Credible Compilation

by

Darko Marinov

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2000

Author . . . .
Department of Electrical Engineering and Computer Science
September 5, 2000

Certified by . . . .
Martin C. Rinard
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Credible Compilation

by

Darko Marinov

Submitted to the Department of Electrical Engineering and Computer Science
on September 5, 2000, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This thesis describes a theoretical framework for building compilers that generate formal guarantees that they work correctly. Traditional compilers provide no such guarantees—given an original source program, a traditional compiler generates only a transformed executable program. The only way to investigate the correctness of a compilation is to run the transformed program on some sample inputs. Even if the transformed program generates expected results for these inputs, it does not ensure that the transformed program is indeed equivalent to the original program for all inputs.

Most previous research on compiler correctness focused on developing compilers that are guaranteed to correctly translate every original program. It is extremely difficult, however, to verify that a complex code, which implements a compiler, is correct. Therefore, a novel approach was proposed: instead of verifying a compiler, verify the result of each single compilation. We require the compiler to generate a transformed program and some additional information that enables a simple verifier to check the compilation. We call this approach *credible compilation*.

This thesis presents a formal framework for the credible compilation of imperative programming languages. Each transformation generates, in addition to a transformed program, a set of standard invariants and contexts, which the compiler uses to prove that its analysis results are correct, and a set of simulation invariants and contexts, which the compiler uses to prove that the transformed program is equivalent to the original program. The compiler has also to generate a proof for all the invariants and contexts. We describe in detail the structure of a verifier that checks the compiler results. The verifier first uses standard and simulation verification-condition generators to symbolically execute the original and transformed programs and generate a verification condition. The verifier then uses a proof checker to verify that the supplied proof indeed proves that verification condition. If the proof fails, the particular compilation is potentially not correct. Our framework supports numerous intraprocedural transformations and some interprocedural transformations.

Thesis Supervisor: Martin C. Rinard
Title: Associate Professor

Acknowledgments

Many people have helped me, directly or indirectly, to complete this work. With the previous sentence, I started the acknowledgments in my Bachelor's thesis. I then explicitly acknowledged only the people who helped me directly. It is now the perfect opportunity to correct the error I made.

First of all, I would like to thank my family—my sister Marina, my mother Ivanka, and my father Slobodan—for all they have done for me in my life. Their love and care for me were always enormous, and I can never repay them. I also appreciate that my parents let me make my own career decisions throughout my life.

I would like to thank my advisor, Martin Rinard, for all the personal and technical help he has provided to me during the last two years. I was fortunate that Martin, the "big boss," was with me in the numerous problematic situations I encountered. His care for students is truly exceptional, and if it were not for him, I would not be typing this now. I especially appreciate that Martin still believes in me, and I hope to justify that belief one day.

I would like to thank the present graduate students in Martin's group: Maria-Cristina Marinescu, Radu Rugină, C. Scott Ananian, Brian Demsky, Alexandru Sălcianu, Karen Zee, and Viktor Kunčak. As a group, "Martin's kids" made my life at MIT a much nicer experience. Each of them also helped me in various ways during my staying here. Maria, my "dearest office-mate," has been like my sister ever since I came to MIT. Radu, the "little boss," had enough patience to teach me program analyses. He and Viktor also provided valuable comments on the thesis. Alex helped me with course work while I was working on the thesis. Scott helped me make my working environment much more pleasant. Brian and Karen helped me correct English in the thesis. I also thank Rachel Allen for helping me with English grammar. All the errors (or is it "all errors"?) remaining in the text are my fault.

I would like to thank Konstantinos Arkoudas, the author of the Athena logical framework. Kostas helped me understand Athena and use it to implement a prototype credible compiler that I have not had time to describe in the thesis. I also thank George Necula for a discussion about proof carrying code.

I would like to thank my colleague and roommate Sarfraz Khurshid for many life lessons he indirectly taught me. Finally, I would like to thank Jonathan Babb for sharing his ideas and discussing life issues with a junior graduate student like me.

Contents

List of Figures

Chapter 1

Introduction

Compilers translate programs from one language to another. Typically, a compiler takes an input program written in a high-level programming language and generates an output program in a target machine language. A compiler usually consists of a front end, which translates the source program to some intermediate representation, a middle end, which transforms the program so that it executes more efficiently, and a back end, which translates the program from the intermediate representation to the machine language.

In most modern compilers, the middle end is structured as a sequence of optimization passes. Each optimization pass transforms the input program to an equivalent output program that is expected to execute faster or require less memory. It is well known that the optimizations are rarely optimal by any measure. We therefore call the optimization passes *transformations*. We distinguish transformations from *translations*. We use the term translations to refer to compiler phases, such as parsing or code generation, that translate the program from one representation to an essentially different representation.[1]

1.1 Traditional Compilation

Traditional compilers offer no formal guarantees that they operate correctly. Even the most reliable compilers can fail to compile a program correctly. The main problem with traditional compilers is that they fail silently; the compiled program is produced in a highly encoded form suitable for machine execution and not designed to be read by programmers. The only reasonable way a programmer can observe an incorrect compilation is by executing the compiled program and observing an incorrect execution. Executions are usually incorrect because of errors in the source program, and the programmer first inspects the source program. When the error is due to the compiler, it takes significantly more time and effort to discover that the error is actually not in the source program.

[1] In many compilers, low-level optimizations such as register allocation take place in the back end. It is possible to perform these optimizations without significantly changing the program representation. We therefore view even these low-level optimizations as transformations, not translations.

Additionally, the execution of the program depends on its input data, and the programmer can test the program only for some sample input data. If the execution is correct for those input data, it does not guarantee that the compiled program is correct for all input data. Furthermore, compiling the same source program with different optimizations produces, in general, different compiled programs. If one of those programs is tested and found correct, there is no guarantee that all of them are correct. Any compiler optimization may potentially introduce an error in the compiled program, and the programmer has to test each compiled program.

Compiler failures are terrible for programmers, but in practice, programmers infrequently encounter compiler errors. Production-quality compilers are among the most reliable software products and almost never incorrectly compile a program. However, producing an extremely reliable compiler requires a large development time. The result is that industry compilers are almost always many years old. They lag behind the advances in programming languages, compiler research, and computer architecture. Compiler maintainers rarely and slowly add new transformations to optimizing compilers. The reason is that a transformation can be added to traditional compilers only when it is correctly *implemented* to work for all input programs. A compiler transformation usually requires a complex implementation that is extremely difficult to formally verify using standard program verification techniques. Therefore, compiler developers only test the implementation for some large class of input programs and add the transformation to the compiler when they believe that it is working correctly.

1.2 Credible Compilation

This thesis presents a fundamentally different approach to building optimizing compilers: implement compiler transformations which, given an input program, generate an output program and some additional information, including a machine-verifiable proof, that the output program is equivalent to the input program. After each transformation, an automated verifier checks whether the supplied proof indeed guarantees that the transformed output program is equivalent to the given input program. If the proof fails, the transformed program is potentially not equivalent, and the compiler should not use this transformation for this input program. The compiler may still be able, though, to compile this input program to the final machine form by omitting this transformation, and this transformation may work correctly for other input programs. Thus, at each pass the verifier checks only *one* particular transformation for *one* particular input program and either accepts or rejects the output program. We call this approach *credible compilation.*

We next briefly mention the results on which we directly build our work. Martin Rinard [41] introduced the name *credible compiler* and described basic techniques for building a compiler that generates equivalence proofs. We advance these previous techniques and this thesis presents a more elaborate theoretical framework for credible compilation. This framework supports credible compiler transformations, and not translations. Rinard [41] also briefly describes credible code generation. In principle, the approach of generating equivalence proofs for each single compiler run and checking them automatically can be used for building a whole compiler, including the translations from one representation to another. The idea of credible translations appeared first in papers by Cimatti et al. [12] and Pnueli et al. [40]. These two papers consider simple programs, consisting of only one loop, and non-optimizing translations, whereas our work considers more complex programs and compiler optimizations. We review the related work in detail in Chapter 7.

We next present the motivation for our work. Credible compilation would provide many practical benefits compared to traditional compilation. Since a transformation has to produce a proof that it operated correctly, the compilation failures are not silent any more. It is immediately visible when a transformation operates incorrectly. This gives the programmer a much higher level of confidence in the compiler and saves the programmer time because she never mistakes a compiler bug for a bug in her own program.

Since credible transformations need to produce a proof, an implementation of a credible transformation is somewhat more complex than an implementation of a traditional transformation. Nevertheless, credible compilation would make compiler development faster, because it is easier to find and eliminate compiler errors. It would also allow adding new transformations into the compilers more aggressively; compiler developers could add a transformation even when the implementation is not correct for all possible input programs. There is no need to verify and trust the implementation of a transformation. It is only an implementation of a verifier for a credible compiler that needs to be trusted, and the verifier is much simpler to build than compiler transformations.

1.3 Non-Credible Compilation Example

In this section we try to clarify two common misunderstandings about credible compilation. The first misunderstanding is that the added complexity of credible compilation is unnecessary because traditional compilers are extremely reliable. We address this misunderstanding by presenting an example program that exposes a bug in an existing industrial-strength compiler. The second misunderstanding is that people do not clearly distinguish safety proofs of the output program from equivalence proofs that involve both the input program and the output program. We address this misunderstanding by presenting an example of a safe output program that does not preserve the semantics of the input program.

Figure 1-1 shows our example C program. The compiler is the Sun Microsystems C compiler, version `WorkShop Compilers 4.2 30 Oct 1996 C 4.2`.[2] This compiler is over three years old, but it is still the default C compiler on the main server of the Computer Architecture Group at MIT.

The program contains three loops of the form:

```
for (i = 0; i < 10; i++) *p = (*p) + i;
```

[2] We have reported the bug, but it had already been observed earlier and corrected in the next versions.

We wrote the loop body in a verbose mode[3] to point out that we do not use any potentially unsafe pointer arithmetic operation. All the loop does is add the numbers 1 to 10 to the variable pointed to by the pointer p. In the three loops we vary where p can point to.

```
#include <stdio.h>
int i, j, x;
void main() {
  int *p;

  p = &x; /* p->x */
  *p = 0; for (i = 0; i < 10; i++) *p = (*p) + i;

  p = &i; /* p->i */
  *p = 0; for (i = 0; i < 10; i++) *p = (*p) + i;
  j = i;

  if (x > 0) p = &i; else p = &x; /* p->x or p->i; actually p->i */
  *p = 0; for (i = 0; i < 10; i++) *p = (*p) + i;

  printf("i=%2d, j=%2d, x=%2d\n", i, j, x);
}
```

Figure 1-1: Non-Credible Compilation Example Program

If the compiler detects that p can point to only one variable, say x, within the loop, it can replace the dereferencing *p with the direct referencing of that variable:

```
for (i = 0; i < 10; i++) x = x + i;
```

This transformation generates an equivalent program even when p can point only to the loop index variable i.

Even when the compiler cannot detect where p exactly points to, but p does not change within the loop[4], the compiler may still be able to optimize the program. The compiler can hoist dereferencing, which is invariant, out of the loop by using a new temporary variable t:

```
t = *p; for (i = 0; i < 10; i++) t = t + i; *p = t;
```

However, the question is: are the two loops equivalent in all cases? The answer is: no; if p can point to i, the transformed loop does not produce the same result as the original loop. This is exactly what the example program exploits. In the first loop,

[3] In C, it would be usually written just as *p+=i.

[4] In C, it is possible to make, using type-unsafe casting, a pointer that points to itself and writing to *p would change p itself in that case.

p points to x, in the second loop p points to i, but in the third loop p is written so that it may point to either i or x, although it actually always points to i.

