \equiv (\forall x,\ even(x) \vee odd(x)) \wedge \neg odd(y) \wedge \neg even(y). \tag{6.2}$$

A store σ associates an integer to variable y. Because vc_{sp} contains quantifiers and uninterpreted functions, it is not easy to evaluate $(\neg vc_{sp})\ \sigma$. In this par-

ticular example, the prover might be lucky and instantiate the quantifier with $x = y$.

$$\left|\neg vc_{sp} \Rightarrow (even(y) \vee odd(y)) \wedge \neg odd(y) \wedge \neg even(y)\right| \tag{6.3}$$

$$\Rightarrow |\neg vc_{sp} \Rightarrow \bot| \tag{6.4}$$

$$\Rightarrow |vc_{sp}| \tag{6.5}$$

The search concludes and vc_{sp} is declared valid. Usually, when quantifiers are involved, the theorem prover cannot conclude with certainty that a store is a counterexample.

In short, the first thing an SMT solver does is to negate the query and to perform basic boolean simplifications on it. In FX7, the boolean operators remaining after simplification are $\wedge$, $\vee$, and $\neg$. The problem can now be rephrased as follows.

Problem 8. *Find a predicate transformer prune such that*

$$|\neg p| \Rightarrow \big(|\neg q| = |\neg(prune\ p\ q)|\big) \tag{6.6}$$

for all predicates p and q. For most SMT provers, the query prune p q should be easier than the query q.

For example, if $p \equiv q$, then $(prune\ p\ q) \equiv \bot$ is a feasible solution, because (6.6) reduces to $|\neg p| \Rightarrow |\neg p|$. Table 6.1 shows a more interesting example. If

$$p \equiv (p_1 \wedge p_2) \tag{6.7}$$

$$q \equiv \big((p_1 \wedge \neg p_2) \vee (p_1 \wedge p_2 \wedge \neg p_3)\big) \tag{6.8}$$

then

$$(prune\ p\ q) \equiv (p_1 \wedge p_2 \wedge \neg p_3) \tag{6.9}$$

is a feasible solution. Note that $(prune\ p\ q) \equiv (p_1 \wedge \neg p_3)$ also satisfies the requirements of Problem 8, but it is likely not easier to prove. Intuitively, the predicate p_2 should act as an intermediate lemma in proving the predicate p_3. To make the example concrete the reader might wish to plug in $p_k \equiv (v > -k)$ for $k \in \{1, 2, 3\}$.

6.3.1 Pruning Predicates

There are many possible solutions for Problem 8. The solution $(prune\ p\ q) \equiv q$ corresponds to the normal operation of program verifier—old VCs are forgotten. The following solution is designed to exploit the structure of VCs produced

Table 6.1: Example of simplification

	Initial	Modified	Simplified
	assume p_1	**assume** p_1	**assume** p_1
	assert p_2	**assert** p_2	**assume** p_2
		assert p_3	**assert** p_3
vc_{sp}	$p_1 \Rightarrow p_2$	$(p_1 \Rightarrow p_2) \wedge \big((p_1 \wedge p_2) \Rightarrow p_3\big)$	$(p_1 \wedge p_2) \Rightarrow p_3$
$\neg vc_{sp}$	$p_1 \wedge \neg p_2$	$(p_1 \wedge \neg p_2) \vee (p_1 \wedge p_2 \wedge \neg p_3)$	$p_1 \wedge p_2 \wedge \neg p_3$

using the strongest postcondition method.

$$prune\ p\ (p \wedge r) \equiv \bot \tag{6.10}$$

$$\begin{aligned} &prune\ \big((p_1 \wedge r_1) \vee \cdots \vee (p_m \wedge r_m)\big)\ \big((r_1 \wedge \cdots \wedge r_m) \wedge (q_1 \wedge \cdots \wedge q_n)\big) \\ &\quad \equiv (r_1 \wedge \cdots \wedge r_m) \\ &\qquad \wedge (prune\ (p_1 \vee \cdots \vee p_m)\ q_1) \\ &\qquad\ \vdots \\ &\qquad \wedge (prune\ (p_1 \vee \cdots \vee p_m)\ q_n) \end{aligned} \tag{6.11}$$

$$prune\ p\ (q_1 \vee \cdots \vee q_n) \equiv (prune\ p\ q_1) \vee \cdots \vee (prune\ p\ q_n) \tag{6.12}$$

$$prune\ p\ q \equiv q \tag{6.13}$$

One possible way to evaluate the predicate transformer *prune* on the predicates in Table 6.1 is the following.

$$\begin{aligned} &prune\ (p_1 \wedge \neg p_2)\ \big((p_1 \wedge \neg p_2) \vee (p_1 \wedge p_2 \wedge \neg p_3)\big) \\ &\equiv \big(prune\ (p_1 \wedge \neg p_2)\ (p_1 \wedge \neg p_2)\big) \vee \big(prune\ (p_1 \wedge \neg p_2)\ (p_1 \wedge p_2 \wedge \neg p_3)\big) \\ &\equiv \bot \vee \big(p_1 \wedge prune\ (\neg p_2)\ (p_2 \wedge \neg p_3)\big) \\ &\equiv p_1 \wedge (prune\ (\neg p_2)\ p_2) \wedge (prune\ (\neg p_2)\ (\neg p_3)) \\ &\equiv p_1 \wedge p_2 \wedge \neg p_3 \end{aligned}$$

In this evaluation, the earliest matching branch is always taken. For example, if both (6.10) and (6.11) match, then (6.10) is applied.

In the rest of this section, i ranges over $1..m$, and j ranges over $1..n$. With these conventions we can write, for example, $(\vee i,\ p_i)$ instead of $(\bigvee_{i \in 1..m} p_i)$. The definition of the predicate transformer *prune* is much shorter with these

notational conventions.

$$prune\ p\ (p \wedge r) \equiv \bot \tag{6.14}$$

$$\begin{aligned} &prune\ (\vee i,\ p_i \wedge r_i)\ ((\wedge i,\ r_i) \wedge (\wedge j,\ q_j)) \\ &\qquad \equiv (\wedge i,\ r_i) \wedge (\wedge j,\ prune\ (\vee i,\ p_i)\ q_j) \end{aligned} \tag{6.15}$$

$$prune\ p\ (\vee j,\ q_j) \equiv (\vee j,\ prune\ p\ q_j) \tag{6.16}$$

$$prune\ p\ q \equiv q \tag{6.17}$$

The following lemmas help establish the correctness of any pruning based on these equations.

Lemma 2. *The pruned predicate* $prune\ p\ q$*, as defined by* (6.14)*,* (6.15)*,* (6.16)*, and* (6.17)*, is stronger than predicate* q*:*

$$|(prune\ p\ q) \Rightarrow q| \tag{6.18}$$

Proof. The proof is by induction on the total size of the two arguments. In each case, σ is an arbitrary store.

Branch (6.14):

$$prune\ p\ (p \wedge r)\ \sigma \tag{6.19}$$

$$= \bot\ \sigma \qquad \text{by (6.14)} \tag{6.20}$$

$$\Rightarrow (p \wedge r)\ \sigma \tag{6.21}$$

Branch (6.15):

$$prune\ (\vee i,\ p_i \wedge r_i)\ \big((\wedge i,\ r_i) \wedge (\wedge j,\ q_j)\big)\ \sigma \tag{6.22}$$

$$= \big((\wedge i,\ r_i) \wedge (\wedge j,\ prune\ (\vee i,\ p_i)\ q_j)\big)\ \sigma \qquad \text{by (6.15)} \tag{6.23}$$

$$\Rightarrow \big((\wedge i,\ r_i) \wedge (\wedge j,\ q_j)\big)\ \sigma \qquad \text{by induction} \tag{6.24}$$

Branch (6.16):

$$prune\ p\ (\vee j,\ q_j)\ \sigma \tag{6.25}$$

$$= (\vee j,\ prune\ p\ q_j)\ \sigma \qquad \text{by (6.16)} \tag{6.26}$$

$$\Rightarrow (\vee j,\ q_j)\ \sigma \qquad \text{by induction} \tag{6.27}$$

Branch (6.17):

$$prune\ p\ q\ \sigma \tag{6.28}$$

$$= q\ \sigma \qquad \text{by (6.17)} \tag{6.29}$$

□

Lemma 3. *The pruned predicate* $prune\ p\ q$*, as defined by* (6.14)*,* (6.15)*,* (6.16)*, and* (6.17)*, is weaker than the predicate* $\neg p \wedge q$*:*

$$|(\neg p \wedge q) \Rightarrow (prune\ p\ q)| \tag{6.30}$$

Proof. The proof is by induction on the total size of the two arguments. In each case, σ is an arbitrary store.

Branch (6.14):

$$\begin{aligned}
&(\neg p \wedge (p \wedge r))\ \sigma && && (6.31)\\
&= \bot\ \sigma && \text{by (6.14)} && (6.32)\\
&\Rightarrow prune\ p\ (p \wedge r)\ \sigma && && (6.33)
\end{aligned}$$

Branch (6.15):

$$\begin{aligned}
&\big(\neg(\vee i,\ p_i \wedge r_i) \wedge (\wedge i,\ r_i) \wedge (\wedge j,\ q_j)\big)\ \sigma && && (6.34)\\
&\Rightarrow \big((\wedge i,\ r_i) \wedge \neg(\vee i,\ p_i) \wedge (\wedge j, q_j)\big)\ \sigma && && (6.35)\\
&= \big((\wedge i,\ r_i) \wedge (\wedge j,\ \neg(\vee i,\ p_i) \wedge q_j)\big)\ \sigma && && (6.36)\\
&\Rightarrow \big((\wedge i,\ r_i) \wedge (\wedge j,\ prune\ (\vee i,\ p_i)\ q_j)\big)\ \sigma && \text{by induction} && (6.37)\\
&= prune\ (\vee i,\ p_i \wedge r_i)\ ((\wedge i,\ r_i) \wedge (\wedge j,\ q_j))\ \sigma && \text{by (6.15)} && (6.38)
\end{aligned}$$

Branch (6.16):

$$\begin{aligned}
&\big(\neg p \wedge (\vee j,\ q_j)\big)\ \sigma && && (6.39)\\
&= (\vee j,\ \neg p \vee q_j)\ \sigma && && (6.40)\\
&\Rightarrow (\vee j,\ prune\ p\ q_j)\ \sigma && \text{by induction} && (6.41)\\
&= prune\ p\ (\vee j,\ q_j)\ \sigma && \text{by (6.16)} && (6.42)
\end{aligned}$$

Branch (6.17):

$$\begin{aligned}
&(\neg p \wedge q)\ \sigma && && (6.43)\\
&\Rightarrow q\ \sigma && && (6.44)\\
&= prune\ p\ q && \text{by (6.17)} && (6.45)
\end{aligned}$$

□

Theorem 5. *Computing a predicate using the predicate transformer* $prune$ *defined by* (6.14)*,* (6.15)*,* (6.16)*, and* (6.17) *terminates, and the resulting predicate*

prune p q satisfies

$$|\neg p| \Rightarrow (q \equiv (prune\ p\ q)) \tag{6.46}$$

for all predicates p and q.

Proof. Evaluating (6.14) and (6.17) terminates in one step. The other branches (6.15) and (6.16) decrease the total size of the two predicates taken as arguments. Therefore, the algorithm encoded by the predicate transformer *prune* terminates.

When $|\neg p|$ holds, Lemma 2 and Lemma 3 reduce to

$$|(prune\ p\ q) \Rightarrow q| \tag{6.47}$$

$$|q \Rightarrow (prune\ p\ q)| \tag{6.48}$$

from which (6.46) follows immediately. □

Corollary 1. *The predicate transformer prune defined by* (6.14), (6.15), (6.16), *and* (6.17) *is a feasible solution for Problem 8.*

A VC computed using the strongest postcondition method is a conjunction of implications, according to (5.22) (on page 74). Let us see how exactly is that structure exploited by the rules (6.14)–(6.17). First, the case when some assertions are added on top of the old ones.

$$vc_{old} \equiv (p_1 \Rightarrow q_1) \tag{6.49}$$

$$vc_{new} \equiv \big((p_1 \Rightarrow q_1) \wedge (p_2 \Rightarrow q_2)\big) \tag{6.50}$$

Then pruning proceeds as follows.

$$prune\ (\neg p_1 \wedge q_1)\ \big((\neg p_1 \wedge q_1) \vee (\neg p_2 \wedge q_2)\big) \tag{6.51}$$

$$\equiv prune\ (\neg p_1 \wedge q_1)\ (\neg p_1 \wedge q_1) \vee prune\ (\neg p_1 \wedge q_1)\ (\neg p_2 \wedge q_2) \tag{6.52}$$

$$\equiv \bot \vee (\neg p_2 \wedge q_2) \tag{6.53}$$

The main observation here is that all old assertions are compared with all new assertions thanks to (6.16). Second, the case where an assumption is added.

$$vc_{old} \equiv (p_1 \Rightarrow q_1) \tag{6.54}$$

$$vc_{new} \equiv \big((p_1 \wedge p_2) \Rightarrow q_1\big) \tag{6.55}$$

Then pruning proceeds as follows.

$$\begin{aligned} & \mathit{prune}\ (\neg p_1 \wedge q_1)\ (\neg p_1 \wedge \neg p_2 \wedge q_1) && (6.56)\\ \equiv\ & \neg p_1 \wedge q_1 \wedge \mathit{prune}\ \top\ (\neg p_2) && (6.57)\\ \equiv\ & \neg p_1 \wedge q_1 \wedge \bot && (6.58)\\ & && (6.59) \end{aligned}$$

The first step uses (6.15).

Other definitions of the predicate transformer *prune* might simplify VCs better. The general proof technique to check if alternative definitions of *prune* are correct is to check that they maintain the invariants (6.18) and (6.30).

6.3.2 Algorithm

Equations (6.14), (6.15), (6.16), and (6.17) do not make an algorithm. It is not clear which one to apply and it is not clear how the pattern matching on arguments should be done. Also, such a declarative description, although convenient for reasoning about correctness, makes it hard to reason about time and space efficiency, because it hides too many details.

Figure 6.2 gives more details. Because predicates are represented by SMT trees, which are hash-consed (Section 3.6.1), two predicates are compared for syntactic equality in constant time and they can be put in (hash) sets. In particular, the children of commutative operators such as $\wedge$ and $\vee$ constitute a set with at least two elements. This way, $p \vee q$ and $q \vee p$ are considered to be syntactically equivalent.

The conditions of the **if** statements on lines 1, 3, and 13 make it clear how branches are chosen. The syntactic condition on line 1 holds only if predicate p is weaker than predicate q; the syntactic conditions on lines 3 and 13 simply look at the root of the SMT tree q.

Example 9. If $p \equiv (v < 0)$ and $q \equiv (v > 0)$ then execution reaches line 15. Equation (6.16) matches if n is set to 1, but the condition on line 13 syntactically checks if the operator $\vee$ is present.

The bulk of method *Prune*, lines 4–9, pattern match the arguments according to (6.15). Line 4 unfolds predicate p into a set of sets; line 5 unfolds predicate q into a set. Line 6 determines the predicates that are denoted by r_i in (6.15). These are removed from P and Q in lines 7–9 and afterwards the sets in the set P correspond to the p_is in (6.15) and the set Q corresponds to the q_js in (6.15).

Because it closely follows (6.14), (6.14), (6.14), and (6.14), the algorithm in Figure 6.2 solves Problem 6.6. Its time and space usage is not immediately clear.

```
FLATTEN(op, t)        // memoized
1  if root of t is not op
2        return t
3  return ⋃_{t'∈children(t)} Flatten(op, t')

PRUNE(p, q)        // memoized
1  if Flatten(∧, p) ⊆ Flatten(∧, q)
2        return ⊥
3  if the root of q is ∧        // use (6.15)
4        P := { Flatten(∧, t) | t ∈ Flatten(∨, p) }
5        Q := Flatten(∧, q)
6        R := Q ∩ (⋃_{C∈P} C)
7        for each C in P
8              C := C − R
9        Q := Q − R
10       p := ⋁_{C∈P} ⋀_{p'∈C} p'
11       R := R ∪ (⋃_{q'∈Q} Prune(p, q'))
12       return ⋀_{r∈R} r
13 if the root of q is ∨        // use (6.16)
14       return ⋁_{q'∈Flatten(∨,q)} Prune(p, q')
15 return q        // fall back to (6.17)
```

Figure 6.2: Algorithm for pruning SMT trees

Consider the following family of predicates.

$$q_0 \equiv \big((u_0 \wedge \cdots \wedge u_n) \vee (v_0 \wedge \cdots \wedge v_n)\big) \tag{6.60}$$

$$q_{k+1} \equiv \big((u_k \wedge q_k) \vee (q_k \wedge v_k)\big) \qquad \text{for } k \in 0..n-1 \tag{6.61}$$

$$p \equiv q_0 \tag{6.62}$$

The call $Prune(p, q_n)$ will recursively call $Prune(q', q_0)$ with 2^n variants q' of the predicate q_0. For each $k \in 0..n-1$, the predicate q' contains either the variable u_k or the variable v_k.

However, the performance is usually much better. The method $Prune(p, q)$ calls itself recursively $Prune(\bullet, q')$ only for descendants q' of the SMT node q. Therefore, $Prune(\_, q')$ is called more than once only if there are multiple ways to descend from the SMT node q to the SMT node q'. If the SMT dag q has no sharing, then the method *Prune* is called at most once for each SMT node q' that is a descendant of the SMT node q. In other words, the number of executions of the body of the method *Prune* is at most the number of nodes in the SMT dag q.

Before being sent to the prover the VC is unshared. An SMT dag produced by the algorithm of Section 5.4.2 has little sharing, and with the proper setup its leafs are the only nodes that may have more than one parent. In such a case, the number of executions of the *Prune* method is at most the number of edges in

the SMT dag q.

Proposition 9. *If the SMT tree q does not have internal sharing (except perhaps for its leafs), then the call $Prune(p, q)$ of the method in Figure 6.2 uses $O(n)$ time and $O(n)$ space, where n is the size of the SMT tree q in memory. Moreover, the print size of the SMT tree $Prune(p, q)$ is at most the print size of the SMT tree q.*

6.4 Correspondence between Trees

Experiments showed that the algorithm of Figure 6.2 is useless by itself, because the SMT dags that represent the predicates p and q seldom share much. In particular, if ESC/Java is used to produce a VC from the code in Figure 6.1 (on page 86) with and without the comment line 1, then there is almost no sharing, despite hash-consing. The problem will still exist if instead of ESC/Java we will use FreeBoogie with a Java frontend.

The issue is better understood if we take a step back and look at the big picture. A frontend transforms the code in Figure 6.1 into a Boogie program; a VC generator transforms the Boogie program into an SMT query; an SMT solver answers the query. The interfaces between these software components are standardized at the language level—multiple static verifiers understand the Boogie language and multiple provers understand the SMT language. But, of course, information must flow in the other direction too, because the user expects to see a result.

How do we easily substitute an SMT solver for another if they report results in different ways? And how do we substitute a VC generator for another if they report results in different ways? Leino et al. [122] describe an elegant solution. The key insight is that no matter how an SMT solver presents a counter-example, it probably includes a set of identifiers. Similarly, no matter how a VC generator reports errors, it probably includes a set of identifiers. Therefore, extra information, like location, can be recovered from ornate identifiers. For example, ESC/Java decorates identifiers by their location, so that it knows to which occurrence of a certain identifier in the Java code the counter-example refers to. (Note that, since different versions are needed for passivation anyway, they can be obtained by attaching strings that are useful for error reporting too, instead of 1, 2, 3, ...)

If the frontend that produces Boogie from the example in Figure 6.1 decorates identifiers with location information, then inserting a comment on line 1 results in a Boogie program with completely different identifiers. Almost all comparisons done by the algorithm of Figure 6.2 return false, and no simplification is done.

6.4.1 Matching SMT dags

To solve this problem, FX7 improves the sharing in SMT dags before pruning. The nodes in an SMT dag are labeled with a string. Some labels, such as "and", "or", and "41", have a special meaning for the prover while others, such as those representing variables in predicates, are uninterpreted. The uninterpreted labels are also called identifiers, because the only semantic information they carry follows from whether two of them are equal as strings or not. It is safe to rename identifiers as long as their semantic information is preserved. To explain formally why this is the case we would need to dig more into the structure of predicates. However, there is an important special case that is easy to prove already. Note that

$$|\neg p| \Rightarrow |\neg((v \leftarrow e)\ p)| \tag{6.63}$$

for any substitution $(v \leftarrow e)$, because, for an arbitrary store σ,

$$\begin{aligned}
&\neg\ ((v \leftarrow e)\ p)\ \sigma && (6.64)\\
&= \neg((v \leftarrow e)\ p\ \sigma) && (6.65)\\
&= \neg(p\ ((v \leftarrow e)\ \sigma)) && (6.66)\\
&= (\neg p)\ ((v \leftarrow e)\ \sigma) && (6.67)\\
&\Leftarrow |\neg p|. && (6.68)
\end{aligned}$$

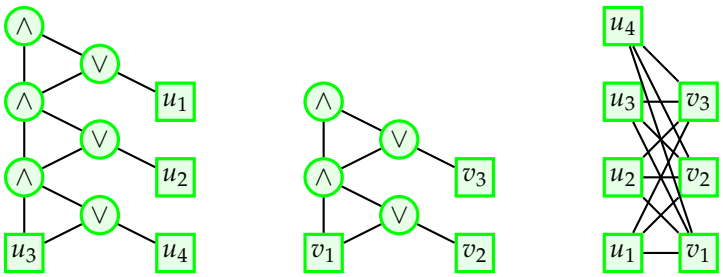
In other words, the correctness of pruning is not affected if the *variables* in predicate p are renamed beforehand, no matter what renaming is done.

The heuristic used by FX7 improves sharing by renaming only variables, and is easier to understand in a slightly more abstract setting.

Problem 9. *Given are two unordered rooted dags p and q. Their nodes and edges are labeled with strings. Find a one-to-one correspondence between some leafs of the dag p and some leafs of the dag q.*

Remark 9. Such labeled dags may represent both SMT dags and Boogie ASTs. Nodes could be labeled with strings such as "not", "or", and "for all". The edges below "not" and "or" would be labeled by the empty string; The edges below "for all" would be labeled with "bound variable" and, respectively, "body". However, the exact encoding is not important. What is important is that dags with labels on nodes and edges can easily encode both SMT dags and Boogie ASTs.

Problem 9 is an incompletely specified optimization problem: It is clear what a feasible solution is, but it is not clear which solutions are better and which are worse. A cost function that evaluates correspondences is missing. One possible



(a) dag p (b) dag q (c) bipartite graph

Figure 6.3: Dags and bipartite graph for Example 10

measure is the size of the hash-consed data structure that represents both dags p and q after corresponding leafs are glued. A cost function is a prerequisite to any analytic performance analysis. However, analytic performance analyses are likely to be very difficult to carry out here and likely to be not very relevant in practice. To name just one obstacle, an average case analysis requires VCs to live in a probability space, and it is not at all clear what probabilities accurately describe real VCs. It is therefore not very useful to fix a cost function. Instead, pragmatism points towards experimental algorithmics. In this case, objective experimental measures include the overall proving time (with and without improving sharing) and the size of the pruned VC (with and without improving sharing).