We first compiled the program without any optimizations. Running the resulting executable gives the expected output:

```
i=15, j=15, x=45
```

We then compiled the program with a high optimization level (-xO4). Running the resulting executable now gives a different output:

```
i=45, j=15, x=45
```

We looked at the assembly code generated by the compiler to find the cause for the different outputs. As expected, the reason is that the compiler aggressively applied dereference hoisting in the third loop. In the first and the second loop, the compiler correctly found that p can point only to x, respectively i, and replaced *p with x, respectively i. However, in the third loop, the compiler failed to find that p can point only to i. Even worse, the compiler incorrectly assumed that p cannot point to i, and performed the transformation with a new temporary variable as described above. This example shows that even a production-quality compilers can incorrectly optimize a program.

We next point out the difference between a compiler that generates equivalence proofs for the input program and the output program and a compiler that generates proofs only about the properties of the output program. One such property is safety, in particular type and memory safety, of the output program. Some research compilers, such as Necula and Lee's Touchstone [38] and Morrisett et al.'s Popcorn [31], generate an evidence of the safety of the compiled program. These compilers are useful in situations where *code consumers*, who run the compiled programs, do not trust *code producers*, who produce these compiled programs. Code producers can use these compilers to compile the original programs and to obtain the evidence together with the compiled program. Code consumers can then receive the compiled program and the evidence from untrusted sources, for example, by downloading the program from the Internet. Before running the program, the code consumer can use the evidence to verify the safety of the program. But, the fact that the compiler has generated the correct evidence of the safety of the compiled program does not imply that the compiler has generated a compiled program that is equivalent to the original program.

For instance, in the presented example, checking only the type safety would not detect the compiler bug.[5] The reason is that the incorrect transformation does not violate type safety. Both the loop with *p and the incorrectly transformed loop with t type-check—type is preserved, but the values are changed. Therefore, even a compiler that uses strongly typed intermediate representation could make the same error. Further, a compiler that generates evidence of the safety of the transformed

[5] The full C language is not type-safe, but observe that the example program does not use any low-level, unsafe C features, such as arbitrary pointer arithmetic or type casts. Hence, this example can be regarded as a program in a type-safe subset of C.

program could generate correct evidence for the incorrectly transformed program. In contrast, a compiler that generates equivalence proofs could *not* generate a correct proof for the incorrectly transformed program. In conclusion, compilers that generate guarantees only about the transformed program are good for sending the transformed program to code consumers. However, such compilers are not good enough for code producers; code producers need credible compilers.

Chapter 2

Overview

In this chapter we present in more detail the structure of a credible compiler. The main idea of credible compilation is that the compiler generates a proof that it correctly transformed the input program. We first define when a compiler transformation is correct. We next describe the organization of a credible compiler and what exactly the compiler has to prove. Finally, we summarize the results of our work.

2.1 Transformation Correctness

In this section we define more precisely our requirement for a correct transformation. So far we have used the intuitive notion of the equivalence between the output program and the input program. We first argue that the requirement that the programs be equivalent is too strong for transforming non-deterministic programs. We then define our requirement for a correct transformation to be that the output program *simulates* the input program.

Usually, a transformation is defined to be correct if it preserves the meaning of the program, as defined by the semantics of the language. Informally, a transformation is considered correct if the transformed program is semantically equivalent to the original program—for all possible inputs, the two programs, given the same input, produce the same output. What is considered as input and output depends on the semantics of the programs. We discuss several examples, and additionally the requirements that could be imposed on a compiler:

- If the programs are non-deterministic, then the original program could itself generate, for the same input, different results in different executions. We could then require the transformed program to be able to also generate all those results, or only some of them.

- The original program may not terminate for some input. We could require the transformed program also not to terminate for that input. Conversely, we could require the transformed program to terminate whenever the original program terminates.

- The original program may end up in an error state for some input (e.g., because of the resource bound violation when the program executes on a real machine). We could require the transformed program to also end up in the error state. Conversely, we could require the transformed program to end up in the error state only if the original program ends up in the error state.

- The output of a program, or more precisely, the observable effects of a program execution, may include more than the final state. We could require the compiler to preserve all the observable effects, or only some of them.

Clearly, the correctness criterion should specify that the transformed program can generate only the results that the original program can generate. However, we do *not* require the transformed program to be able to generate *all* the results that the original program may generate. This allows the transformed program to have less non-determinism than the original program. The reason is that the compiler transformations bring the program closer to the final executable form, and the programs execute mostly on deterministic machines. Therefore, the compiler need not preserve the non-determinism that might be present in the original program.

We specify our requirement using the notion of *simulation*. Informally, program P_1 simulates program P_2 if P_1 can generate only the results that P_2 generates. More precisely, for all executions of P_1, there exists an execution of P_2 which generates the same output (for the same input). Additionally, if P_1 may not terminate (i.e., P_1 has an infinite execution) for some input, then P_2 also may not terminate for that input. We require the compiler to generate a transformed program that simulates the original program. We give a formal definition of simulation in Section 4.4. Our framework can easily support a stronger notion of correctness, namely *bi-simulation*. Programs P_1 and P_2 bi-simulate each other if P_1 simulates P_2 and, conversely, P_2 simulates P_1. We could require the compiler to prove that the transformed program bi-simulates the original program by proving both that the transformed program simulates the original program and that the original program simulates the transformed program.

In general, simulation is not a symmetric relationship between programs. If P_1 simulates P_2, then P_2 may generate more results than P_1, and therefore P_2 need not simulate P_1. This means that when P_1 simulates P_2, the two programs need not be equivalent in the sense that they can generate the same set of results. However, if programs are deterministic, they can generate only one result. Therefore, when the transformed program simulates the original program that is deterministic, the two programs are equivalent.[1] In our basic framework, presented in Chapter 4, we consider *almost deterministic* programs. We call the programs almost deterministic because the result of a program execution may depend on the unknown values of uninitialized local variables, although we consider a language without non-deterministic constructs. Therefore, we will sometimes use the term equivalence, instead of simulation, to refer to the correctness requirement.

[1] Observe that P_1 and P_2 are equivalent if P_1 bi-simulates P_2.

2.2 Credible Compiler Structure

In this section we first compare the general structures of a traditional compiler and a credible compiler. The main difference is that a credible compiler has a *verifier* that checks the results of the compiler. The verifier uses some additional information that the compiler generates beside the output program. We briefly discuss the additional information, and we describe the general structure of the verifier.

Figure 2-1 shows the simplified structure of a traditional optimizing compiler. First, the front end translates the input program from the source language to the intermediate representation. Next, the transformations, which include the optimizations and the back end passes that are not highly machine dependent, transform the program within the intermediate representation. Finally, the code generator produces the machine code. Clearly, for a compilation to be correct, all passes need to be correct and to produce an output program that simulates the input program. In traditional compilation there is no checking of the results generated by any of the passes; they are all blindly trusted to be correct.

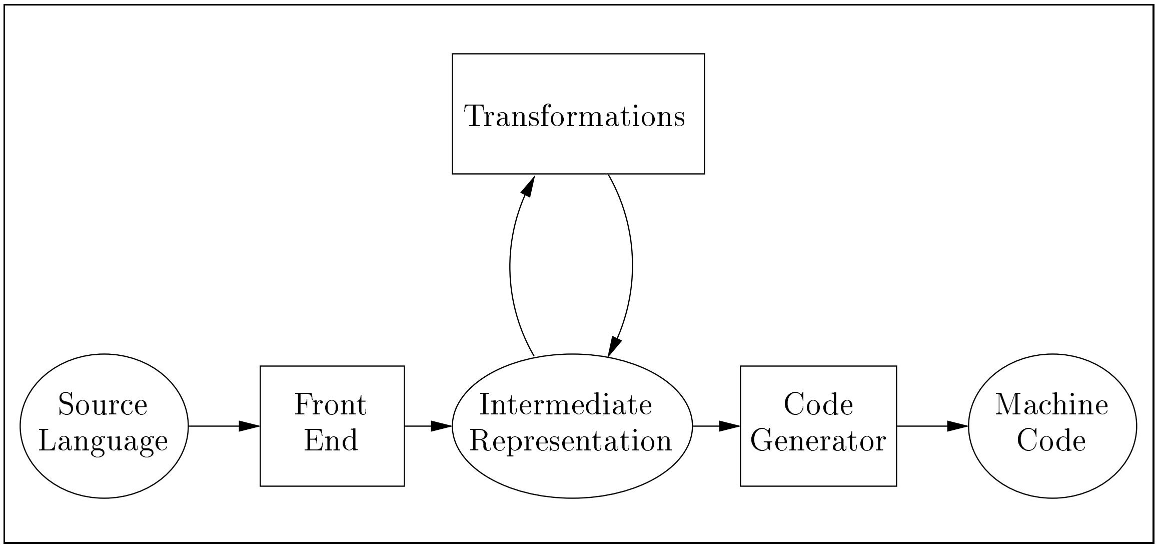

Figure 2-1: Structure of a Traditional Compiler

Figure 2-2 shows the simplified structure of a credible compiler. It differs from a traditional compiler in that there is checking of the results generated by the transformation passes. Since the transformations are not blindly trusted, we represent them as a "black box" in the figure. After each transformation, the verifier checks that the output program simulates the input program. To check the simulation of the two programs, the verifier uses the additional information that the transformation generates beside the output program. We first argue why it is necessary that the transformations generate some additional information, and then we describe how the verifier uses that information for checking.

At first glance, it seems possible that a transformation need only generate the output program, and the verifier can check the simulation of the two programs. However,

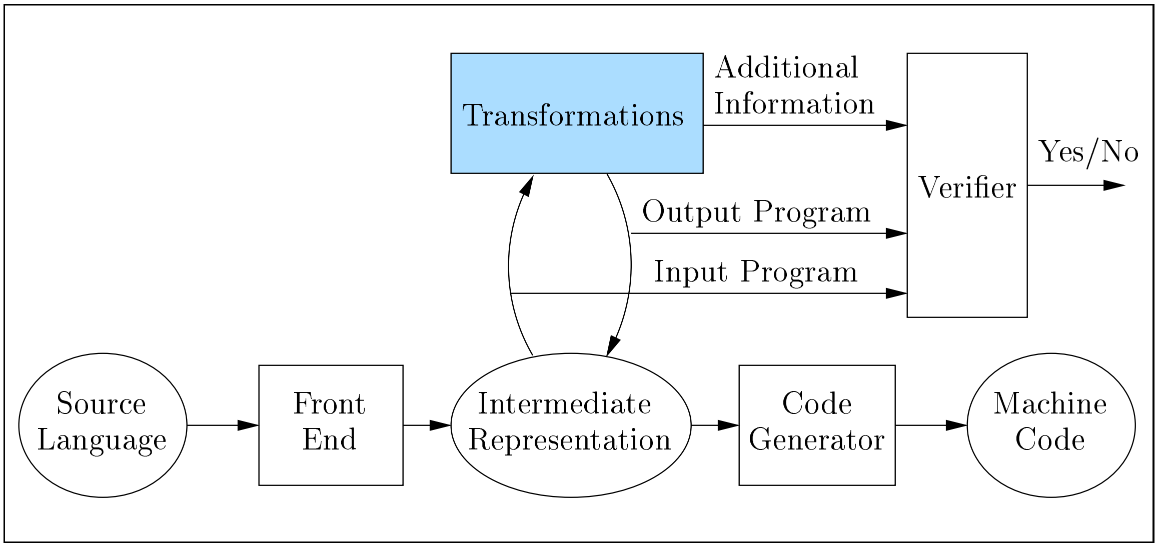

Figure 2-2: Structure of a Credible Compiler

directly building such a powerful verifier for arbitrary programs in sufficiently expressive languages is not a good idea. First, from theory we know that it is undecidable to determine the equivalence/simulation of two arbitrary programs. Hence, a verifier for the general case cannot be built. In practice, it is possible to build a big verifier that would check the results of common compiler transformations. However, building a big verifier which is itself not verified only shifts the possibility of introducing errors from the compiler implementation to the verifier implementation. Additionally, such a verifier may need to be modified each time a new transformation is added to the compiler. Therefore, credible compiler transformations need to generate some additional information that allows their results to be checked with a relatively simple verifier. We explain later what we mean by relatively simple.

We next present the additional information that a credible compiler transformation generates. Conceptually, the compiler generates a set of *contexts* and a *proof* that those contexts hold. A compiler usually applies a transformation in two steps:

- In the analysis step, the compiler analyzes the input program to determine the properties relevant for the transformation.

- In the transformation step, the compiler changes the input program, taking into account the results of the analysis, and generates an output program.

Our approach to credible compilation supports this two-step organization. For each step, a credible compiler generates a set of contexts:

- *Standard contexts* express properties of only one program. The compiler uses the standard contexts to represent the analysis results.

- *Simulation contexts* express the correspondence between two programs. The compiler uses simulation contexts to represent the simulation relationships between the input and output programs.

Each context contains a set of *invariants*. (Contexts also contain some other additional information, which we present later in the text.) More precisely, each standard context contains a set of *standard invariants* and each simulation context contains a set of *simulation invariants*. (We introduce several other concepts that have both standard and simulation form; we omit standard and simulation when it is clear from the context or when we refer to both.)

The standard and simulation invariants are formulas in a logic. (We present in Section 4.2 the details of the logic that we use, which is an extension of first-order predicate logic.) If all the invariants hold, then all the contexts hold, and the output program simulates the input program. The verifier does not try to prove that the contexts hold. Instead, the input to the verifier consists of the two programs, the contexts, and additionally a proof that those contexts hold for those programs. The verifier only checks that the proof indeed shows that the contexts hold.

Both the contexts and the proof are conceptually generated by the compiler. In practice, a credible compiler consists of two parts: a part that actually performs the transformations and generates the output program and the contexts and a part that proves that the contexts hold. We call the latter part the *proof generator*. We use this term, instead of the common *theorem prover*, to point out that a credible compiler does not use a general purpose theorem prover but a very specialized one.

2.2.1 Credible Compiler Verifier

We next describe the organization of a verifier for a credible compiler and what exactly the proof generator has to prove. Figure 2-3 shows the detailed structure of the credible compiler transformations. The module that performs the transformations and the proof generator are shown as "black boxes" because they can be implemented in any arbitrary way. They are not trusted, but the verifier checks their results. The verifier, however, needs to be trusted. The verifier consists of two parts: the *verification-condition generator* and the actual *proof checker*. Before we proceed to describe the verifier parts, we explain what a relatively simple verifier means. On the one hand, the proof checker, and thus the verifier, cannot be too simple since the expected proofs are non-trivial. On the other hand, the verifier should still be simpler to implement and verify, using standard program verification techniques, than the compiler transformations.

The verification-condition generator (VCG) for a credible compiler consists of two parts. We call them the *standard verification-condition generator* (StdVCG) and the *simulation verification-condition generator* (SimVCG). The StdVCG takes as input one program at a time (be it compiler input or output) and standard contexts for that program. We postpone the details of how the StdVCG works for the example in Section 3.1.1 and we give the full algorithm in Section 4.3.3. Suffice to say that the StdVCG symbolically executes the given program. The output of the StdVCG is the *standard verification condition* (StdVC) for the given program and its standard contexts. The StdVC is a logic formula whose validity implies that the given contexts hold for the given program. Namely, the results of the compiler analysis are correct if the StdVC is valid.

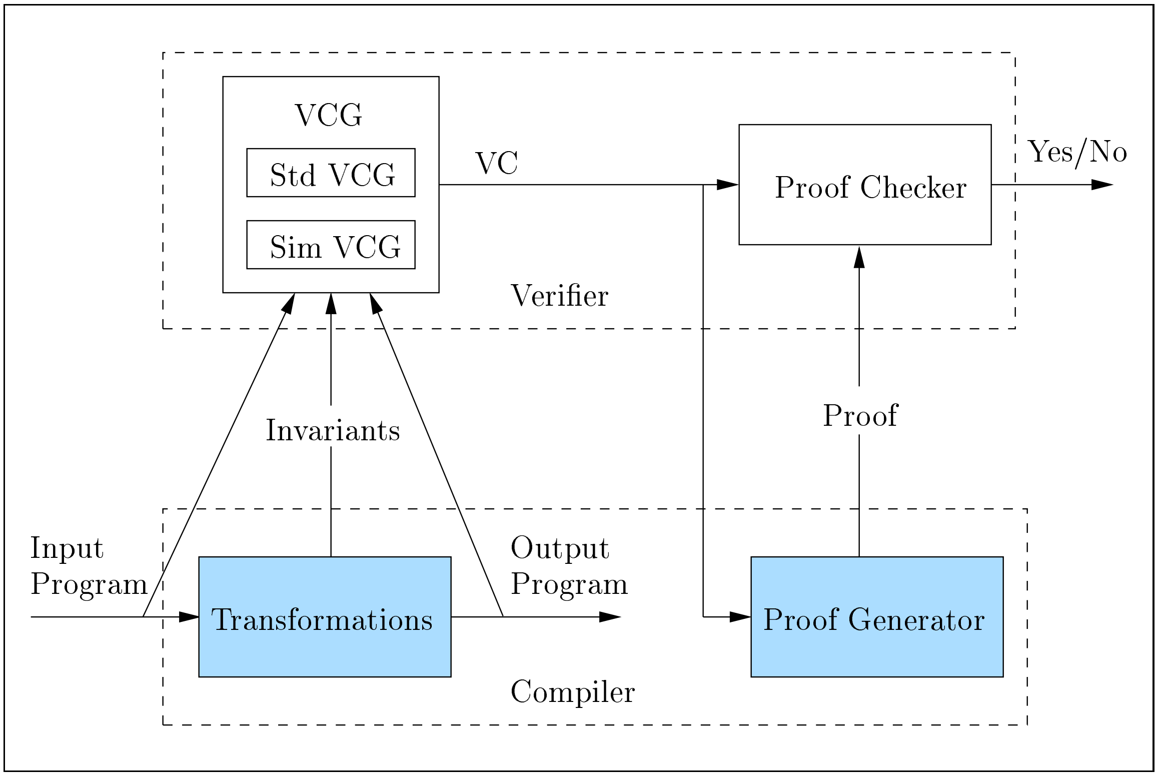

Figure 2-3: Verifier for a Credible Compiler

The SimVCG takes as input *two* programs (both the compiler input and output programs) and simulation contexts for them. Again, we postpone the details of how the SimVCG works for the example in Section 3.2.1 and we give the full algorithm in Section 4.4.3. Suffice to say that the SimVCG symbolically executes both given programs. The output of the SimVCG is the *simulation verification condition* (SimVC) for the given two programs and their simulation contexts. The SimVC is a logic formula whose validity implies that the given contexts hold for the given program. Namely, the result of the compiler transformation is correct (i.e., the output program simulates the input program) if the SimVC is valid.

The verifier for a credible compiler works as follows. It first takes the compiler input and output programs and attached contexts, and uses the VCG to generate the standard and simulation verification conditions for those programs. We call the conjunction of those verification conditions the *verification condition* (VC) for those two programs. The VCG does not prove that the VC is valid. The VCG only performs syntactic (and static semantic) checks on the programs and the contexts; the VCG rejects the programs if the compiler output is ill-formed. The verifier next uses the proof checker to verify that the proof provided by the proof generator actually proves the particular VC. If the proof checker rejects the proof, the compiler considers the transformed program to be incorrect and continues transforming the input program. Otherwise, the transformed program simulates the input program, and the compiler continues transforming further the transformed program.

2.3 Summary

In this section we first briefly present the previously published results on credible compilation [41, 42]. The initial work on credible compilation did not use VCG. We describe how using VCG in the verifier reduces the size of the proofs that the compiler needs to generate. We next present the scope of this thesis and we finally list the contributions of the thesis.

Rinard [41] describes the basic techniques for building credible compilers. The main idea is that the compiler generates a set of standard and simulation invariants together with the output program. The compiler then proves that it correctly transformed the input program by proving that these invariants hold for the input and output programs. Rinard devised a set of rules for proving that standard and simulation invariants hold for two programs.

The rules for standard invariants use a variation of the Floyd-Hoare rules [16, 23] for proving properties about one program. These rules propagate an invariant backward (opposite to the flow of control) through the program until another invariant is reached, at which point the compiler should prove that the reached invariant implies the propagated invariant. The rules for simulation invariants work in a conceptually similar way. However, those rules propagate simulation invariants through both programs. To the best of our knowledge, the simulation invariants and related rules were first introduced for proving compiler optimizations correct in [41]. The simulation invariants are similar to the (bi-)simulation relations in concurrency theory, but the techniques and applications used in that context are completely different. An overview of concurrency theory can be found in the article by Milner [30].

The initial work on credible compilation presented the rules for standard and simulation invariants as proof rules in a logic. The rules were derived from the structure of the programs, and therefore involved syntactic elements, for instance: a program contains a particular statement. In general, it is possible to encode the proofs directly using those rules, without using a VCG in the verifier. However, using a VCG dramatically decreases the size of the proofs. The reason is that the VCG performs syntactic checks on the programs and the invariants while generating the VC. The compiler can encode the proofs of VC using rules which do not include (many) syntactic elements from programs. The drawback of using a VCG is that the verifier, whose implementation needs to be trusted, gets larger.

We used the original rules for invariants as a guidance for making VCG, in particular the algorithm for the SimVCG. As described, the original rules for the simulation invariants did not give a precise algorithm for their application; they are just proof rules. This is somewhat analogous to type reconstruction—the typing rules provide a way to check the well-typedness of a program given the types of variables, but the rules do not provide a direct way to come up with those types when they are not given. However, an algorithm for type reconstruction infers those types, whereas the algorithm for the SimVCG does not try to infer any properties. The SimVCG simply requires the compiler to provide more additional information.

Our approach to using the VCG is motivated by Necula's PhD thesis [35]. In fact, the original incentive came through personal communication with George Necula at the PLDI '99 conference. Necula attributes the verification-condition generation to Floyd and King; the term "verification condition" is itself introduced by Floyd [16], and the concept of verification-condition generation is explained by King [27]. Those verification-condition generators are for only one program, or, in terms of credible compilation, they are standard verification-condition generators, whereas our VCG also includes a SimVCG. Recently, Necula [36] presented his work on verification of compiler transformations which uses a strategy similar to SimVCG. We compare that work to ours in Chapter 7, after we describe the details of our approach.

2.3.1 Scope

This thesis describes a theoretical framework for credible compilation. We next list questions that arise in the context of credible compilation and we explain how we address those questions in this thesis. We also briefly present our initial experience with a small prototype of a credible compiler.

- What is the language that is compiled?

  We present a framework for compiling imperative programming languages. The intermediate representation that we use can be considered as a subset of the C programming language [26]. The introductory paper on credible compilation [41] described a framework for a rather simple language: the programs consisted of only one procedure and operated only on simple integer variables. Rinard and Marinov [42] then extended the language with C-like pointers.

  In the basic framework, presented in Chapter 4, we extend the initial language for credible compilers with procedures[2]. Adding procedures to the language is important because it makes the language more realistic. In our basic language we consider the simple semantics of programs that execute on an idealized abstract machine without error states. The state of an execution is observed only at the end of the execution, if the program terminates at all. In Chapter 6 we discuss how to extend the basic framework to handle other common constructs of imperative programming languages. We point out that the framework which we present can formally handle only a subset of a C-like language. The main obstacle for a formal treatment of the full C language is the semantics of "undefined" C constructs, such as out-of-bound array accesses. We also discuss in Chapter 6 some limitations of our current framework.

- What transformations are supported by the framework?

  We say that a framework *supports* a transformation if that framework allows the compiler to prove that the results of that transformation are correct. Clearly, a framework which supports more transformations is preferable to a framework

[2] We use the term procedure to refer to subprograms because, for simplicity, we consider only the subprogram calls which are statements, and not expressions. The procedures do not return a result directly, but they can modify the memory state. We use the term function to refer to the meta-language objects.

which supports fewer transformations. Due to fundamental undecidability constraints, we cannot hope to develop a general framework which would allow the compiler to *automatically* prove that all possible transformations are correct. However, we want to develop a framework that allows the compiler to prove at least that the results of "standard transformations" are correct. The term "standard transformations" loosely refers to the common transformations performed by industry quality optimizing compilers.