Intuitively, we expect any reasonable heuristic to work well in simple cases. For example, if two big VCs can be obtained one from the other by a one-to-one renaming of variables, then we expect a good heuristic to recover the necessary one-to-one renaming. This is the case with VCs generated from the program in Figure 6.1 (on page 86) with and without the comment on line 1, if the frontend decorates variables with location information. In general, a good heuristic matches what a human would do.

Example 10. The dags from Figure 6.3 represent two fairly big predicates. All edge labels are empty. Most humans asked to match the variables of dag p with the variables of dag q would match u_1 — v_3 and u_2 — v_2 and perhaps others. If the sharing in dags p and q would be eliminated (see Section 5.4) then these correspondences are harder to notice.

Because sharing benefits the search for identifier correspondences but hurts pruning, an implementation in FreeBoogie should

1. find correspondences between dags,
2. then eliminate sharing (Section 5.4.2),

3. and then prune the VCs (Section 6.3.1).

Approach We begin by constructing a complete bipartite graph—its left nodes are the leafs of the dag p and its right nodes are the leafs of the dag q. For each node u on the left and each node v on the right there is an edge u — v whose weight indicates how similar are nodes u and v. Then we find a maximum weight matching in this bipartite graph.

Any matching in a complete bipartite graph formed by the leafs of dag p and, respectively, the leafs of dag q is a one-to-one correspondence between some leafs of dag p and some leafs of dag q as Problem 9 requires. Which matching is chosen depends on the heuristic used to compute the edge weights. This heuristic is the knob we use to improve the results, while we keep the overall structure of the solution unchanged.

Example 11. Continuing Example 10, Figure 6.3(c) shows the complete bipartite graph that we build in the first step. Intuitively, we want the weights on edges u_1 — $v3$ and u_2 — v_2 to be relatively big, such that the second step selects those edges.

Similarity of Leafs The remaining problem is simpler. Given a leaf u in the dag p and a leaf v in the dag q, we must find a weight $w_{p,q}(u, v)$ which corresponds to the intuitive notion of similarity. The weight might be computed, for example, by looking at $label(u)$ and $label(v)$, the strings that label leafs u and v. But Example 10 suggests that the location of a leaf in the dag is important. The location of a leaf u in a labeled unordered dag p, denoted $location_p(u)$, is a multiset of string sequences, each string sequence corresponding to a path from leaf u to the root of dag p.

Definition 26. The multiset $location_p(u)$ contains the string sequence $label(v_1)$, $label(e_1)$, $label(v_2)$, $label(e_2)$, …, $label(v_n)$ once for each path $v_1 \overset{e_1}{\leftarrow} v_2 \overset{e_2}{\leftarrow} \cdots v_n$ from the leaf $u = v_1$ to the $root(p) = v_n$.

Note that $location_p(u)$ is a set if the dag p was built using hash-consing, that is, if structural equality is equivalent to reference equality.

It is intuitive to use the information from leaf labels independently from the information about the location of leafs.

$$w_{p,q}(u, v) = c(w_1(label(u), label(v)), w_2(location_p(u), location_q(v)) \tag{6.69}$$

Function w_1 measures the similarity of two strings, function w_2 measures the similarity of two multisets, and function c combines two similarities. Function c

should be increasing with respect to each of its arguments. Possible choices for function c include

$$c(w_1, w_2) = \alpha w_1 + \beta w_2 \qquad \text{for positive } \alpha \text{ and } \beta \tag{6.70}$$

$$c(w_1, w_2) = w_1 w_2 \qquad \text{for positive } w_1 \text{ and } w_2 \tag{6.71}$$

Possible choices for function w_1 include

- the longest common subsequence [101], $w_1(s,t) = lcs(s,t)$,
- the Jaro–Winkler [176] similarity,
- the inverse of a string distance, such as the edit distance [127], and
- the similarity of the multisets of labels' $n$grams, using various alternatives available for the function w_2.

Possible choices for function w_2 include

- the Jaccard similarity [129, Chapter 1], $w_2(A,B) = 1 - |A \cap B| / |A \cup B|$,
- Dice's coefficient [68], $w_2(A,B) = 2|A \cap B| / (|A| + |B|)$, and
- the opposite of a set distance, such as the Hamming distance [91].

(All these are easily adapted to multisets.)

FX7 uses

$$c(w_1, w_2) = w_1 + w_2 \tag{6.72}$$

$$w_1(s,t) = lcs(s,t) \tag{6.73}$$

$$w_2(A,B) = 2|A \cap B| - \big||A| - |B|\big| \tag{6.74}$$

No experiment revealed a situation where a correspondence found by a human was better than the one found by FX7. However, it is comforting to now that there is ample room for tuning the heuristic, in case better results are needed.

Locations within Dags In the worst case, the location of a leaf u in a dag p has size exponential in the number of nodes in the dag p.

Example 12. For dag p in Figure 6.3 (on page 97) we have

$$|location_p(u_1)| = 1 \tag{6.75}$$

$$|location_p(u_2)| = 2 \tag{6.76}$$

$$|location_p(u_4)| = 4 \tag{6.77}$$

$$|location_p(u_3)| = 8 \tag{6.78}$$

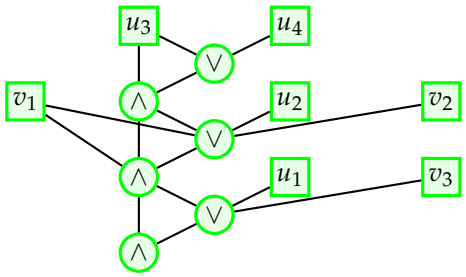


Figure 6.4: The trie of paths from Figure 6.3

```
LOCATION(x)        // memoized
1  children := []      // empty list
2  for each incoming edge e
3        append (label(e), Location(source(e))) to children
4  return hashCons(label(x), children)
```

Figure 6.5: Algorithm for computing locations

A naive implementation of the function w_2 is very slow in the worst case. To compute it quickly, locations must be represented in a compact form.

Example 13. The locations of the leafs of dags p and q from Figure 6.3 can be represented by the dag in Figure 6.4.

Hash-consing is used again. Notice that once the data structure in Figure 6.4 is built, a single reference comparison is enough to conclude that u_1 and v_3 have the same location, and another reference comparison is enough to conclude that u_2 and v_2 have the same location. Figure 6.5 shows the algorithm that computes the data structure for representing locations, in the general case. On line 2 it is assumed that the incoming edges of the original dag are available. This is usually not the case for SMT dags and for Boogie ASTs, which is why a simple preprocessing step is necessary. Hash-consing is invoked on the last line: The call to *hashCons* returns an old data structure, if there is one with the same content. Finally, note that the algorithm in Figure 6.5 assumes that there is a way to extract labels from nodes and edges of the original dag (the method *label*).

6.4.2 Matching Boogie ASTs

If sharing is improved at the level of SMT dags, then the caches of FreeBoogie's transformers are not used. The new Boogie program is read, a new VC is computed repeating all the work, and only then commonalities are observed. Unfortunately, an implementation that improves sharing at the level of Boogie

ASTs is much harder to get right than one at the level of SMT dags, although there does not seem to be any insurmountable conceptual issue.

One problem is that the substitution must be applied on the new Boogie AST, which corresponds to dag q. The proof that any variable substitution in the old VC is sound (see (6.63) on page 96) does not help here. Instead, it must be argued that any one-to-one renaming does not change the semantics of Boogie programs. Another problem is that variables with a limited scope are much more common, so some special treatment for them may be desirable. (In fact, the implementation in FχZ7 does treat quantifiers specially because they limit the scope of variables.) On the bright side, some issues disappear. For example, since there is no sharing within one Boogie AST (Section 3.3), locations within a Boogie AST may be represented in the naive way.

In general though, any solution to Problem 9 should be adaptable to Boogie ASTs. Nodes' labels are the class names; edges' names are the field names. (See Section 3.3.)

6.5 Example

Let us go back to the example in Figure 6.1 on page 86. The method *dayOfYear* is the method of interest. The other methods may or may not have bodies, but that is irrelevant because the verification is modular. Three types of changes are illustrated. First, adding line 1 is a trivial change. Second, adding line 8 is a change in the specification of another method. In general, whenever the contract of a method is changed, all the methods that depend on this contract must be re-verified. In particular, if a contract's postcondition is strengthened, then all dependent methods that were correct remain correct. Third, adding line 15 or line 30 instructs the verifier to check extra properties. Note that adding line 15 also entails re-verification of all the methods that depend on the method *dayOfYear*, which are not of interest in this example.

Proving the VC generated by ESC/Java for the program of Figure 6.1 takes about 17 seconds with Simplify in all cases. FχZ7 spends about 2 seconds to prune the VC in all cases. This could be improved in a careful implementation in FreeBoogie. For the original program plus line 1 or plus line 8, the pruned VC is handled by Simplify in < 0.1 seconds. For the original program plus line 15, the pruned VC is not handled faster by Simplify, so verification is slowed down overall. For the original program plus line 30, the pruned VC is handled in about 8 seconds by Simplify, so proving time is approximatively cut in half. These numbers show that, in some cases, a proper implementation of pruning may be useful. Proving the same method with Z3 2.0 takes 0.5 seconds and with

F*7 takes about 2000 seconds. This suggests that pruning might be worthwhile only for methods that are difficult to verify.

It is also possible to obtain the VC with FreeBoogie, although not completely automatically. The program in Figure 6.1 without annotations can be compiled with javac (0.7 seconds) and then transformed into Boogie with B2BPL (0.5 seconds). Then, annotations must be added manually in the Boogie code. FreeBoogie then generates a VC (1.4 seconds) and then a prover is used (times are as before). Keep in mind that all these times include repeatedly parsing and writing to disk around 10 KB of data. The time taken by FreeBoogie is quite big by comparison with the other stages of the pipeline, which suggests that improving sharing at the Boogie AST level might be worthwhile even for relatively easy problems like this one.

6.6 Conclusions

The previous sections of this chapter present a design for a future component of FreeBoogie. The design is explored theoretically and also practically, through the prototype implementation in F*7. The most appealing property of the design is its *flexibility*.

The two important ideas are

1. finding common parts of two trees under renaming of (some) leafs and
2. pruning a VC based on an old valid VC.

The solution to the first problem is flexible because it depends on a similarity measure for which there are many possible choices. The solution for the second problem is flexible because of the generic proof technique, which can be used to quickly check whether other simplification rules are sound.

The main drawback is the lack of any guarantees that pruning reduces the query size when it is possible. All we know is that *if* it reduces the size, then it does so in a sound way.

6.7 Related Work

The basic idea, that of taking advantage of the results of old runs to speed up new runs, appears in incremental compilation [160], extreme model checking [94], proof reuse [38], and in many other places. The work on proof reuse of Beckert and Klebanov [38] is most similar to the work presented in this chapter. However, they focus on interactive theorem proving, and their similarity heuristic is quite different.

Detecting differences between trees is an area rich in research results. However, it seems that the goal of improving sharing by the renaming of leafs was not addressed before. Usually the goal is to compute the tree edit distance [164] and the accompanying edit script that completely transforms a tree into the other. For unordered trees, such as those with associative–commutative operators, finding the tree edit distance is NP-hard [43], while for ordered trees cubic time was achieved [66]. The cubic time matches that of the heuristic described in Section 6.4, where the cubic time Hungarian algorithm [115] is used. However, profiling shows that computing the matching takes far less time than computing similarity weights between all pairs of leafs.

The idea of automatically pruning an SMT query based on the information in a similar known query seems to be new.

There are many other approaches for improving the speed of program verifiers. James and Chalin [103] run multiple different provers in parallel. They also cache the text of VCs so that they never send the exact same query to a prover twice. Although they speculate that normalizing identifiers would reduce the brittleness of the cache, it seems that the techniques of Section 6.4 fit perfectly their requirements.