The framework that we present supports numerous "standard" intraprocedural transformations, ranging from constant propagation to induction variable elimination to loop unrolling. Although it is presumptuous to say that a framework supports all transformations, we are not aware of any "standard" transformation that is not supported. The framework also supports some interprocedural analyses and whole program transformations.

In general, the compiler can transform any number of procedures to generate the transformed program. Our framework is designed so that the compiler proves the simulation relationships between pairs of procedures. To prove that the transformed program simulates the original program, the compiler then has to prove that the starting procedure[3] of the transformed program simulates the starting procedure of the original program.

The use of multiple *standard contexts* allows the compiler to prove correct even the results of some context-sensitive interprocedural analyses. (We describe the concept of standard contexts through the example in Section 3.1, and we formalize standard contexts in Section 4.3.) The use of *simulation contexts* allows the compiler to prove correct the results of some interprocedural transformations, such as procedure specialization. (We introduce the concept of simulation contexts through the example in Section 3.2, and we formalize simulation contexts in Section 4.4.) Simulation contexts also support reordering accesses to global variables across the procedure calls.

The framework, however, does not support procedure inlining and related transformations that change the structure of the call graph. The "culprit" is the simulation verification-condition generator. It operates on two procedures at a time and requires that whenever the transformed procedure reaches a call site, the original procedure also reaches a call site. (We present details on this later in the text.) Since procedure inlining is an important optimization, we can extend the framework to support it by adding a specialized part to the verifier that separately checks only inlining. In principle, support for any transformation can be added as a specialized check for that transformation. However, doing so for every transformation would make the verifier prohibitively large, as large as the compiler itself.

- Can the compiler transformations generate the invariants (and other required

[3] The *starting procedure* for a program is the procedure where the execution of the program starts; in C, the starting procedure is called `main`.

additional information, except for the proof)?

The compiler can generate the standard invariants simply by generating the formulas that represent the compiler analysis results. It is only that the language used for formulas should be expressive enough so that the compiler can indeed represent its results. The compiler is also able to automatically generate the simulation invariants, because intuitively the compiler "knows" how it performs the transformation and which entities in the output program correspond to which entities in the input program.

The approach in which the compiler generates some additional information is also explained by Morrisett et al. [32], who credit Necula and Lee [34,37]. They used the approach to build their compilers, Touchstone [38] and Popcorn [31]. However, both of these compilers prove only the properties of the output program, more precisely the type safety of the output program. In terms of credible compilation, the additional information that those compilers generate is only a set of standard invariants. In contrast, credible compilers also generate a set of simulation invariants. The simulation invariants are crucial for the concept of credible compilation because they allow the compiler to prove that the output program simulates the input program.

- Can the proof generator automatically generate a proof?

  The proof generator that accompanies a credible compiler generates proofs for the standard and simulation verification conditions. The main requirement for the proof generator is that it needs to be *fully automatic*. For each compiler transformation and analysis, there needs to be a decision procedure that can prove the SimVC and StdVC, respectively, generated for every possible input program. Note that the compiler developer can determine the general structure of the verification conditions for each transformation and analysis. Namely, each VC is a formula that depends on the invariants (and on the VCG). From the placement of the invariants and the general structure of their formulas, it is possible to find the general structure of the VC.

  We believe that it is possible to develop a decision procedure that can prove all verification conditions of the particular general structure. The reason is that the compiler developer knows why a transformation is correct, and potentially has a meta-proof that shows the transformation to be correct for all input programs. Developing a decision procedure is then a matter of translating the meta-proof into an algorithm that generates a proof for each possible instance of verification conditions. The complexity of the decision procedures depends on the particular transformations. The whole proof generator, which combines the decision procedures, needs only to be as powerful as the transformations whose results it needs to prove.

  For example, consider a proof generator that needs to prove the results of constant propagation, constant folding, and algebraic simplifications. The proof generator for constant propagation requires only a relatively simple decision procedure that uses a few logic rules, such as the congruence rule for equality, and does not need to "know" anything about the numbers. The proof generator for constant folding needs a more sophisticated decision procedure that uses arithmetic rules. Further, the proof generator for algebraic simplifications needs an even more sophisticated decision procedure that uses algebraic rules. However, this does not imply that every new transformation requires a new decision procedure. Many compiler transformations may have the same general structure of the verification conditions and can thus share the same decision procedure. For instance, the explained proof generator could prove the results of copy propagation and even common subexpression elimination. We do not consider the proof generator further in this thesis.

An implementation of a credible compiler can answer the last two of the listed questions: is it possible for a credible complier to generate the required additional information and to automatically prove the verification conditions. Additional pragmatic issues in the context of credible compilation are the difficulty of generating the proofs, the size of the generated proofs, and the difficulty of checking the proofs. To explore these issues, we have started developing a prototype of a credible compiler. We have implemented a small system for the language without procedures, but with pointers. We have used the Java programming language [8] for implementing a flow-sensitive pointer analysis and constant propagation analysis/transformation.

For proof representation and verification we use Athena [5,6], a *denotational proof language* [7] developed by Kostas Arkoudas at MIT. Athena is a flexible logical framework that allows a compact, procedural representation of proofs. This makes it possible to balance the division of labor between the proof generator and the proof checker, while retaining the full soundness guarantee. It also simplifies the construction of the compiler by simplifying the proof generator and allowing the compiler developer to easily generate proofs. Based on our initial positive experience with Athena, we believe that a key enabling feature to obtaining reasonable proof sizes and compiler complexity is the use of such a flexible logical framework. We do not present the prototype implementation in this thesis.

2.3.2 Contributions

The main contribution of the previously published work on credible compilation [41, 42] is introduction of a theoretical framework in which a compiler, using *simulation invariants*, can prove that it correctly transformed an input program. The contributions of this thesis to the existing work on credible compilation are the following:

- We extend the language for credible compilers with procedures.

- We use standard contexts and we introduce *simulation contexts* that allow the compiler to prove that the results of interprocedural analyses and transformations are correct.

- We present an algorithm for the verification-condition generator, in particular for the *simulation verification-condition generator*.

Chapter 3

Example

In this chapter we give an example of a credible compiler transformation. We explain what the compiler generates and how a verification condition-generator (VCG) generates a verification condition (VC). The compiler proves that it correctly performed the transformation by supplying a proof that the VC is valid. For clarity of presentation, we use a simple program fragment given in Figure 3-1. The fragment presents a procedure p in a program with a global variable g. Procedure p has two local variables i and c. We show a simple transformation on this example procedure, namely constant propagation. The example is written in C, but at the present time we can handle only a small subset of a C-like language within the credible compilation framework. On the other hand, our framework supports many other transformations that change the procedure structure in more complex ways. More examples can be found in [42].

```
int g;
void p() {
  int i, c;
  i = 0;
  c = 3;
  do {
    g = 2 * i;
    q();
    i = i + c;
  } while (i < 24);
}
```

Figure 3-1: Example Program Fragment

We use an intermediate representation based on control flow graphs. Figure 3-2 shows the graph for the example procedure. The graph contains several nodes, each with a unique label. Most of the nodes have syntax and semantics as in C. For example, the node with label 3 assigns the value of the expression 2*i to variable g. Node 4 is a procedure call node. Control flows from this node to the beginning of the called procedure q. When (and if) the called procedure returns, the execution continues from the next node. Node 6 is a conditional branch node. If the value of variable i is less than 24, the control flows to node 3; otherwise, the control flows to the procedure return node 7.

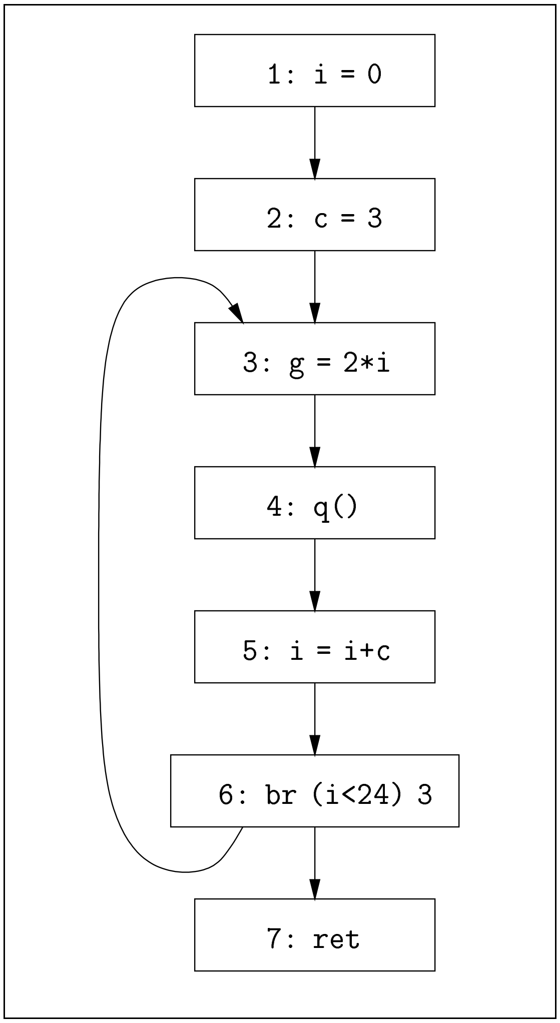

Figure 3-2: Original Procedure

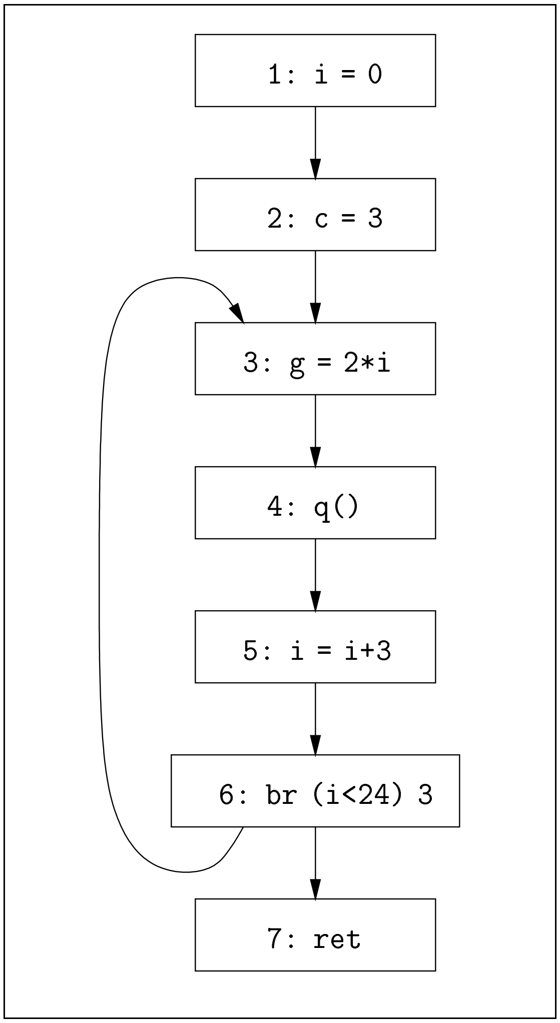

Figure 3-3: Procedure After Constant Propagation

Figure 3-3 shows the graph after constant propagation. To perform such an optimization, the compiler first analyzes the code to discover certain properties. In our example, the compiler discovers that the variable c always has value 3 before the node with label 5 executes. The compiler next performs the transformation of the code. In our example, the compiler propagates the definition of variable c at node 2 to the use of the same variable at node 5. In addition to generating the transformed code, a credible compiler also generates standard and simulation invariants. The standard invariants are used to prove that the analysis results are correct, and those results, together with the simulation invariants, are used to prove that the transformation is correct. We next describe the invariants that the compiler may generate in this example, and how VCG uses the invariants to generate VC.