Verifast [102] and jStar [71] are Java program verifiers that use symbolic execution as an alternative to VC generation and their good performance seems to owe to this design choice. It is interesting to note that Moore [134] shows that interpreting a program according to some operational semantics leads to exactly the same proof obligations as the subgoals necessary to prove a VC.

Babić and Hu [26] present a set of techniques for improving efficiency, including caching results, combining a rough symbolic execution with VC generation, and using application-specific theorem provers. They also use "maximally shared graphs," which are almost the same as hash-consed expressions. (Their definition is more general because it allows cycles, but it seems that all their graphs are actually dags.) Hash-consing was described in 1958 by Ershov [73], and is a concept that keeps being rediscovered, so it probably deserves to be better known.

The SMT community standardized a language [150] for expressing predicates and is in the process of standardizing a command language for communicating with the prover. The latter may alleviate the need for ornate identifiers.

In the theorem proving community, the string sequences that are part of locations, as defined in Section 6.4.1, are known as *path strings* [156, Chapter 26].

Chapter 7

Semantic Reachability Analysis

"To be successful you have to be selfish, or else you never achieve. And once you get to your highest level, then you have to be unselfish. Stay reachable. Stay in touch. Don't isolate."
— Michael Jordan

Program verifiers typically check for partial correctness or termination, but there are other interesting analyses. This chapter (1) explains when and why reachability analysis is useful, (2) presents its theoretical underpinnings, and then (3) presents and analyzes algorithms that make it practical.

7.1 Motivation

Semantic reachability analysis finds four seemingly unrelated types of problems: dead code, doomed code, inconsistent specifications, and bugs in the frontend of the program verifier.

7.1.1 Dead Code

Figure 7.1 illustrates two problems. Line 10 is dead in the standard sense; line 5 is dead only if annotations are taken into account.

Dead code analysis is standard in compilers. Java, for example, forbids any statement following the statement **return**, because such situations are usually unintended bugs. The novelty here is that we take into account annotations,

```
1  static void m(int x)
2    requires x >= 0;
3  {
4    if (x < 0)
5      throw new IllegalArgumentException("x must be nonnegative");
6    ...
7    if (y < 1) {
8      ...
9      if (y > 1) {
10       ...
11   }}
12 }
```

Figure 7.1: Dead code

which may be non-executable. When the precondition holds (line 2) the first **if** condition (line 4) does not, so no exception is thrown.

In current practice, programmers check arguments at the beginning of the method body with a series of guarded **throw** statements. The pattern is so common that it is encapsulated in libraries [11] so that the code is slightly shorter and slightly easier to read. The small gains add up. In the presence of annotations, however, the code that checks if arguments are legal is mostly redundant.

If all calls to method m are checked statically against its specification, then no runtime check is necessary. A runtime check remains desirable when unverified code may call method m. There are Java compilers [51] that generate bytecode from JML annotations. The translation is not perfect. One technical problem is that JML preconditions do not have descriptions. If the expression is particularly simple (such as $x \geq 0$ in Figure 7.1) then it is possible to construct the description automatically ("x must be nonnegative"). For longer expressions, though, the automatically generated description is unwieldy and unuseful. The problem is just technical since it has the simple solution of modifying JML to have descriptions for preconditions, postconditions, and assertions. The second problem is more serious. If the expression contains quantifiers, which in JML are typically over very large domains like the integers, then it is not always possible to check it efficiently at runtime. However, in such situations it is likely that the programmer would have not written a runtime check.

7.1.2 Doomed Code

Figure 7.2 illustrates two problems, one on line 7 and one on line 14. These where the most common issues in practice (Section 7.4).

Execution always crashes at line 7, before printing. Any test that covers line 7 would detect the problem, but test suites rarely have complete coverage. The example is adapted from Hoenicke et al. [96], whose example is in turn

```
1   abstract class C {
2     int f, g;
3     static void m1(C x) {
4       if (x != null)
5         x.f = 0;
6       else
7         System.out.println(x.f);
8     }
9     static void m2();
10      modifies f, g;
11    static void m3()
12      modifies f;
13    {
14      m2();
15      ...
16    }
17  }
```

Figure 7.2: Doomed code

inspired by an old bug in Eclipse. Such bugs do occur in real software even if they are easy to detect. It would therefore be beneficial to find such problems automatically and without writing any test. (But this is true about any kind of bug.) Another reason given by Hoenicke et al. [96] is compelling: *The lack of annotations does not lead to false positives.* Most frequent complaints from practicians about formal methods tools are that

1. it is too much work to add annotations and
2. there are too many false positives.

To be more precise, the lack of *assumptions* (stemming, for example, from the lack of preconditions and object invariants) does not lead to false positives. A program point is *doomed* if it crashes in every execution. If an assumption is removed (which may mean, for example, that more values are allowed for an argument), then all the old executions are still possible. In other words, removing an assumption never turns a non-doomed program point into a doomed program point.

Line 14 is also doomed, this time because of annotations, not because of the code. The **modifies** clause lists the fields that a method is allowed to assign to. From the point of view of the program verifier the execution never proceeds past line 14, so all potential bugs that follow are hidden.

7.1.3 Inconsistent Specifications

Figure 7.3 illustrates two problems caused entirely by specifications.

The contract of method *m1* cannot possibly correspond to a correct implementation, yet there is no implementation to check against the contract. In this case the implementation is written in a different language, but sometimes it

```
1  pure static native int m1()
2    ensures result == result + 1;
3  static int m2(int x)
4    requires x < 0;
5    requires x > 0;
6  {
7    return x / 0;
8  }
```

Figure 7.3: Inconsistent specifications

```
1  int x;
2  for (int i = 5; --i >= 0;)
3    x = i;
4  x = 1 / x;
```

Figure 7.4: Unsoundness of the frontend

is simply unavailable, for example if it comes from a proprietary library. The example may seem silly since no one would make the mistake to think that some integer equals its successor. However, inconsistencies may arise from other causes. One example is specification inheritance: Since the conflicting annotations are in different files it is hard for a human to notice. Another example is the use of pure methods in specifications. The literature concerned with tackling this latter source of inconsistencies is reviewed in Section 7.5.

Lines 4 and 5 illustrate another type of bug caused by inconsistent specifications. The body of method *m2* is available and the program verifier will always report it is correct. Without reachability analysis, the bug is caught only if method *m2* is called. Such behavior is somewhat akin testing, which catches bugs (only) by calling the (potentially) buggy methods.

In practice, inconsistent specifications are common.

7.1.4 Unsoundness and Bugs

The code in Figure 7.4, apart from being more than a little silly, will always crash at line 4. That is not the issue relevant here, however. The issue is that the problem may be missed by an unsound program verifier.

Again, the example might seem contrived. Why not get rid of unsoundness in the first place instead? And why would we expect this problem to appear often? Before answering these questions let us see why this is indeed a problem in ESC/Java (with default options). ESC/Java was designed primarily to be useful to programmers and it tackled the problem of false positives by choosing to be unsound. One example of unsoundness is that, by default, loops are unrolled three times. This means that all executions for which the body of the loop is executed more than three times are considered to miraculously terminate

with success. All executions in Figure 7.4 go through the loop body five times so, just before executing the loop body for the fourth time, ESC/Java will consider that the execution finishes with success. This makes ESC/Java miss the problem on line 4. Note that if the loop would be executed n times instead of 5 times, where n is a variable possibly less than four, then the problem on line 4 would be caught.

We are now in a better position to answer the two questions.

First, a tool may choose to be unsound for engineering reasons. This is not as bad as it may sound. We saw that a loop with a variable bound instead of a constant bound does not cause problems, and variable bounds are probably more common. Also, this is only the default behavior. A novice user is supposed to get rid of the errors signaled in this unsound default mode of execution. An expert user is supposed to ask ESC/Java to treat loops in a safe way. In this case, the tool usually requires more guidance in the form of annotations.

Second, the unsoundness might not be intended, but rather a bug. In this guise, reachability analysis is useful as a safeguard against buggy implementations of the program verifier itself. It is common for good programmers to write routines that check for complex invariants at runtime and use these routines during development. Reachability analysis has been used in this way in the Boogie tool from Microsoft Research under the name of "smoke testing".

7.1.5 An Unexpected Benefit

Finally, in conjunction with loop invariant inference techniques, reachability analysis detects certain non-terminating programs. Consider the following code fragment.

```
1  for (int i = 0; i < 10; ++j)
2    sum += i;
3  ...
```

It is easy to infer the loop invariant $i = 0$, which contradicts the negation $i \geq 10$ of the loop condition. The code after the loop is unreachable because the loop never terminates.

7.2 Theory

Semantic reachability analysis is performed after passivation (Chapter 4), so the program it analyzes is an acyclic flowgraph without assignments.

Definition 27. A node x in a flowgraph is *semantically reachable* when the flowgraph has an execution containing the state $\langle \sigma, x \rangle$ for some store σ.

For a correct flowgraph (see Theorem 3 on page 70), a node x is semantically reachable when the flowgraph becomes incorrect after replacing the statement of node x with **assert false**. (This is because according to (2.7) on page 14 there is a transition $\langle \sigma, x : \textbf{assert false} \rangle \rightsquigarrow \mathit{error}$ for all stores σ.) A naive algorithm for semantic reachability analysis replaces each statement in turn by **assert false** and reverifies the program. However, this algorithm handles only correct flowgraphs and is slow.

There is a very easy way to make any flowgraph correct: Transform all assertions into assumptions! Besides fixing all bugs, this transformation has the useful property that it maintains semantic reachability.

Proposition 10. *Consider a flowgraph G with no assignments and the flowgraph H obtained by replacing each statement **assert** q with the statement **assume** q. Node y is semantically reachable in flowgraph G if and only if it is semantically reachable in flowgraph H.*

Proof. According the operational semantics of core Boogie (see Chapter 2, especially (2.6)), a flowgraph has an execution

$$\langle \sigma, x_1 : \textbf{assert/assume } p_1 \rangle, \ldots, \langle \sigma, x_n : \textbf{assert/assume } p_n \rangle, \langle \sigma, y \rangle$$

when there is a path $x_1 \to \cdots \to x_n \to y$ in the flowgraph and there is a store σ that satisfies $p_1 \wedge \ldots \wedge p_n$. It is irrelevant whether intermediate statements are assertions or assumptions as there is only one rule in the operational semantics for both. □

There is still the issue of efficiency. Chapter 5 showed that both vc_{wp} and vc_{sp} can be computed in time linear in the size of the flowgraph. Since we need to reverify the program once for each node in the flowgraph, the total time to compute the prover queries is quadratic in the size of the program. Most flowgraphs in practice have at most hundreds of nodes and hundreds of edges, which means that a quadratic algorithm will be very fast and the real problem is the time spent in the prover.

The next section is concerned with reducing the number of calls to the prover. Before that, let us briefly see how to compute *all* prover queries at once in linear time.

If there is an execution that contains the state $\langle \sigma, y \rangle$, then the previous states $\langle \sigma, x \rangle$ correspond to nodes x that have a path $x \rightsquigarrow y$ to node y. Hence, to determine whether such executions exist we can look at the sub-flowgraph induced by nodes x that can reach node y. This observation speeds the computation of VCs by a factor of two, for both methods—weakest precondition and strongest postcondition.

Observe now that the precondition a_y computed by the strongest postcondition method (Section 5.2.3) depends only on nodes y that can reach node x. In other words, the precondition a_y does not change when we select a subflowgraph corresponding to some other node x', so it needs not be recomputed. We simply compute all preconditions a_x in linear time according to the equations for the strongest postcondition method in Section 5.2.3. When we replace statement x by **assert false** it becomes the only assertion in the flowgraph and the VC according to (5.22) is $\neg a_x$. We proved the following.