3.1 Compiler Analysis

After performing an analysis, the compiler presents its results in the form of standard invariants. The invariants are assertions about the program state at different program points. In the language that we consider, an invariant is a relationship between variables at different nodes in the control flow graph. The compiler claims the relationship to be true whenever control reaches the corresponding node. The verifier uses the standard verification-condition generator (StdVCG) to generate the standard verification condition (StdVC) for the assertions. If the compiler can prove the StdVC, then the claimed assertions always hold; they are indeed invariants, and thus the results are correct.

Each standard invariant consists of a formula[1] and the label of a node in the control flow graph. The main invariant in our running example is that the value of the variable `c` is always `3` at the node with label `5`. We use the formula $\texttt{c} = 3$ to represent the predicate on `c`, and we denote the whole invariant as $\texttt{5:inv c} = 3$. The compiler usually generates several invariants to represent the analysis results at different nodes. The compiler may also need to generate additional invariants to be able to prove the results. The reason is that the StdVCG requires at least one invariant for each loop in the procedure. Otherwise, the StdVCG cannot generate the StdVC and marks the compiler output as incorrect. In this example, the invariant $\texttt{5:inv c} = 3$ is sufficient for the StdVCG to generate the StdVC. For expository purposes, we consider two invariants: $\texttt{3:inv c} = 3$ and $\texttt{5:inv c} = 3$.

We next describe how the compiler presents the summary results for a procedure in the form of *standard contexts*. The StdVCG uses standard contexts at call sites. A (standard) context for a procedure is a pair consisting of a *standard input context* and a *standard output context*:[2]

- a (standard) input context is a formula that can contain only the global program variables and the procedure parameters; it represents a relationship between the values of these variables that the compiler assumes to hold at the beginning of the procedure;
- a (standard) output context is a formula that can contain only the global program variables; it represents a relationship between the values of these variables that the compiler claims to hold at the end of the procedure.

[1] Formulas are predicates from the logic that we present in detail in Section 4.2. We use a slightly different syntax for the logic formulas than for the program expressions; in particular, we use `typewriter` font for the program syntax entities, such as variables or labels.

[2] In program verification, a standard context is traditionally called a procedure specification and it consists of a procedure precondition and a procedure postcondition. A precondition for a procedure is formally a predicate on the program state at the beginning of the procedure. We call such a predicate a standard input context because a credible compiler proves the correctness of program analysis results, and the predicate represents the (program) context which the compiler assumes at the beginning of the procedure. A postcondition for a procedure is, in general, a relationship between the program states at the beginning and at the end of the procedure. We call such a relationship a standard output context, and we use, in our basic framework, a simplified version in which the relationship is only unary on the state at the end of the procedure.

The compiler proves that the results (invariants and the output context) for a given context are valid assuming that the input context holds. The compiler may generate many contexts for the same procedure. This allows the compiler to prove that the results of context-sensitive interprocedural analyses are correct.

In our example, the analysis of the procedure p does not assume any initial values for variables; the results hold for all input contexts, and the analysis does not generate any result for the output context. Hence, both input and output contexts are simply true; we will use $\texttt{true}^{p_{in}}$ and $\texttt{true}^{p_{out}}$ to point out which formula we refer to. In this example, for the call to procedure q we also take both input and output contexts to be true, in notation $\texttt{true}^{q_{in}}$ and $\texttt{true}^{q_{out}}$. The reason is that the analysis of p does not require any result from q, except that q does not modify the local variable c of p. The semantics of our language guarantees that the callee cannot access the local variables of the caller. Procedure q may still modify the global variable g.

In our example, all of the formulas for the contexts are simply true. In general, the procedure input and output context can be arbitrary formulas that include the global variables and the parameters of the procedure. For instance, the compiler might express that when g is even before the call to procedure q, then g is 0 after the call. The input context would be $\texttt{g}\%2 = 0$ and the output context $\texttt{g} = 0$. The compiler would need to prove that this context indeed holds for procedure q.

3.1.1 Standard Verification-Condition Generator

We next illustrate how the StdVCG uses the standard invariants and contexts to generate the StdVC for our example. The StdVCG symbolically executes the whole procedure in the same direction in which the control flows through the procedure graph. The symbolic execution uses a symbolic state that maps each program variable to an expression representing the (symbolic) value of that variable. The StdVCG propagates the symbolic state from a procedure node to all its immediate successors. The StdVCG splits the execution at branch nodes into two independent paths. The effect of each node is modeled by appropriately changing the symbolic state and/or generating a part of StdVC.

When the symbolic execution reaches an invariant for the first time, the StdVCG generates the part of StdVC that requires the invariant to hold in the current state. The StdVCG then generates the invariant as a hypothesis for proving the rest of the procedure. The execution finishes when it reaches an invariant for the second time, or when it gets to a return node. At return nodes, the StdVCG adds the standard output context to the StdVC. The StdVC is a formula in the logic which we present in Section 4.2. In our logic formulas we distinguish the program variables from the variables introduced in the formulas by quantification. We call the latter variables *logic variables*. (Not related to the logic variables used in logic programming.)

In our running example, the symbolic execution proceeds in the following steps:

- The execution starts from node 1 and a fresh symbolic state. All program variables are mapped to fresh logic variables that symbolically represent the (unknown) values of the program variables at the beginning of the procedure. In this case, for the program variables `i`, `c`, and `g`, we use the logic variables i^1, c^1, and g^1, respectively. We use superscripts to distinguish different logic variables that represent the values of the same program variable at different program points. (The numbers in the superscripts are not related to procedure labels.) These symbolic values represent all possible concrete values of the variables, and therefore the StdVCG generates a StdVC that universally quantifies over these logic variables. The analysis results should hold for all the initial values that satisfy the standard input context. In this example, it is just $\mathtt{true}^{p_{in}}$, so the StdVC starts as: $\forall i^1.\ \forall c^1.\ \forall g^1.\ \mathtt{true}^{p_{in}} \Rightarrow \ldots$, and the rest of the symbolic execution generates the rest of the StdVC. We abbreviate several consecutive universal quantifications to: $\forall i^1, c^1, g^1.\ \mathtt{true}^{p_{in}} \Rightarrow \ldots$.

- The execution of an assignment node does not generate any part of the StdVC; the StdVCG only modifies the symbolic state: node `1` assigns the expression 1 to `i`, and node `2` assigns the expression 3 to `c`.

- The execution next reaches the invariant `3:inv c` $= 3$ for the first time. The StdVCG substitutes the program variables occurring in the formula of the invariant (only `c` in this case) with the symbolic values of those variables in the current state (3 in this case) and generates the substituted formula as part of the StdVC. Intuitively, this part requires that the invariant holds in the base case of the induction. The StdVCG then generates a fresh symbolic state and substitutes the program variables in the invariant with fresh logic variables. The substituted invariant becomes the assumption used in the StdVC for proving the rest of the procedure. This part can be regarded as an inductive step; the StdVCG uses a fresh state because the invariant has to hold for all successive executions that reach this node. Therefore, the StdVC is extended with: $\ldots 3 = 3 \wedge \forall i^2, c^2, g^2.\ c^2 = 3 \Rightarrow \ldots$.

- The execution continues at node `3` which modifies the symbolic state by mapping `g` to $2 * i^2$. The execution next reaches node `4` which is a call node. At call sites, the StdVCG performs an operation similar to the operation for the invariants. The StdVCG uses the current symbolic state to substitute the appropriate expressions for the global variables and the procedure parameters of the callee in the input context of the callee. In our example, this simply generates $\mathtt{true}^{q_{in}}$. Next, the StdVCG generates a new symbolic state, but replacing only the symbolic values of the global program variables with fresh logic variables (`g` becomes g^3); local procedure variables do not change (`i` and `c` remain i^2 and c^2, respectively). The StdVCG then substitutes the program variables occurring in the output context of the callee with fresh logic variables from the new symbolic state. In our example, it again generates just $\mathtt{true}^{q_{out}}$, and thus the execution of this node extends the StdVC with: $\ldots \mathtt{true}^{q_{in}} \wedge \forall g^3.\ \mathtt{true}^{q_{out}} \Rightarrow \ldots$.

- The execution reaches the other invariant `5:inv c` $= 3$, and the StdVCG does the same as for the first invariant. The difference is that the symbolic value of `c` is c^2 (and not 3) at this point. Thus, c^2 appears in the StdVC: $\ldots c^2 = 3 \wedge \forall i^3, c^3, g^4.\ c^3 = 3 \Rightarrow \ldots$.

- The next node is the assignment node `5`, and the symbolic state before this node maps `i` to i^3, and `c` to c^3. After the node executes, `i` gets mapped to the expression $i^3 + c^3$.
- The conditional branch node `6` splits the execution in two branches:
  - For the true branch, the StdVCG adds the branch condition (after appropriate substitutions) as the assumption to the StdVC and the execution continues at node `3`. At this point, the invariant `3:inv c` = 3 is reached again. The substitution of the invariant is performed as before, and it is added to the StdVC. However, the execution of this branch finishes here because the invariant is reached for the second time.
  - For the false branch, the StdVCG adds the negation of the branch condition (after appropriate substitutions) as the assumption to the StdVC and the execution continues at node `7`. This is the return node, so the StdVCG performs the appropriate substitution on the standard output context and adds it to the StdVC. The execution finishes at the return node.

Finally, the whole StdVC for this example of a standard context is:

$$\begin{array}{l}
\forall i^1, c^1, g^1.\ \texttt{true}^{p_{in}} \Rightarrow \\
\quad 3 = 3 \wedge \forall i^2, c^2, g^2.\ c^2 = 3 \Rightarrow \\
\quad\quad \texttt{true}^{q_{in}} \wedge \forall g^3.\ \texttt{true}^{q_{out}} \Rightarrow \\
\quad\quad\quad c^2 = 3 \wedge \forall i^3, c^3, g^4.\ c^3 = 3 \Rightarrow \\
\quad\quad\quad\quad (i^3 + c^3 < 24 \Rightarrow c^3 = 3) \wedge \\
\quad\quad\quad\quad (\neg(i^3 + c^3 < 24) \Rightarrow \texttt{true}^{p_{out}}).
\end{array}$$

The compiler has to prove that this StdVC holds to show that the analysis results are correct. The compiler generates a proof using the proof rules for the logic presented in Section 4.2.

3.2 Compiler Transformation

In this section we describe the simulation invariants that the compiler generates to prove that the transformed procedure `p` simulates the original procedure with the same name `p`. To avoid repeating transformed and original, we use subscript 1 for the entities from the transformed procedure (program), and subscript 2 for the entities from the original procedure (program). Therefore, we describe the simulation invariants for proving that $\texttt{p}_1$ simulates $\texttt{p}_2$. The compiler generates the simulation invariants together with the transformed program and the standard invariants.[3]

[3] Note, however, that the reasons for generating the standard invariants differ slightly from the reasons for generating the simulation invariants. Namely, the standard invariants both represent the

The simulation invariants represent a correspondence between the states of the two programs at particular program points. Each simulation invariant consists of a formula that represents a relationship between the variables from the two programs and two labels of nodes in procedures $\mathtt{p}_1$ and $\mathtt{p}_2$. For example, the compiler might express that variable $\mathtt{g}_1$ at node $\mathtt{3}_1$ has the same value as variable $\mathtt{g}_2$ at node $\mathtt{3}_2$. We denote such relationship as $\mathtt{3}_1\mathtt{,3}_2\mathtt{:sim\text{-}inv}\ \mathtt{g}_1 = \mathtt{g}_2$. Simplified, this simulation invariant holds if *for all* executions of $\mathtt{p}_1$ that reach $\mathtt{3}_1$, there *exists* an execution of $\mathtt{p}_2$ that reaches $\mathtt{3}_2$ such that $\mathtt{g}_1 = \mathtt{g}_2$ holds. We define precisely when a set of simulation invariants hold for some procedures later in the text.

In our example, we use the simulation invariant that additionally claims that $\mathtt{i}_1$ has the same value as $\mathtt{i}_2$: $\mathtt{3}_1\mathtt{,3}_2\mathtt{:sim\text{-}inv}\ \mathtt{g}_1 = \mathtt{g}_2 \wedge \mathtt{i}_1 = \mathtt{i}_2$. The compiler usually needs to generate a set of simulation invariants. This is analogous to the standard invariants, where the StdVCG executes one procedure to generate the StdVC; for the simulation invariants, the SimVCG executes both procedures to generate the SimVC. There should be enough simulation invariants for the SimVCG to perform the executions. Otherwise, the SimVCG marks the compiler output as incorrect. In our running example, the simulation invariant $\mathtt{3}_1\mathtt{,3}_2\mathtt{:sim\text{-}inv}\ \mathtt{g}_1 = \mathtt{g}_2 \wedge \mathtt{i}_1 = \mathtt{i}_2$ is sufficient[4] for generating a SimVC and we will use only that one invariant.

We next describe how the compiler presents the summary results for the simulation of two procedures in the form of *simulation contexts*. As each standard context is a pair of a standard input context and a standard output context, each *simulation context* is a pair of a *simulation input context* and a *simulation output context*:

- a simulation input context represents a relationship that the compiler assumes to hold between the states of the two programs at the beginning of the two procedures;

- a simulation output context represents a relationship that the compiler claims to hold between the states of the two programs at the end of the two procedures.

As for the standard contexts and standard invariants, the compiler proves that the simulation invariants hold for a particular simulation context.

The compiler is free to choose arbitrary simulation contexts for any pair of procedures within two programs as long as it can prove that those contexts hold. The only requirement is that the compiler has to prove that the transformed program simulates the original program by proving that the starting procedure of the transformed program simulates the starting procedure of the original program for the simulation context which consists of:

- the simulation input context that states that the global variables of the two programs and the parameters of the respective starting procedures have the same values, and

(analysis) results and are used by the StdVCG to generate the StdVC to prove those results. On the other hand, the simulation invariants are not a result by themselves, but a means for generating the SimVC to prove simulation relationships.

[4] Actually, it is possible to generate a provable SimVC using only $\mathtt{3}_1\mathtt{,3}_2\mathtt{:sim\text{-}inv}\ \mathtt{i}_1 = \mathtt{i}_2$.

- the simulation output context that states that the global variables of the two programs have the same values.

Intuitively, this way the compiler proves that the two programs generate the same output, given that they have the same input.

In our example, suppose that p is the starting procedure of the two programs and g is the only global variable in the two programs. Then, the compiler has to prove the simulation context for procedures p_1 and p_2 with input formula $g_1 = g_2$ and output formula $g_1 = g_2$. Additionally, procedure p calls procedure q, or, in general, p_1 calls q_1, and p_2 calls q_2. The compiler needs to use some simulation context to represent the effect of those calls on the global variables g_1 and g_2.

We assume that for the pair q_1 and q_2, the compiler also uses the simulation context with both input and output contexts being $g_1 = g_2$.[5] The compiler could come up with this context in several ways. For example, procedures q_1 and q_2 can be identical procedures, i.e., procedure q from the program where the compiler has optimized p, if the compiler has not transformed q. Alternatively, q_2 can be generated by the compiler by transforming q_1. In this case the compiler would need to prove that the simulation context it claims for q_1 and q_2 indeed holds. Finally, calls to q_1 and q_2 might be calls to the same library procedure.

3.2.1 Simulation Verification-Condition Generator

We next illustrate how the SimVCG uses the simulation invariants and contexts to generate the SimVC in our example. The SimVCG concurrently executes both procedures p_1 and p_2. The executions are symbolic and similar to the symbolic execution that the StdVCG performs. For each procedure, the SimVCG uses a symbolic state that maps variables from that procedure to symbolic expressions representing the values of the variables. The SimVCG propagates the symbolic states through the nodes of the respective procedures and appropriately changes the symbolic states and/or generates a part of SimVC. The SimVCG executes the nodes from p_1 and p_2 independently, except for pairs of nodes that are related, such as call nodes, return nodes, or simulation invariants. The SimVCG needs to *simultaneously* execute the related nodes. Also, the SimVCG needs to interleave the executions of other nodes from p_1 and p_2.

We allow the compiler to specify an arbitrary interleaving of the executions of p_1 and p_2. An interleaving is described with a sequence of *actions* that the compiler generates in addition to a set of simulation invariants. The SimVCG starts the executions of p_1 and p_2 from the beginning nodes and then consecutively uses the actions from the sequence to determine which node(s) to execute next. For example, the action $\texttt{execute}_1$ instructs the SimVCG to execute a node from p_1. We next explain the action $\texttt{execute}_2$ B, where B is a boolean value; this action is used for conditional branch nodes of p_2. We discuss other actions in detail later in the text.

[5] The simulation contexts that the compiler generates need not be only the equality of the two program states. For instance, even in this example, the compiler can generate the simulation input context just true and prove that the simulation output context $g_1 = g_2$ holds.

While executing the procedures, the SimVCG encounters the branch nodes. There are two possible paths from a branch node. For procedure $\mathtt{p}_1$, there is an implicit universal quantification over the control paths—each simulation invariant must hold *for all* paths in $\mathtt{p}_1$ that lead to that simulation invariant. The SimVCG therefore splits the execution at branch nodes of $\mathtt{p}_1$ into two independent executions. For procedure $\mathtt{p}_2$, there is an implicit existential quantification over the control paths—for each simulation invariant in $\mathtt{p}_1$, there *exists* a path in $\mathtt{p}_2$ that leads to a corresponding simulation invariant. The SimVCG therefore follows only one path after a branch node of $\mathtt{p}_2$. (the SimVCG could follow both paths, but that would make the SimVC unnecessarily long.) The SimVCG does not try to determine by itself which path to follow. Instead, the compiler needs to generate the action $\mathtt{execute}_2\ B$, which instructs the SimVCG to execute the branch node and to follow the branch B (taken or not-taken).

In this example, we assume that the action sequence is such that the SimVCG interleaves the executions of $\mathtt{p}_1$ and $\mathtt{p}_2$ using the following strategy:

- first, execute procedure $\mathtt{p}_1$ until it gets to one of the simulation invariants, procedure call nodes, or return nodes;

- then, execute procedure $\mathtt{p}_2$ until it gets to one of the simulation invariants, procedure call nodes, or return nodes;

- finally, execute simultaneously the related nodes from $\mathtt{p}_1$ and $\mathtt{p}_2$ and continue the execution again from $\mathtt{p}_1$ unless the executions finish.

The SimVCG finishes the executions when a simulation invariant is reached for the second time, or when both procedures get to return nodes. At return nodes, the SimVCG also adds the simulation output context to the SimVC. When both executions reach a simulation invariant for the first time, the SimVCG generates the part of SimVC that requires the invariant to hold in the current states. The SimVCG then generates the invariant as a hypothesis and continues executing the rest of the procedures. When both executions reach a call site, the SimVCG uses the simulation context of the callees to generate a part of SimVC and then continues the executions. Note that the actions determine only *when*, and not how, the SimVCG executes particular nodes.

In our running example, the symbolic executions proceed in the following steps:

- The executions start from nodes $\mathtt{1}_1$ and $\mathtt{1}_2$ with fresh symbolic states for both procedures. All program variables are mapped to fresh logic variables that symbolically represent the values of the program variables at the beginning of the procedure. In this case we use the logic variables i_1^1, c_1^1, and g_1^1 for the program variables $\mathtt{i}_1$, $\mathtt{c}_1$, and $\mathtt{g}_1$, and i_2^1, c_2^1, and g_2^1 for the program variables $\mathtt{i}_2$, $\mathtt{c}_2$, and $\mathtt{g}_2$. These symbolic values represent all possible concrete values of the variables. The SimVCG generates a SimVC that universally quantifies over the logic variables representing all program variables from program 1. However, for the variables from program 2, the SimVCG universally quantifies only the logic variables representing values of global variables in program 2 and procedure parameters of $\mathtt{p}_2$, but the SimVCG only existentially quantifies the logic variables representing local variables of $\mathtt{p}_2$ which are not parameters of $\mathtt{p}_2$. We explain later why the SimVCG existentially quantifies the values of local variables of $\mathtt{p}_2$.

The SimVCG starts generating the SimVC for a simulation context by substituting the logic variables for the appropriate program variables in the simulation input context of that context. In this example, the simulation input context is $\mathtt{g}_1 = \mathtt{g}_2$ and the substitution gives $g_1^1 = g_2^1$. Thus, the SimVC starts as: $\forall g_1^1, i_1^1, c_1^1, g_2^1.\ \exists i_2^1, c_2^1.\ g_1^1 = g_2^1 \Rightarrow \ldots$, and the rest of the symbolic executions generate the rest of the SimVC.

- The SimVCG first executes nodes $\mathtt{1}_1$ and $\mathtt{2}_1$ from procedure $\mathtt{p}_1$, modifying its symbolic state. Node $\mathtt{1}_1$ assigns the expression 1 to $\mathtt{i}_1$, and node $\mathtt{2}_1$ assigns the expression 3 to $\mathtt{c}_1$. The execution of this path reaches a simulation invariant. The SimVCG next executes $\mathtt{p}_2$, and nodes $\mathtt{1}_2$ and $\mathtt{2}_2$ modify the symbolic state of $\mathtt{p}_2$.

- The executions reach the simulation invariant $\mathtt{3}_1\mathtt{,3}_2\mathtt{:sim\text{-}inv}\ \mathtt{g}_1 = \mathtt{g}_2 \wedge \mathtt{i}_1 = \mathtt{i}_2$ for the first time. The SimVCG substitutes the program variables occurring in the formula of the invariant ($\mathtt{g}_1$, $\mathtt{g}_2$, $\mathtt{i}_1$, and $\mathtt{i}_2$ in this case) with the symbolic values of those variables in their respective states (g_1^1, g_2^1, 0, and 0 in this case) and generates the substituted formula as the part of SimVC. Similar to the part of StdVC, this part intuitively requires that the invariant holds in the base case of induction. The SimVCG then generates a fresh symbolic state and substitutes the program variables in the invariant with fresh logic variables. The substituted invariant becomes the assumption used in the SimVC for proving the rest of the simulation. This part can be regarded as an inductive step; the SimVCG uses a fresh state because the invariant has to hold for all successive executions that reach this node. This means that the SimVCG now universally quantifies over all logic variables in *both* procedures. Thus, the SimVC is extended with: $\ldots g_1^1 = g_2^1 \wedge 0 = 0 \wedge \forall g_1^2, i_1^2, c_1^2, g_2^2, i_2^2, c_2^2.\ g_1^2 = g_2^2 \wedge i_1^2 = i_2^2 \Rightarrow \ldots$.

- The compiler can use the analysis results to prove the transformation correct. In this example, the compiler specifies with the action $\mathtt{use\text{-}analysis}_2$ that it uses the standard invariant `3:inv c = 3` from $\mathtt{p}_2$.[6] The SimVCG adds the invariant (after the appropriate substitution) as an assumption to the SimVC and extends it with: $c_2^2 = 3 \Rightarrow \ldots$.

- The execution of $\mathtt{p}_1$ continues at node $\mathtt{3}_1$ which modifies the symbolic state of $\mathtt{p}_1$, and then the execution reaches node $\mathtt{4}_1$ which is a call node. The execution of $\mathtt{p}_2$ continues at node $\mathtt{3}_2$. Both executions are now at call sites, and the SimVCG performs a similar operation as the StdVCG. The SimVCG uses the

[6] The original procedure is $\mathtt{p}_2$ because we are showing that $\mathtt{p}_1$, the transformed procedure, simulates $\mathtt{p}_2$.

current symbolic states to substitute the appropriate expressions for the global variables and the procedure parameters of the callees in the simulation input context of the callees. In our example, this generates $2 * i_1^2 = 2 * i_2^2$. Next, the SimVCG generates for each procedure a new symbolic state, but replacing only the symbolic values of the global program variables with fresh logic variables; local procedure variables do not change. Then, the SimVCG substitutes the program variables occurring in the simulation output context of the callees with the fresh logic variables from the new symbolic states. In our example, $g_1^3 = g_2^3$. The execution of call nodes extends the SimVC with: $\ldots 2 * i_1^2 = 2 * i_2^2 \wedge \forall g_1^3, g_2^3.\ g_1^3 = g_2^3 \Rightarrow \ldots$.