Proposition 11. *A node x is semantically reachable if and only if its precondition a_x computed using the strongest postcondition method (Chapter 5) is satisfiable, that is, when $|\neg a_x|$ holds.*

According to (5.21) (on page 73) there is no difference between **assume** and **assert** when computing preconditions a_x and postconditions b_x using the strongest postcondition method. So one advantage of the strongest postcondition method is that we do not have to transform assertions into assumptions at all. Another clear advantage of the strongest postcondition method is that it produces all prover queries in linear time, as opposed to the weakest precondition method which requires quadratic time. It is intuitive that 'going forward' is better suited for analyzing semantic reachability.

Types of Problems Nodes that are semantically unreachable are likely to be caused by bugs. For example, the contradictory preconditions in Figure 7.3 (on page 107) cause the body of method *m2* to be semantically unreachable. It is therefore interesting to find potential causes of unreachable code.

A node x is a *blocker* when it is reachable but lets no execution pass through it. That is, there is no execution containing $\langle \_, x \rangle, \langle \_, \_ \rangle$. Yet in other words, the only possible successor of state $\langle \_, x \rangle$ (in an execution) is the *error* state. If the only predecessor of node y is a blocker, then node x is semantically unreachable, according to (5.20) (on page 73).

There are three types of problems detected by the semantic reachability analysis:

1. Semantically unreachable nodes. This includes all dead code.
2. Blocker assumptions. These are likely root causes for semantically unreachable nodes, so they point closer to the actual bug.
3. Blocker assertions, also known as doomed program points. These are bugs that manifest in every execution.

```
NAIVEREACHABILITYANALYSIS(G)
1  for each statement x of G
2      if Valid(Not(Pre(x)))
3          report that node x is semantically unreachable
4      if statement x is an assertion and Valid(Not(Post(x)))
5          report that node x is doomed
```

Figure 7.5: A simple algorithm for semantic reachability analysis

Formally, there can be no error after a blocker. That is, there is no execution that goes wrong after it passed through a blocker, for the simple reason that there is no execution passing through a blocker. However, intuitively it is helpful to think of blockers as potentially hiding subsequent bugs.

7.3 Algorithm

Figure 7.5 summarizes the algorithm suggested by the previous section. The methods *Pre* and *Post* are those from Figure 5.2 (on page 76); the method $Not(p)$ constructs the SMT dag that represents the predicate $\neg p$, perhaps carrying out basic simplifications; the method *Valid* checks whether its argument represents a valid predicate by querying the prover. The $|V|$ calls to the method *Pre* take $O(|V| + |E|)$ time and similarly for method *Post* (see the proof of Proposition 7 on page 75).

This algorithm reports too many errors. For the Boogie program

```
1  assume false;
2  assert p1;
3  ...
4  assert pn;
```

it prints n error messages, one for each assertion. Surely, the user would prefer one error message that points at the assumption, which is the cause of all the other errors.

More seriously, it turns out that this algorithm, even if it takes just linear time to build the prover queries, is unusable in practice because it is too slow. It is simply not acceptable to call the prover $|V|$ times for one method. A typical method has $|V| \approx 100$ nodes, and one call to the prover typically takes between a tenth of a second and one second.

Remark 10. All the times are given for an implementation inside ESC/Java, which is activated by the option `-era`. ESC/Java has a robust frontend for Java with JML annotations, so it is easier to perform meaningful experiments. The implementation first constructs a flowgraph essentially equivalent to those built by FreeBoogie, so the implementation should be easy to adapt.

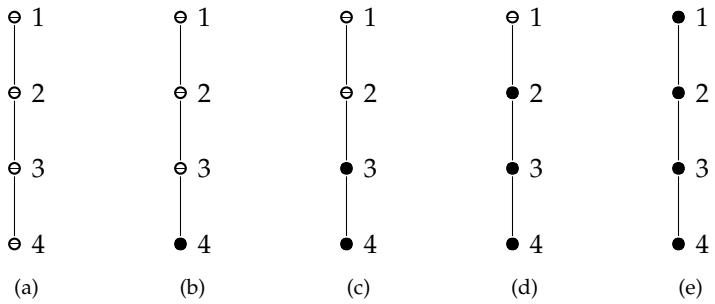


Figure 7.6: A path flowgraph with 4 nodes

7.3.1 The Propagation Rules

Figure 7.6(a) shows a path flowgraph. (Node 1 is the initial node.) Node 4 is semantically reachable when there is an execution that contains the state $\langle \sigma, 4 \rangle$, for some store σ. Such an execution must correspond to some path from the initial node 1 to node 4, and the only such path is $1 \to 2 \to 3 \to 4$. Hence, the execution is $\langle \sigma, 1 \rangle$, $\langle \sigma, 2 \rangle$, $\langle \sigma, 3 \rangle$, $\langle \sigma, 4 \rangle$, which means that the other nodes are semantically reachable too. (The store is not modified by assertions or assumptions.) This motivates the study of the following problem, which is more abstract.

Problem 10. *Alice and Bob play a game. They start by sharing the flowgraph G. Then Alice secretly colors the nodes with white and black such that each white node is connected by a white path with the initial node. Bob's goal is to reconstruct the coloring by asking Alice as few questions as possible. He is allowed to ask "what is the color of node x?" for any node x and Alice will answer truthfully.*

Remark 11. Alice is the theorem prover, Bob is FreeBoogie (or ESC/Java), white nodes are semantically reachable nodes, and black nodes are semantically unreachable nodes.

Bob's flowgraph initially has only gray nodes and, by the end, he colors each with either white or black. At some intermediary stage, he has some white nodes, some black nodes, and the rest are gray. We say that two colorings are *consistent* when there is no node that is white in one coloring and black in the other. Bob maintains his coloring consistent with Alice's coloring. (Also, because it is not useful, he never changes the color of a node into gray.) A *valid coloring* has no gray node and satisfies the condition that

every white node is reachable from the initial node by a white path. (7.1)

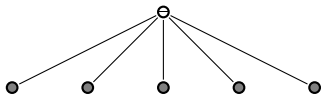


Figure 7.7: Worst case flowgraph

For example, Figure 7.6 shows all the valid colorings of a path flowgraph. A *possible coloring* (at some intermediate stage) is a valid coloring that is consistent with Bob's coloring. The rule which Bob uses to paint nodes white or black is simple: If node x is white in all possible colorings, then Bob colors it white; if node x is black in all possible colorings, then Bob colors it black. Obviously, this maintains the invariant that Bob's coloring is consistent with Alice's coloring. To ensure progress, Bob asks Alice from time to time about the color of a node that is still gray in his coloring.

Example 14. Consider the flowgraph in Figure 7.7. The initial node is always reachable. Assuming that Bob starts with the initial node white and the others gray, he must ask 5 questions. After asking k questions he must consider 2^{5-k} possible colorings. However, the preconditions of all non-initial nodes are the same as the postcondition of the initial node. Therefore, they are either all semantically reachable or all semantically unreachable, and this can be determined with one call to the prover.

Example 14 seems to suggest that performance is hurt because Problem 10 is a *too* abstract model. Related to this, it seems that the theorem prover is never queried about doomed code (unsatisfiable postconditions), but only about semantically unreachable code (unsatisfiable preconditions). Both these issues have an easy fix. Instead of playing the color game on the original flowgraph, Alice and Bob play it on a modified flowgraph that has two nodes x_i and x_o for each node x in the original flowgraph. Node x_i takes over the incoming edges of node x; node x_o takes over the outgoing edges of node x. Also, there is an edge $x_i \to x_o$. When Bob asks Alice about the color of node x_i, this means that ESC/Java (or FreeBoogie) asks the theorem prover about the satisfiability of the precondition a_x; when Bob asks Alice about the color of node x_o, this means that ESC/Java (or FreeBoogie) asks the theorem prover about the satisfiability of the postcondition b_x.

Bob does not want to look at each possible coloring, because there might be an exponential number of them. Hence, he must characterize the nodes whose color he can infer without referring to all possible colorings.

Definition 28. Node x of a flowgraph *dominates* node y when all paths from the initial node to node y contain node x. Node x *immediately dominates* node y if $x \neq y$ and all nodes that dominate node y also dominate node x.

Proposition 12. *A gray node is white in all possible colorings if and only if it dominates a white node in the non-black subgraph (in Bob's coloring).*

Proof. Because Bob's coloring is consistent with a valid coloring it is possible, for each white node y to find a non-black path from the initial node to node y. Consider the coloring constructed by picking such a path for each white node and then painting all these paths white; then paint the remaining gray nodes black. This coloring is possible by construction. Also, whenever white node y is processed, choose a path that avoids node x, if possible. This shows that if node x does not dominate a white node, then there are some possible colorings in which it is black.

For the converse, consider a coloring in which node x is black and dominates white node y. Then (7.1) is violated and the coloring is invalid. □

Similarly, one can prove the following.

Proposition 13. *A gray node x is black in all possible colorings if and only if all the paths from the initial node to node x contain a black node (in Bob's coloring).*

An immediate consequence of the last two proposition will prove useful.

Corollary 2. *A gray node is sometimes white and sometimes black in possible colorings if and only if in the non-black subgraph (a) it does not dominate a white node and (b) it is connected to the initial node. (Again, in Bob's coloring.)*

Figure 7.8 shows how Bob paints his flowgraph. The method *ComputeDominators* computes the dominator tree of a graph (Section 7.3.2). The method *PickQueryNode* returns some gray node (Section 7.3.3). The main loop of the method *RecoverColoring* maintains the invariants:

1. The coloring is consistent with Alice's coloring.
2. For each gray node, there are possible colorings in which it is white and possible colorings in which it is black.

It is easy to see that both invariants hold when all the nodes are gray. Let us see why they are maintained, first in the case when Alice says that node x is white and then in the case when Alice says that node x is black.

If Alice says that node x is white then there are no possible colorings in which it is black, so it must be painted white to maintain invariant 2. This is done on line 1 of the method *PropagateWhite* when it is first called from line 6 of the method *RecoverColoring*. However, painting node x in white might break

```
PROPAGATEWHITE(y, T)
1  paint node y in white
2  PropagateWhite(parent of y in tree T)

PROPAGATEBLACK(x, G)
1  paint node x in black
2  for each gray successor y of node x in G
3       if all predecessors of node y are black
4            PropagateBlack(y, G)

RECOVERCOLORING(G)
1  paint in gray all the nodes of the flowgraph G
2  while there are gray nodes
3       T := ComputeDominators(non-black subgraph of G)
4       x := PickQueryNode(G, T)
5       if AskAlice(x) = white
6            PropagateWhite(x, T)
7       else
8            PropagateBlack(x, G)
```

Figure 7.8: Solution outline for Problem 10

invariant 2 with respect to another gray node, because gray nodes must not dominate white nodes (Corollary 2). This problem is fixed by painting all the dominators of node x in white, which is allowed by Proposition 12. It follows from the definition of an immediate dominator (Definition 28) that method *PropagateWhite* visits exactly the dominators of node x.

If Alice says that node x is black then, for similar reasons as in the previous case, it must be painted black. This may break the other part of invariant 2, namely, it may make disconnect gray nodes from the initial node in the non-black subgraph. Should this happen, Proposition 13 says that all the disconnected gray nodes must be painted black. This is the purpose of the method *PropagateBlack*. Let us say that a node is immediately disconnected when all its parents are black. Initially no node is immediately disconnected (invariant 2) but one may become so when its parent is painted black (on line 1 of the method *PropagateBlack*). Whenever this happens it will be visited. Therefore, the method *PropagateBlack* visits and paints in black all immediately blocked nodes. Now it is easy to see that if all immediately blocked nodes are black, then there is no gray node that is disconnected from the initial node in the non-black subgraph. Pick some gray node. Because it is not black, it is not immediately blocked, so it has a non-black predecessor. Then repeat.