- The next node is the assignment node $\mathtt{5}_1$, and the execution modifies the symbolic state by mapping $\mathtt{i}_1$ to $i_1^2 + 3$. The global execution reaches branch node $\mathtt{6}_1$ and splits into two paths:

  - For the true branch, the SimVCG adds the branch condition (after appropriate substitutions) as the assumption to the SimVC and the control flows to node $\mathtt{3}_1$. At this point, an invariant is reached. The SimVCG next executes node $\mathtt{5}_2$ and reaches branch node $\mathtt{6}_2$. At this point, the SimVCG uses additional information provided by the compiler to decide which branch to take. In this case, the SimVCG also follows the true branch and adds the branch condition (after appropriate substitutions) as the part of SimVC. The difference is that this part is not used as an assumption, but as a consequence. The reason is that for each path in $\mathtt{p}_1$, there should be one appropriate path in $\mathtt{p}_2$, but the compiler has to prove that this appropriate path is indeed taken. At this point, both programs reach, for the second time, the invariant $\mathtt{3}_1,\mathtt{3}_2$`:sim-inv` $\mathtt{g}_1 = \mathtt{g}_2 \wedge \mathtt{i}_1 = \mathtt{i}_2$. The substitution of the invariant is performed as before, and it is added to the SimVC. The executions finish here because the invariant is reached for the second time.
  - For the false branch, the SimVCG adds the negation of the branch condition (after appropriate substitutions) as the assumption to the SimVC and the execution of $\mathtt{p}_1$ continues at node $\mathtt{7}_1$. This is a return node, so the SimVCG continues executing $\mathtt{p}_2$. After the SimVCG executes node $\mathtt{5}_2$, it again reaches branch node $\mathtt{6}_2$. In this case, the SimVCG takes the false branch of that node, and adds the negation of the branch condition (after appropriate substitutions) to the SimVC. At this point, both procedures are at return nodes. The SimVCG performs the appropriate substitution on the simulation output context and adds it to the SimVC. The execution of this path finishes at return nodes.

Finally, the whole SimVC for this example of a simulation context is:

$$
\begin{aligned}
&\forall g_1^1, i_1^1, c_1^1, g_2^1.\ \exists i_2^1, c_2^1.\ g_1^1 = g_2^1 \Rightarrow \\
&\quad g_1^1 = g_2^1 \land 0 = 0 \land \forall g_1^2, i_1^2, c_1^2, g_2^2, i_2^2, c_2^2.\ g_1^2 = g_2^2 \land i_1^2 = i_2^2 \Rightarrow c_2^2 = 3 \Rightarrow \\
&\quad\quad 2 * i_1^2 = 2 * i_2^2 \land \forall g_1^3, g_2^3.\ g_1^3 = g_2^3 \Rightarrow \\
&\quad\quad\quad (i_1^2 + 3 < 24 \Rightarrow i_2^2 + c_2^2 < 24 \land g_1^3 = g_2^3 \land i_1^2 = i_2^2) \land \\
&\quad\quad\quad (\neg(i_1^2 + 3 < 24) \Rightarrow \neg(i_2^2 + c_2^2 < 24) \land g_1^3 = g_2^3).
\end{aligned}
$$

In this example, we have not described all the details of the actual SimVCG (presented in Section 4.4). The actual SimVCG expects the compiler to provide some more additional information and, for this example of a simulation context, the actual SimVCG generates a different, but equivalent, SimVC. We next illustrate some more details and show the actual SimVC.

Additional Information for Simulation Verification-Condition Generator

We next discuss some additional information that the SimVCG requires the compiler to generate and we show how the SimVCG uses that information to generate the SimVC. We present two extensions to the SimVCG presented so far, and we also describe the actions for the SimVCG in more detail.

The first extension to the presented SimVCG regards generating related symbolic states for the two procedures. In the example, we have used the simulation invariant $\mathtt{3}_1,\mathtt{3}_2$`:sim-inv` $\mathtt{g}_1 = \mathtt{g}_2 \land \mathtt{i}_1 = \mathtt{i}_2$ that asserts that the values of $\mathtt{g}_1$ and $\mathtt{i}_1$ are the same as the values of $\mathtt{g}_2$ and $\mathtt{i}_2$, respectively. In general, simulation invariants mostly assert that the variables from one program have the same values as the corresponding variables from the other program. Instead of using two different fresh logic variables, say x_1 and x_2, for those two program variables, the actual SimVCG uses the same logic variable for both program variables in their respective symbolic states. That way the SimVCG does not need to generate $x_1 = x_2 \Rightarrow \ldots$ in the SimVC. Additionally, when substituting the program variables in the invariants with the logic expressions, the SimVCG checks (syntactic) equality of the logic expressions, and does not generate them if they are equal. These changes result in a much shorter SimVC. (We present all the details of related symbolic states in Section 4.4.)

The second extension to the presented SimVCG regards the existentially quantified logic variables representing local variables of $\mathtt{p}_1$. We first explain why the SimVCG does not universally quantify those variables. Recall first that the StdVCG starts the symbolic execution with a fresh symbolic state, which maps all program variables to fresh logic variables, and that the StdVCG universally quantifies all those logic variables in the resulting StdVC. The SimVCG similarly starts the symbolic executions with fresh symbolic states for both procedures. The state for $\mathtt{p}_1$ maps variables from program 1 to fresh logic variables, and the state for $\mathtt{p}_2$ maps variables from program 2 to fresh logic variables. However, the SimVCG does not universally quantify all these variables in the SimVC. The reason is that, in general, the resulting SimVC would not hold and thus would not be provable, although $\mathtt{p}_1$ simulates $\mathtt{p}_2$. The problem is that uninitialized local variables lead to the non-determinism in the following sense:

different executions of a program may generate different results for the same input, where we regard as program input only the values of the global variables (and the procedure parameters) at the start of the execution.[7]

Consider, for instance, two identical procedures without parameters that only set global variable `g` to the value of (uninitialized) local variable `l` and return. We would like to have a framework in which we can prove that one of those procedures simulates the other. (Although the equivalence of two programs is undecidable in general, we want at least to be able to prove that two identical procedures are equivalent, no matter what they do!) If we used a SimVCG that universally quantifies over all logic variables, we would obtain as SimVC (for the context stating that $\mathtt{g}_1$ is the same as $\mathtt{g}_2$ at the end of the procedures) $\forall l_1^1, l_2^1.\ l_1^1 = l_2^1$, which clearly does not hold. Therefore, we require that the SimVC be universally quantified over all logic variables representing possible inputs to the procedures (i.e., global variables and parameters). But, for local variables, we require only that for all possible initial values of the local variables of $\mathtt{p}_1$, there exist some initial values of the local variables of $\mathtt{p}_2$ such that the SimVC holds. In the case of the $\{$`g = l;ret`$\}$ procedures, it gives $\forall l_1^1.\ \exists l_2^1.\ l_1^1 = l_2^1$, which clearly holds.

The actual SimVCG requires the compiler to provide the initial expressions for the local variables of $\mathtt{p}_2$. These expressions are usually just equalities of the local variables of $\mathtt{p}_2$ with the corresponding local variables in $\mathtt{p}_1$. (We present all the details in Section 4.4.) In the initial symbolic state for $\mathtt{p}_2$, the SimVCG then maps the local variables of $\mathtt{p}_2$ to the expressions provided by the compiler. Therefore, the SimVCG does not introduce fresh logic variables for the initial values of the local variables of $\mathtt{p}_2$, and the generated SimVC has no existential quantification. This makes it easier for the proof generator to prove the SimVC.

The actual SimVCG, which uses related symbolic states and logic expressions for the initial values of local variables of $\mathtt{p}_2$, generates, for the previous example of a simulation context, the following SimVC:[8]

$$\begin{array}{l}
\forall g_1^1, i_1^1, c_1^1.\ \mathtt{true} \Rightarrow \\
\quad \mathtt{true} \wedge \forall g^2, i^2, c_1^2, c_2^1.\ \mathtt{true} \Rightarrow c_2^1 = 3 \Rightarrow \\
\quad\quad \mathtt{true} \wedge \forall g^3.\ \mathtt{true} \Rightarrow \\
\quad\quad\quad (i^2 + 3 < 24 \Rightarrow i^2 + c_2^1 < 24 \wedge \mathtt{true}) \wedge \\
\quad\quad\quad (\neg(i^2 + 3 < 24) \Rightarrow \neg(i^2 + c_2^1 < 24) \wedge \mathtt{true}).
\end{array}$$

We next describe the actions for the SimVCG in more detail. We first explain why the compiler generates actions. As mentioned, the compiler uses a sequence of actions to guide the SimVCG in performing the symbolic executions of procedures. This is different from the StdVCG, which has a fixed structure in its symbolic execution.

[7] Since the programs have no non-deterministic constructs, the result of an execution is determined by the state of the (whole) memory in which the program starts the executions. But, considering as program input also the values of the uninitialized local variables would disallow many compiler transformations, e.g., the compiler could not add temporary variables.

[8] In practice, the SimVCG does not even generate "`true`" in "$\mathtt{true} \Rightarrow F$", "$\mathtt{true} \wedge F$", or "$F \wedge \mathtt{true}$".

Namely, the StdVCG starts the execution from the beginning of the procedure, and the execution proceeds sequentially until one of the following is reached: a branch node, a standard invariant, a call node, or a return node. At branch nodes, the execution splits into two paths and each of them follows the same algorithm. When a standard invariant is reached, depending on whether it is the first time it is reached or not, the execution either proceeds from the next node or finishes. When a call node is reached, a standard context is used for generating a part of StdVC, and the execution always continues from the next node. Finally, the execution always finishes at a return node.

The SimVCG differs from the StdVCG in that the SimVCG executes two procedures, and has to interleave the symbolic executions of the nodes from those procedures. It is possible to use a fixed structure for this interleaving; in particular, the SimVCG could follow the general strategy that we described in the example: first execute $\mathtt{p}_1$ until a "switch" node, then execute $\mathtt{p}_2$ until the corresponding node, and then execute two nodes simultaneously. However, there are pairs of procedures for which following this fixed structure would generate unnecessarily long SimVCs. We therefore allow the compiler to describe an arbitrary interleaving. Note that it is not necessary that the compiler itself generate all the steps describing the interleaving. The compiler can generate only the set of simulation invariants and potentially some guidelines for the interleaving, and a different module, following those guidelines, can generate the full sequence of the interleaving steps.

We next present the actions that the SimVCG performs while generating the SimVC in our example of a simulation context. We represent the actions in the following way: $\mathtt{ex}_1$ and $\mathtt{ex}_2$ instruct the SimVCG to execute nodes from $\mathtt{p}_1$and $\mathtt{p}_2$, respectively; $\texttt{ex-b}$ instructs the SimVCG to execute nodes from both procedures; and $\mathtt{an}_2$ instructs the SimVCG to include the results of the analysis of $\mathtt{p}_2$. The full sequence of actions is:

$$\mathtt{ex}_1, \mathtt{ex}_1, \mathtt{ex}_2, \mathtt{ex}_2, \texttt{ex-b}, \mathtt{an}_2, \mathtt{ex}_1, \mathtt{ex}_2, \texttt{ex-b}, \mathtt{ex}_1, \mathtt{ex}_1, \mathtt{ex}_2, \mathtt{ex}_2\ T, \texttt{ex-b}, \mathtt{ex}_2, \mathtt{ex}_2\ F, \texttt{ex-b}.$$

Recall that the SimVCG splits the execution of $\mathtt{p}_1$ at branch nodes. In this example, the last $\mathtt{ex}_1$ action in the sequence instructs the SimVCG to execute a branch node. The next three actions in the sequence—$\mathtt{ex}_2, \mathtt{ex}_2\ T, \texttt{ex-b}$—correspond to one path of the execution, and the last three actions—$\mathtt{ex}_2, \mathtt{ex}_2\ F, \texttt{ex-b}$—correspond to another path. We can therefore represent the sequence of actions as an *action tree*:

$$\mathtt{ex}_1, \mathtt{ex}_1, \mathtt{ex}_2, \mathtt{ex}_2, \texttt{ex-b}, \mathtt{an}_2, \mathtt{ex}_1, \mathtt{ex}_2, \texttt{ex-b}, \mathtt{ex}_1, \mathtt{ex}_1 \left< \begin{array}{l} \mathtt{ex}_2, \mathtt{ex}_2\ F, \texttt{ex-b} \\ \mathtt{ex}_2, \mathtt{ex}_2\ T, \texttt{ex-b} \end{array} \right.$$

The action tree describes the interleaving of the symbolic executions of $\mathtt{p}_1$ and $\mathtt{p}_2$. Note that we can make an action tree from a sequence of actions by knowing the flow graphs of the procedures and the placement of simulation invariants. We use actions trees in the presentation of the actual SimVCG in Section 4.4.