We have proved that the algorithm is correct. It is also fast.

Proposition 14. *If calls ComputeDominators(), PickQueryNode(), and AskAlice()*

each takes constant time, then the method RecoverColoring takes $O(|V| + |E|)$ time.

Proof. The methods *PropagateWhite* and *PropagateBlack* are called only for gray nodes and the first action paint the given node in black or white. Therefore, they are called at most once per node.

The method *PropagateBlack* examines the outgoing edges of node x, so it might examine all edges by the end. The condition on line 3 of the method *PropagateBlack* can be evaluated in constant time if each node keeps a count of its non-black parents. □

7.3.2 Dominators Tree for Dags

This section is a brief reminder of basic algorithms related to dominators. It explains how the method *ComputeDominators* in Figure 7.8 is implemented.

The dominators tree T of a flowgraph G is a tree in which the parent of node x, denoted $idom(x)$, is the immediate dominator of node x in the flowgraph G. The root of the dominators tree is the initial node of the flowgraph, which is the only node without an immediate dominator. All other nodes y obey the equation

$$idom(y) = LCA(\{\, x \mid x \to y \,\}), \tag{7.2}$$

where $LCA(S)$ is the root of the smallest dominators subtree that contains the set S of nodes.

The method *ComputeDominators* examines nodes in a topological order of the flowgraph and inserts them as leafs in the proper place in the dominators tree that it builds. When node y is processed all its predecessors x have taken their place in the dominators tree so *LCA* can be computed.

7.3.3 Choosing the Query

A greedy algorithm always asks the query whose answer provides most information. If the probability of receiving the answer *black* is p, then the information provided by the answer is $p \lg p + (1 - p) \lg(1 - p)$, which has a maximum at $p = 1/2$ and is symmetric around that point. In other words, the greedy information theoretic approach says that we should always inquire about the node whose probability of being black is as close as possible to $1/2$.

An Example To estimate the probability of various nodes being black we need a model. Let us go back to the example flowgraph in Figure 7.6 (on page 112). The simplest possible model is the one in which each node has an independent probability p of having a *bug* and probability $q = 1 - p$ of not having a bug.

A node with a bug is black and so are all that follow it. In this model, the five possible colorings of Figure 7.6 have probabilities (a) q^4, (b) q^3p, (c) q^2p, (d) qp, and (e) p. The probability that node x is black is the sum of probabilities of possible colorings in which node x is black. For the nodes in Figure 7.6 we have (1) $1-q$, (2) $1-q^2$, (3) $1-q^3$, and (4) $1-q^4$.

In general, for a path flowgraph with nodes $1 \to 2 \to \cdots \to n$, the probability that node k is black is

$$p_k = 1 - (1-p)^k \tag{7.3}$$

where p is the probability that a node has a bug. In the common case, we expect the analysis to be run on code that is mostly correct. Indeed, bugs are usually clustered in the region of the code that is actively developed, while the program verifier is run on the whole code base. Hence, in a first approximation, it is reasonable to expect p to be very small. Then $p_k \approx kp$ and, if p is really small, the probability that is closest to $1/2$ is p_n. In other words, if we are fairly confident that the code is correct, then we should query the last node in the path. If the code is indeed correct, then we are done after one query.

However, if the first query returns *black* then there is at least one bug. The expected number of bugs for the model we use is $b = pn$. (The number of bugs has the probability generating function

$$B(z) = \sum_k \binom{n}{k} (1-p)^{n-k} p^k z^k = (1-p+pz)^n, \tag{7.4}$$

so the average is $B'(1) = pn$.) Therefore, if we expect the program to have b bugs, then it makes sense to choose

$$p = b/n \tag{7.5}$$

in our model. Solving $p_k = 1/2$ we obtain

$$k = -1/\lg(1-p) \approx (\ln 2)/p. \tag{7.6}$$

Plugging in the estimate (7.5) we finally arrive at

$$k \approx 0.7 \cdot n/b \tag{7.7}$$

In other words, the information theoretic greedy strategy is a binary search that splits the interval $i..j$ not at $i + 0.5(j-i)$, but at $i + (0.7/b)(j-i)$, where b is the expected number of bugs in the interval $i..j$.

```
REACHABILITYANALYSIS(G)
 1  construct H by splitting nodes of G
 2  paint the initial node of H white and the others gray
 3  T := ComputeDominators(H)
 4  while H has gray nodes
 5      x := a leaf from the deapest level of T
 6      if Valid(Not(Formula(x)))
 7          PropagateBlack(x, H)
 8          let [y_1, ··· , y_n] be the path in T from y_1 = root(T) to y_n = x
 9          i := 1,   j := n
10          while i + 1 < j        // y_i is white, y_j is black
11              k := round(i + 0.7(j − i))
12              if Valid(Not(Formula(y_k)))
13                  PropagateBlack(y_k, H)
14                  j := k
15              else
16                  PropagateWhite(y_k, T)
17                  i := k
18          T := ComputeDominators(non-black subgraph of H)
19      else
20          PropagateWhite(x, T)
```

Figure 7.9: Algorithm for semantic reachability analysis

The General Case Most observations for path flowgraphs generalize easily. We will have two modes of work, one in which we expect no bug and one in which we expect one bug. When we expect no bug we will pick a gray node that is far from the initial node. When we expect one bug we will perform a binary search on a path of nodes in the dominator tree. The key here is that color propagation rules on paths in the dominator tree are exactly the same as the color propagation rules for a path flowgraph.

Figure 7.9 summarizes all the observations made so far. Line 1 splits each node x of the flowgraph into a node x_i carrying the precondition a_x and a node x_o carrying the postcondition b_x. Later these predicates are accessed using the method *Formula*(). Line 2 colors the initial node white to establish the invariant of the loop on line 10.

7.3.4 Performance

If the verified flowgraph has no bugs (no semantically unreachable statements and no doomed code) then l queries are necessary and sufficient, where l is the number of leafs in the flowgraph. The algorithm reaches this bound. In other words, if there are no bugs, then the prover is queried once for each **return** statement in the program. For each bug, there are $\sim \lg h$ extra queries, where

h is the (average) length of a path in the flowgraph from the initial node to a **return** node. In total, there are roughly $l + b \lg h$ queries for a program with b bugs.

Experiments The frontend of ESC/Java (JavaFE) contains 1890 methods and is one of the largest coherent JML-annotated code base. Verifying it takes 31589 seconds (almost 9 hours), out of which 34.8% is spent in semantic reachability analysis, out of which 99.8% is spent in the prover. The total number of leafs in flowgraphs is 3256 and the total number of prover queries is 3351. In other words, on average per method

- semantic reachability analysis takes 5.82 seconds,
- out of which 5.81 seconds are spent in 1.77 prover queries (not much more than the number of **return** statements, which is 1.72)
- and the other 0.01 seconds are spent computing prover queries, deciding which queries to perform, and propagating semantic reachability information in the flowgraph.

(This benchmark was run using the default treatment of loops in ESC/Java—unrolling once.)

7.4 Case Study

The frontend of ESC/Java (known as JavaFE) consists of 217 annotated classes with 1890 methods. Semantic reachability analysis revealed ≈ 50 issues. Short descriptions of these problems follow, each description preceded by the number of cases to which it applies.

7.4.1 Dead Code

1 A **catch** block was unreachable because it was catching a *RuntimeException* and the specifications of methods called in the **try** block did not signal that exception type.

9 Informal comments indicated that the dead code is intended. JML has an equivalent annotation, which ESC/Java understands and should be used instead of the informal comments.

1 The code was semantically unreachable in the classic sense (no annotation involved.)

7.4.2 Doomed Code

6 A few assertions were bound to fail on any input.

9 The scenario illustrated by methods *m2* and *m3* in Figure 7.2 on page 106. The method call was bound to fail.

7.4.3 Inconsistent Specifications

5 Inconsistencies in the JDK specifications that ship with ESC/Java. They were filed as ESC/Java bugs #595, #550, #568, #549, and #545. These are bugs whose cause was hard to track. Before, they have escaped for years the ESC/Java's test-suite that was designed to catch them.

7.4.4 Unsoundness and Bugs

1 The JDK specification used a JML *informal comment*. According to the semantics of JML, an informal comment should not cause warning messages. Roughly, this means that, depending on the context, an informal comment should be treated sometimes as **true** and sometimes as **false**. However, ESC/Java always treats is as **true**. See ESC/Java bug #547 for details.

4 Loops with a constant bound that is bigger than the loop unrolling limit.

2 If the **modifies** clause is missing on a method *m*, ESC/Java considers that method *m* is allowed to modify any global variable while checking it, but assumes that method *m* does not modify any global state while checking methods that call it. This is obviously unsound (and motivated by the desire to not flood new users with irrelevant warnings).

7.4.5 Others

12 The cause of these warnings provided by the semantic reachability analysis is unclear. Given the experience of tracking down the cause of some of the other warnings, it is likely that a complex interaction between annotations located in different files is involved.

7.5 Conclusions and Related Work

This chapter presents the theoretical underpinnings of semantic reachability analysis for annotated code and an efficient algorithm for finding

1. dead code,
2. doomed code,
3. inconsistent specifications, and
4. unsoundness bugs.

Abstract interpretation [62] and symbolic execution [109] are program verification techniques that can easily detect semantically unreachable code as a by-product, although they seldom seem to be used for this purpose. To ensure termination, these techniques usually over-approximate the set of possible states, which means that they might conclude that some code is semantically reachable, although it is not.

Lermer et al. [125] give semantics for execution paths using predicate transformers. In particular, they define *dead paths* and *dead commands*. A dead path is one that is taken by no execution; a dead command is a program whose VC computed by the weakest precondition method is unsatisfiable.

Hoenicke et al. [96] explain why it is desirable to detect doomed code. Their implementation uses the weakest precondition method to compute prover queries associated with each node. The irrelevant portions of the flowgraph are turned off using auxiliary variables rather than explicitly manipulating the flowgraph (see Section 7.2).

Chalin [52] explains why it is desirable to have automatic sanity checks for specifications and designs an automated analysis that finds bugs in JML annotations, other than plain logic inconsistencies.

Cok and Kiniry [58, 59] explain how ESC/Java handles method calls that appear in annotations. Darvas and Müller [64] point out that that treatment is unsound because some annotations are translated by ESC/Java's frontend into inconsistent assumptions. Their solution is a set of syntactic constraints on the use of method calls in annotations. Rudich et al. [158] relax these constraints while maintaining soundness. Leino and Middelkoop [121] replace the syntactical constraints with queries to the theorem prover. Semantic reachability analysis will signal most problems caused by the use of method calls in specification, because they manifest as inconsistencies at the level of the intermediate representation (Boogie). In particular, it is similar to the approach of Leino and Middelkoop [121] in its use of a theorem problem. However, being a generic analysis, the error message is usually not very informative. Another shortcoming of the semantic reachability analysis compared to the other specialized solutions is that it does not ensure that specifications are well-founded.

The naive version of semantic reachability analysis is used in the static verifier Boogie from Microsoft Research under the name "smoke testing" and is

used mostly for catching bugs in the verifier itself.

ESC/Java [76] is a program verifier for Java annotated with JML [117, 59]. By design, ESC/Java favors user friendliness over soundness.

The method *ComputeDominators* uses the algorithm of Cooper [61], which is easy to implement and works fast in practice [83]. (There are linear time algorithms, such as the one of Buchsbaum et al. [50].) Probability generating functions, such as the one used in Section 7.3.3, are treated by most probability textbooks, such as Graham et al. [86, Chapter 8].

Chapter 8

Conclusions

"I love to travel but I hate to arrive."
— Albert Einstein

The design of FreeBoogie (Chapter 3) does not surprise much those who write *program verification* tools, although it has a few distinctive characteristics. FreeBoogie spends most of its time transforming step-by-step a Boogie program into simpler and simpler Boogie programs. The steps are always as small as possible. For example, even the statements **havoc** and **call** could be desugared (into **assume** and **assert** statements) in one step, FreeBoogie does it in two distinct steps. After each transformation, FreeBoogie type-checks the intermediate program to rule out egregious bugs. *Correctness has a very high priority in deciding the design.* The main data structures of FreeBoogie are immutable. A nice side-effect of immutability shows that correctness and efficiency are not always competing goals. Because pieces of Boogie programs are represented by immutable data structures, they can act as keys in caches, which makes it easy to avoid re-doing the same work (Chapter 6).

Operational semantics, Hoare logic, and predicate transformers are ways to define *programming languages*. This dissertation uses all three for a subset of Boogie and lingers on the relations between the three (Chapter 5). Being able to quickly switch between different points of view on the semantics of Boogie programs is essential in understanding the techniques presented later (Chapters 6 and 7).

The *algorithmic* problems solved in this dissertation (Chapters 4 and 7) are not particularly difficult. The most difficult part is the proof of Theorem 2 in Section 4.4.1. Program verifiers try to solve instances of an undecidable problem, which makes for a good excuse to not analyze their algorithms. But, there is not much undecidable going on in many parts of a program verifier. In

particular, a VC generator is a place full of little algorithms that deserve to be better understood.

The goal of this dissertation is to give an account of some interesting but often overlooked parts of a VC generator. The meta-goal of this dissertation is to find connections between program verification*,* programming languages*, and* algorithms*.*

It also happens that the dissertation could serve as a suitable guide for a developer who wants to contribute to the VC generator FreeBoogie. There are 32 000 lines of code that handle the tasks described in this dissertation. (About 23 000 of them are generated by ANTLR, CLOPS, and AstGen.) This infrastructure could serve for future developments such as invariant inference (perhaps via abstract interpretation), termination detection (which would probably entail enriching the Boogie language), and other experiments with the Boogie language (such as adding operators from separation logic).

Appendix A

Notation

"The best notation is no notation; whenever it is possible to avoid the use of a complicated alphabetic apparatus, avoid it. A good attitude to the preparation of written mathematical exposition is to pretend that it is spoken. Pretend that you are explaining the subject to a friend on a long walk in the woods, with no paper available; fall back to symbolism only when it is really necessary."

— Paul Halmos [162]

notation	description
$A, B, \ldots$	sets; objects with an internal structure
$\mathbb{Z}$	the set of integers $-1, 0, 1, \ldots$
$\mathbb{Z}_+$	the set of nonnegative integers $0, 1, \ldots$
$\mathbb{R}$	the set of real numbers
$\mathbb{R}_+$	the set of nonnegative real numbers
$a \in A$ or $a : A$	a is an element of the set A
$A \times B$	the cartesian product of the sets A and B
$A \to B$ or B^A	the set of functions from A to B; $\to$ is right associative
$f\ x$ or $f(x)$	function application; space is left associative and binds tighter than all else
$w(P)$	$\{\, w(x) \mid x \in P \,\}$
$f \circ g$	function composition: $(f \circ g)\ x = f\ (g\ x)$ for all x in the domain of g
$\top$	true

$\bot$	false
$\mathbb{B}$	the set of booleans $\{\top, \bot\}$
e, f	expressions, possibly predicates
p, q, r	predicates
a	preconditions
b	postconditions
u, v, w	variables
x, y, z	statements, nodes in a graph
m, n	integer constants
i, j, k	integer variables, indexes
wp	the weakest precondition predicate transformer
sp	the strongest postcondition predicate transformer
$x \rightarrow y$	directed edge from node x to node y
$x \overset{P}{\rightsquigarrow} y$	path P from node x to node y; the path P is sometimes seen as a set of nodes that includes the endpoints
$\ominus$	graph node
$\bullet$	graph node of interest; read-write flowgraph node
$\ominus$	read-only flowgraph node
$\ominus$	write-only flowgraph node
$x \ominus\cdots\ominus y$	read node with incoming edges from the nodes x and y
G, H	graphs, flowgraphs, programs
V	the set of nodes of a graph
E	the set of edges of a graph
P, Q	paths, sets of nodes
$[p]$	1 if p evaluates to $\top$, 0 otherwise
$\lvert p \rvert$	$\top$ if the predicate p is valid, $\bot$ otherwise

Bibliography

[1] ACL2 homepage.
http://www.cs.utexas.edu/users/moore/acl2/.

[2] Bison homepage.
http://www.gnu.org/software/bison/.

[3] BLAST homepage.
http://mtc.epfl.ch/software-tools/blast/.

[4] Bogor homepage.
http://bogor.projects.cis.ksu.edu/.

[5] CHESS homepage.
http://research.microsoft.com/en-us/projects/chess/.

[6] Coq homepage.
http://coq.inria.fr/.

[7] Crystal homepage.
http://code.google.com/p/crystalsaf/.

[8] Frama-C homepage.
http://frama-c.cea.fr/.

[9] Fx7 homepage.
http://nemerle.org/~malekith/smt/en.html.

[10] FxCop blog.
http://blogs.msdn.com/fxcop/.

[11] Google collections library.
http://code.google.com/p/google-collections/.

[12] HOL homepage.
http://hol.sourceforge.net/.

[13] Isabelle homepage.
http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[14] Java collections API design FAQ.
http://java.sun.com/j2se/1.5.0/docs/guide/collections/designfaq.html.

[15] Java Pathfinder homepage.
http://babelfish.arc.nasa.gov/trac/jpf.

[16] NQuery homepage.
http://www.codeplex.com/NQuery.

[17] NuSMV homepage.
http://nusmv.irst.itc.it/.

[18] PMD homepage.
http://pmd.sourceforge.net/.

[19] PVS homepage.
http://pvs.csl.sri.com/.

[20] RuleBase homepage.
http://www.haifa.ibm.com/projects/verification/RB_Homepage/.

[21] SatEEn homepage.
http://vlsi.colorado.edu/~hhkim/sateen/.

[22] SPIN homepage.
http://spinroot.com/spin/whatispin.html.

[23] Yices homepage.
http://yices.csl.sri.com/index.shtml.

[24] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison–Wesley, 2007.

[25] Lloyd Allison. *A Practical Introduction to Denotational Semantics*. Cambridge University Press, 1986.

[26] Domagoj Babic and Alan J. Hu. Calysto: scalable and precise extended static checking. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE*, pages 211–220. ACM, 2008.

[27] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors,

IFM, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004.
http://research.microsoft.com/en-us/projects/slam/.

[28] Anindya Banerjee, Michael Barnett, and David A. Naumann. Boogie meets regions: A verification experience report. In Natarajan Shankar and Jim Woodcock, editors, *VSTTE*, volume 5295 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2008.

[29] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005.

[30] Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In Michael D. Ernst and Thomas P. Jensen, editors, *PASTE*, pages 82–87. ACM, 2005.

[31] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec$^\sharp$ programming system: An overview. *Lecture Notes in Computer Science*, 3362:49–69, 2005.

[32] Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The satisfiability modulo theories library (SMT-LIB).
http://www.smt-lib.org/, 2009.

[33] Clark Barrett and Cesare Tinelli. CVC3. In Damm and Hermanns [63], pages 298–302.
http://www.cs.nyu.edu/acsys/cvc3/.

[34] Clark W. Barrett, Leonardo Mendonça de Moura, and Aaron Stump. SMT-COMP: Satisfiability modulo theories competition. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 20–23. Springer, 2005.
http://www.smtcomp.org/.

[35] Gilles Barthe, Lilian Burdy, Julien Charles, Benjamin Grégoire, Marieke Huisman, Jean-Louis Lanet, Mariela Pavlova, and Antoine Requet. JACK–a tool for validation of security and behaviour of Java applications. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4709 of *Lecture Notes in Computer Science*, pages 152–174. Springer, 2006.
http://www-sop.inria.fr/everest/soft/Jack/jack.html.

[36] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C specification language.
http://www.frama-c.cea.fr/download/acsl_1.4.pdf.

[37] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of object-oriented software: the KeY approach*, volume 4334 of *Lecture Notes in Artificial Intelligence*. Springer, 2007.
http://www.key-project.org/.

[38] Bernhard Beckert and Vladimir Klebanov. Proof reuse for deductive program verification. In *SEFM*, pages 77–86. IEEE Computer Society, 2004.

[39] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Avner Landver. RuleBase: An industry-oriented formal verification tool. In *DAC*, pages 655–660, 1996.

[40] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In Neil D. Jones and Xavier Leroy, editors, *POPL*, pages 14–25. ACM, 2004.

[41] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[42] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *STTT*, 9(5–6):505–525, 2007.

[43] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1–3):217–239, 2005.

[44] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The Barcelogic SMT solver. In Gupta and Malik [89], pages 294–298.
http://www.lsi.upc.es/~oliveras/bclt-main.html.

[45] Sascha Böhme, K. Rustan M. Leino, and Burkhart Wolff. HOL-Boogie—an interactive prover for the Boogie program-verifier. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 150–166. Springer, 2008.

[46] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible markup language (xml) 1.1.
http://www.w3.org/TR/xml11/.

[47] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In Stefan Kowalewski and Anna Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009.
http://fmv.jku.at/boolector/.

[48] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The mathsat 4smt solver. In Gupta and Malik [89], pages 299–303.
http://mathsat.itc.it/.

[49] Bruno Buchberger, Adrian Craciun, Tudor Jebelean, Laura Kovács, Temur Kutsia, Koji Nakagawa, Florina Piroi, Nikolaj Popov, Judit Robu, and Markus Rosenkranz. Theorema: Towards computer-aided mathematical theory exploration. *J. Applied Logic*, 4(4):470–504, 2006.

[50] Adam L. Buchsbaum, Loukas Georgiadis, Haim Kaplan, Anne Rogers, Robert Endre Tarjan, and Jeffery Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM J. Comput.*, 38(4):1533–1573, 2008.

[51] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.

[52] Patrice Chalin. Early detection of JML specification errors using ESC/Java2. In *Proceedings of the 2006 conference on Specification and verification of component-based systems*, page 32. ACM, 2006.

[53] Craig Chambers and Gary T. Leavens. Typechecking and modules for multi-methods. In *OOPSLA*, pages 1–15, 1994.

[54] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[55] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 1999.

[56] Curtis Clifton, Todd D. Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.*, 28(3):517–575, 2006.

[57] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.

[58] David R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, 2005.

[59] David R. Cok and Joseph Kiniry. Esc/java2: Uniting esc/java and jml. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.

[60] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.

[61] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm. `http://www.cs.rice.edu/~keith/EMBED/dom.pdf`, 2000.

[62] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[63] Werner Damm and Holger Hermanns, editors. *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3–7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*. Springer, 2007.

[64] Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, 2006.

[65] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In Ramakrishnan and Rehof [149], pages 337–340. `http://research.microsoft.com/en-us/um/redmond/projects/z3/index.html`.

[66] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Transactions on Algorithms*, 6(1), 2009.

[67] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[68] Lee R. Dice. Measures of the amount of ecologic association between species. *Ecological Society of America*, 26(3):297–302, July 1945.

[69] Edsger Wybe Dijkstra. The humble programmer. EWD340.

[70] Edsger Wybe Dijkstra. Guarded commands, non-determinancy and a calculus for the derivation of programs. In Friedrich L. Bauer and Klaus Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *Lecture Notes in Computer Science*, pages 111–124. Springer, 1975.

[71] Dino Distefano and Matthew J. Parkinson. jStar: towards practical verification for Java. In Gail E. Harris, editor, *OOPSLA*, pages 213–226. ACM, 2008.

[72] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. GraphViz—open source graph drawing tools. In *Graph Drawing*, pages 483–484, 2001.

[73] Andrei P. Ershov. On programming of arithmetic operations. *Commun. ACM*, 1(8):3–9, 1958.

[74] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In Andrew Kennedy and François Pottier, editors, *ML*, pages 12–19. ACM, 2006.

[75] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Damm and Hermanns [63], pages 173–177.
`http://why.lri.fr/`.

[76] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.

[77] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 191–202, New York, NY, USA, 2002. ACM.

[78] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *POPL*, pages 193–205, 2001.

[79] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[80] Robert W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19–32):1, 1967.

[81] Edward Fredkin. Trie memory. *Commun. ACM*, 3:490–499, September 1960.

[82] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Adison–Wesley, 1995.

[83] Loukas Georgiadis, Robert Endre Tarjan, and Renato Fonseca F. Werneck. Finding dominators in practice. *J. Graph Algorithms Appl.*, 10(1):69–94, 2006.

[84] Oded Goldreich. *Computational complexity: a conceptual perspective*. Cambridge University Press, 2008.

[85] Martin Charles Golumbic. *Algorithmic graph theory and perfect graphs*. Elsevier, second edition, 2004.

[86] Ronald L. Graham, Donald Ervin Knuth, and Oren Patashnik. *Concrete Mathematics*. Adison–Wesley, 1998.

[87] Radu Grigore, Julein Charles, Fintan Fairmichael, and Joseph Kiniry. Strongest postcondition of unstructured programs. In *Formal Techniques for Java-like Programs*, pages 6:1–6:7, New York, NY, USA, 2009. ACM.

[88] Radu Grigore and Michał Moskal. Edit and verify. In Silvio Ranise, editor, *FTP*, pages 101–112. University of Liverpool, September 2007. ULCS-07-018.

[89] Aarti Gupta and Sharad Malik, editors. *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7–14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*. Springer, 2008.

[90] Christian Haack and Clément Hurlin. Resource usage protocols for iterators. *Journal of Object Technology*, 8(4):55–83, 2009.

[91] Richard Wesley Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 24(2), April 1950.

[92] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[93] Nicholas J. A. Harvey. Algebraic algorithms for matching and matroid problems. *SIAM J. Comput.*, 39(2):679–702, 2009.

[94] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Marco A. A. Sanvido. Extreme model checking. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 332–358. Springer, 2003.

[95] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[96] Jochen Hoenicke, K. Rustan M. Leino, Andreas Podelski, Martin Schäf, and Thomas Wies. It's doomed; we can prove it. In Ana Cavalcanti and Dennis Dams, editors, *FM*, volume 5850 of *Lecture Notes in Computer Science*, pages 338–353. Springer, 2009.

[97] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison–Wesley, 2003.

[98] Charles Hoover. A methodology for determining response time baselines. In *Int. CMG Conference*, pages 85–94. Computer Measurement Group, 2006.

[99] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[100] David Hovemeyer and William Pugh. Finding bugs is easy. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA Companion*, pages 132–136. ACM, 2004.

[101] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[102] Bart Jacobs and Frank Pissens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, August 2008.
`http://www.cs.kuleuven.be/~bartj/verifast/`.

[103] Perry R. James and Patrice Chalin. Faster and more complete extended static checking for the java modeling language. *J. Autom. Reasoning*, 44(1–2):145–174, 2010.

[104] Mikolás Janota, Radu Grigore, and Michal Moskal. Reachability analysis for annotated code. In Arnd Poetzsch-Heffter, editor, *SAVCBS*, pages 23–30. ACM, 2007.

[105] Mikoláš Janota, Fintan Fairmichael, Viliam Holub, Radu Grigore, Julien Charles, Dermot Cochran, and Joseph R. Kiniry. CLOPS: A DSL for

command line options. In Walid Mohamed Taha, editor, *DSL*, volume 5658 of *Lecture Notes in Computer Science*, pages 187–210. Springer, 2009.

[106] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries — The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[107] Richard M. Karp. Reducibility among combinatorial problems. *Complexity of computer computations*, 43:85–103, 1972.

[108] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-aided reasoning: ACL2 case studies*. Springer, 2000.

[109] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[110] Stephen Cole Kleene. *Representation of Events in Nerve Nets and Finite Automata*. RAND Corporation, August 1951.

[111] Donald Ervin Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison–Wesley, 1968.

[112] Donald Ervin Knuth. Computer programming as an art. *Commun. ACM*, 17(12):667–673, 1974.

[113] Donald Ervin Knuth. Postscript about NP-hard problems. *SIGACT News*, 6(2):15–16, 1974.

[114] Donald Ervin Knuth. *The TEXbook*. American Mathematical Society and Addison–Wesley, 1986.

[115] Donald Ervin Knuth. The Stanford GraphBase: A platform for combinatorial algorithms. In *SODA*, pages 41–43, 1993.

[116] Shuvendu K. Lahiri and Shaz Qadeer. Verifying properties of well-founded linked lists. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 115–126. ACM, 2006.
http://research.microsoft.com/en-us/projects/havoc.

[117] Gary Leavens, Albert Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
http://www.eecs.ucf.edu/~leavens/JML/.

[118] Gary T. Leavens, David A. Naumann, and Stan Rosenberg. Preliminary definition of Core JML. Technical report, Stevens Institute of Technology, 2006.

[119] K. Rustan M. Leino. This is Boogie 2. KRML 178.

[120] K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.

[121] K. Rustan M. Leino and Ronald Middelkoop. Proving consistency of pure methods and model fields. In Marsha Chechik and Martin Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2009.

[122] K. Rustan M. Leino, Todd D. Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Sci. Comput. Program.*, 55(1–3):209–226, 2005.

[123] K. Rustan M. Leino and Rosemary Monahan. Automatic verification of textbook programs that use comprehensions. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*. Citeseer, 2007.

[124] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2010.

[125] Karl Lermer, Colin J. Fidge, and Ian J. Hayes. Formal semantics for program paths. *Electr. Notes Theor. Comput. Sci.*, 78, 2003.

[126] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[127] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 10(8):707–710, February 1966.

[128] Fangzhen Lin. On strongest necessary and weakest sufficient conditions. In *KR*, pages 167–175, 2000.

[129] Zdravko Markov and Daniel T. Larose. *Data mining the Web: uncovering patterns in Web content, structure, and usage*. Wiley-Interscience, 2007.

[130] Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In Zhiming Liu and Jifeng He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 400–419. Springer, 2006.

[131] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

[132] John McCarthy. A basis for a mathematical theory of computation, preliminary report. *AFIPS Joint Computer Conferences*, pages 225–238, 1961.

[133] J. Strother Moore. A mechanically verified language implementation. *J. Autom. Reasoning*, 5(4):461–492, 1989.

[134] J. Strother Moore. Inductive assertions and operational semantics. *STTT*, 8(4–5):359–371, 2006.

[135] Michał Moskal. Rocket-fast proof checking for SMT solvers. In Ramakrishnan and Rehof [149], pages 486–500.

[136] Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. CHESS: A systematic testing tool for concurrent software. Technical Report 149, Microsoft Research, 2007.

[137] George C. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94, 2000.

[138] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[139] Tobias Nipkow, editor. *Isabelle/HOL: a Proof Assistant for Higher-order Logic*. Springer, 2002.

[140] Frederick M. Noad. *The complete idiot's guide to playing the guitar*. Alpha Books, second edition, 2002.

[141] Diego Novillo. From source to binary: The inner workings of GCC. *The Red Hat Magazine*, December 2004. http://www.redhat.com/magazine/002dec04/features/gcc/.

[142] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *CADE*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[143] Terence John Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Softw., Pract. Exper.*, 25(7):789–810, 1995.

[144] Simon Peyton-Jones. How to write a good research paper. http://research.microsoft.com/en-us/um/people/simonpj/papers/giving-a-talk/writing-a-paper-slides.pdf.

[145] Gordon D. Plotkin. A structural approach to operational semantics, 1981.

[146] Gordon D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004.

[147] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.

[148] Vaughan Pratt. Enriched categories and the Floyd–Warshall connection. In *Proc. First International Conference on Algebraic Methodology and Software Technology, Iowa City*, pages 177–180. Citeseer, 1989.

[149] C. R. Ramakrishnan and Jakob Rehof, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*. Springer, 2008.

[150] Silvio Ranise and Cesare Tinelli. The SMT-LIB standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006.
http://www.smt-lib.org.

[151] Didier Rémy. Using, understanding, and unraveling the OCaml language. from practice to theory and vice versa. In Gilles Barthe, Peter Dybjer, Luis Pinto, and João Saraiva, editors, *APPSEM*, volume 2395 of *Lecture Notes in Computer Science*, pages 413–536. Springer, 2000.

[152] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[153] Martin C. Rinard and Darko Marinov. Credible compilation with pointers. In *Proceedings of the Workshop on Run-Time Result Verification*. Citeseer, 1999.

[154] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC / SIGSOFT FSE*, pages 267–276. ACM, 2003.

[155] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*, volume 1. Gulf Professional Publishing, 2001.

[156] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*, volume 2. Gulf Professional Publishing, 2001.

[157] Bernard Roy. Transitivité et connexité. *C. R. Acad. Sci. Paris*, 249:216–218, 1959.

[158] Arsenii Rudich, Ádám Darvas, and Peter Müller. Checking well-formedness of pure-method specifications. In Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, editors, *FM*, volume 5014 of *Lecture Notes in Computer Science*, pages 68–83. Springer, 2008.

[159] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for Java. In *ISSRE*, pages 245–256. IEEE Computer Society, 2004.

[160] Mayer D. Schwartz, Norman M. Delisle, and Vimal S. Begwani. Incremental compilation in Magpie. In *SIGPLAN Symposium on Compiler Construction*, pages 122–131. ACM, 1984.

[161] Raymond M. Smullyan. *Theory of formal systems*. Princeton University Press, 1961.

[162] Norman E. Steenrod, Paul R. Halmos, Menahem M. Schiffer, and Jean A. Dieudonne. *How to Write Mathematics*. American Mathematical Society, 1973.

[163] Alexander J. Summers and Sophia Drossopoulou. Considerate reasoning and the Composite design pattern. In Gilles Barthe and Manuel V. Hermenegildo, editors, *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 328–344. Springer, 2010.

[164] Kuo-Chung Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979.

[165] Microsoft The Spec$^\sharp$ Project. The Boogie benchmarks, October 2006. http://research.microsoft.com/en-us/projects/specsharp/.

[166] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot—a Java bytecode optimization framework. In Stephen A. MacKay and J. Howard Johnson, editors, *CASCON*, page 13. IBM, 1999.

[167] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In Tiziana Margaria and Wang Yi, editors, *TACAS*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2001. Tool not available online.

[168] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[169] Peter Vanbroekhoven, Gerda Janssens, Maurice Bruynooghe, and Francky Catthoor. A practical dynamic single assignment transformation. *ACM Trans. Design Autom. Electr. Syst.*, 12(4), 2007.

[170] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[171] Frédéric Vogels, Bart Jacobs, and Frank Piessens. A machine-checked soundness proof for an efficient verification condition generator. In *Proceedings of the 2010 ACM symposium on Applied Computing*, 2010. To appear.

[172] Philip Wadler. The expression problem. `http://www.daimi.au.dk/~madst/tool/papers/expression.txt`, 1998.

[173] Philip Wadler. Proofs are programs: 19th century logic and 21st century computing. *Dr. Dobbs Journal*, December 2000. Special supplement on Software in the 21st century.

[174] Jim Waldo. On system design. In Peri L. Tarr and William R. Cook, editors, *OOPSLA*, pages 467–480. ACM, 2006.

[175] Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.

[176] William E. Winkler. String comparator metrics and enchanced decision rules in the Fellegi–Sunter model of record linkage. In *Proceedings of the Section on Survey Research Methods*. American Statistical Association, 1990.