Chapter 4

Basic Framework

In this chapter we describe a formal framework for credible compilation. We first define a language for which we build the basic framework. For clarity of presentation, we use a simple language that we call the basic language (BL). In Chapter 6 we describe some extensions to BL and how to handle them in the framework. We also present some limitations of the current credible compilation framework.

We first define the syntax and semantics of BL in Section 4.1. We next define syntax and semantics of the logic formulas for credible compilation in Section 4.2. The logic that we use is a first-order predicate logic with simple integer variables and an extension for referring to program variables within the formulas. The logic formulas are used for two purposes: for representing the (standard and simulation) invariants and for representing the verification conditions.

In Section 4.3 we describe the standard contexts in detail and formally define when the standard contexts hold for some program. We also present how the standard verification-condition generator uses the standard contexts to generate the standard verification condition. In Section 4.4 we describe the simulation contexts in detail and formally define when one BL program simulates another. We also present how the simulation verification-condition generator uses the simulation contexts to generate the simulation verification condition.

The notation that we use for the meta-language mostly follows the notation from the unpublished textbook used in the MIT Programming Languages course [46]. Instead of using the juxtaposition fs to denote the application of a function f to an element s, we use the more traditional notation $f(s)$. We will also use common infix notation for standard binary functions. We explain other abbreviations as we introduce them.

4.1 Basic Language

In this section we first define the syntax of BL and then its formal operational semantics. We make a number of simplifications in designing BL; we present some alternatives later in the text.

BL is a toy imperative language that describes a compiler intermediate representation based on control flow graphs. BL can be regarded as a small subset of the C programming language [26]. However, BL is not a high-level language for writing source programs. We discuss credible compiler translations, in particular translation from a source language to an intermediate representation, in Section 6.3. BL is not a low-level language, either. More specifically, variables have symbolic names, and programs do not operate only with registers and memory as in an assembly language. (In Section 6.1.1 we show how to model registers with variables with special names and memory accesses with pointer accesses.)

4.1.1 BL Syntax

Figure 4-1 shows the abstract syntax of BL. The main syntactic elements are programs, procedures, nodes, and expressions. Each program Q consists of a sequence of declarations of global variables and a sequence of procedures. Each procedure P consists of a sequence of formal parameters, a sequence of declarations, and a sequence of nodes. In the abstract grammar we use x^* to denote a possibly empty sequence and x^+ to denote a sequence with at least one element. In the concrete grammar we use ";" or "," for sequencing. Thus, a more concrete way to describe a procedure is: $P \equiv \texttt{proc}\ I\texttt{(}I_1\texttt{,}\ \ldots\texttt{,}I_n\texttt{)}\ D_1\texttt{;}\ \ldots\texttt{;}D_m\ \{N_1\texttt{;}\ \ldots\texttt{;}N_k\}$. We use "$\equiv$" to denote syntactic equality.

Each node N has a unique label L for identification. There are four groups of nodes, and their informal semantics is the following:

- An assignment node $I\texttt{=}E$ evaluates the expression E, assigns its value to I, and the execution continues at the next node.

- A branch node $\texttt{br(}E\texttt{)}L'$ evaluates the expression E and if it is true, the execution continues at the node with label L'; otherwise, the execution continues at the next node.

- A return node `ret` finishes the execution of the current procedure and the execution continues in the procedure that called this procedure; if there is no such procedure, then the program terminates.

- A call node $I\texttt{(}E_1\texttt{,}\ \ldots\texttt{,}E_n\texttt{)}$ evaluates the expressions E_1 to E_n, passes their values as actual parameters to procedure I, and the execution continues at the first node of procedure I.

Expressions are constructed from variables, integer and boolean constants, and operators. We use expressions without side effects to simplify the presentation. Thus, procedure calls are not expressions, but statements; procedures do not return a result, but they can change the global variables. We consider expressions with side effects in Section 6.1.4. Declarations of BL program variables have no types. We assume that programs operate on integers. We adopt the C convention for boolean values: a non-zero integer represents true, and zero is false. We present some extensions to the language in Section 6.1.

Syntactic Domains :

$Q \in$ Program
$P \in$ Procedure
$D \in$ Declaration
$N \in$ Node
$L \in$ Label
$E \in$ Expression
$I \in$ Identifier
$U \in$ Unary-operator $= \{\texttt{!}, \texttt{-}\}$
$O \in$ Binary-operator $= \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\} \cup \{\texttt{==}, \texttt{!=}, \texttt{>}, \texttt{<}, \texttt{>=}, \texttt{<=}\} \cup \{\texttt{\&\&}, \texttt{||}\}$
$B \in$ Boolean-literal $= \{\texttt{TRUE}, \texttt{FALSE}\}$
$Z \in$ Integer-literal $= \{\ldots, \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots\}$

Production Rules :

$E \equiv Z$	[Integer Literal]
$\mid B$	[Boolean Literal]
$\mid I$	[Variable Reference]
$\mid E_1 \; O \; E_2$	[Binary Operator]
$\mid U \; E_1$	[Unary Operator]
$L \equiv I$	[Textual Label]
$\mid Z$	[Numerical Label]
$N \equiv L\texttt{:}I\texttt{=}E$	[Assignment Node]
$\mid L\texttt{:br(}E\texttt{)}L'$	[Branch Node]
$\mid L\texttt{:ret}$	[Return Node]
$\mid L\texttt{:}I\texttt{(}E^*\texttt{)}$	[Call Node]
$D \equiv I$	[Implicit Size]
$P \equiv \texttt{proc}\; I\texttt{(}I^*\texttt{)}\; D^* \; \{N^+\}$	[Procedure]
$Q \equiv \texttt{prog}\; D^* \; P^+$	[Program]

Figure 4-1: Abstract Syntax of BL

We introduce some additional notation and describe semantic checks for BL programs. We write $P \in Q$ to denote that procedure P is in program Q and $N \in P$ to denote that node N is in procedure P. We write $P(L)$ for the node with label L in procedure P. We require that target labels of all branches in P be labels of some nodes in P. For a node with label L in procedure P, the label of the next node is denoted $L +_P 1$. This is the label of the node following L in the static sequence of nodes, not necessarily in the execution. There is exactly one next node for each node except the last node. We require that the last node be a return node. We write the label of the starting node of procedure P as start-label(P) or P^0.

Each procedure $P \equiv$ `proc` I`(`I^*`)` D^* $\{N^+\}$ has three sets of variables in the scope; we denote these sets as: locals(P) for the local variables of P (variables in D^*), params(P) for the parameters of P (variables I^*), and globals(P) (or globals(Q)) for the global variables of the program Q that contains P. We require that params(P) $\cap$ locals(P) $= \{\}$. We use vars(P) $=$ params(P)$\cup$locals(P)$\cup$globals(P) for the set of all variables in the scope of P. Therefore, locals(P)$\cup$params(P) is a set of all non-global variables of P and vars(P) $-$ locals(P) is a set of all non-local variables of P. We require that all variables used in a procedure/program be declared. Also, we require that procedures in all call nodes be procedure (not variable) identifiers and that the number of actual parameters be the same as the number of formal parameters.

4.1.2 BL Semantics

In this section we present the formal semantics of BL. We use a structured operational semantics which is formally a five-tuple $\langle \mathcal{C}, \rightarrow, \mathcal{F}, \mathcal{I}, \mathcal{O} \rangle$. $\mathcal{C}$ is a set of configurations that represent the state of a machine executing a BL program. $\rightarrow$ is a relation describing transitions between configurations. $\mathcal{F}$ is a set of final configurations in which a program can finish its execution. $\mathcal{I}$ is an input function that maps a program and its input data to an initial configuration. $\mathcal{O}$ is an output function that maps a final configuration to a program's output.

Before we define $\mathcal{C}$, we introduce some additional domains:

$$\begin{aligned}
V &\in \text{Value} = \text{Integer-literal} \\
&\ \text{Address} = \text{Integer-literal} \\
m &\in \text{Memory} = \text{Address} \rightarrow \text{Value} \\
p &\in \text{Alloc-Pointer} = \text{Address} \\
&\ \text{One-Environment} = \text{Identifier} \rightharpoonup \text{Address} \\
a &\in \text{Environment} = \text{One-Environment} \times \text{One-Environment} \\
h &\in \text{History} = (\text{Label} \times \text{Procedure} \times \text{Environment})^*.
\end{aligned}$$

The Value domain represents the values to which BL expressions can evaluate; BL programs operate on integers. The values of program variables are stored in a memory. Memory locations have addresses from the domain Address. The Alloc-Pointer domain represents the values of the (stack) pointer used for memory allocation. The domain Memory contains functions that map addresses to values stored at those addresses. The domain One-Environment contains functions that map variable names to the memory addresses where values of those variables are stored. We use a pair of environments to represent separately the environments for global and local variables. We will use superscripts g and l to refer to the global and local parts; for example, a^g denotes the first component of pair a, and a^l denotes the second component.

Elements of the History domain basically represent (control) stacks. They are sequences of triples. Each triple consists of the information necessary to resume the execution after a procedure call returns: the label of the next node to be executed after the return, the procedure in which that node is, and the environment for that procedure. Data stacks, namely the values of the local variables and procedure parameters, are stored in memory.

We next define configurations of BL operational semantics. They are six-tuples consisting of the label of the node to execute, the current state of the memory, the environment for the current scope, the value of the stack pointer, the history of procedure calls, and the current procedure:

$$\mathcal{C} = \text{Label} \times \text{Memory} \times \text{Environment} \times \text{Alloc-Pointer} \times \text{History} \times \text{Procedure}.$$

To be even more precise, we should include in the configurations the program that is executing. However, the program does not change during the execution, and we omit it. Also, during the execution of a fixed procedure activation, the stack pointer, the history, and the procedure do not change[1] except at call sites, and we abbreviate the configurations to triples $\langle L, m, a\rangle$.

We next explain the stack allocation in BL. The basic function that allocates space for a local variable I is:

$$\text{alloc}^l(\langle m, a, p\rangle, I) = \langle m, \langle a^g, a^l[I \mapsto p]\rangle, p+1\rangle.$$

This function only extends the local environment and increments the stack pointer; the memory remains the same. To initialize the value of the local variable we additionally change the memory location:

$$\text{alloc-init}^l(\langle m, a, p\rangle, I, Z) = \langle m[p \mapsto Z], \langle a^g, a^l[I \mapsto p]\rangle, p+1\rangle.$$

We also define an analogous function for initializing a global variable:

$$\text{alloc-init}^g(\langle m, a, p\rangle, I, Z) = \langle m[p \mapsto Z], \langle a^g[I \mapsto p], a^l\rangle, p+1\rangle.$$

We will use these functions for sequences of variables (and their initial values when appropriate). In particular, function alloc-locals is a map of alloc^l over a sequence I^*:

$$\begin{array}{l}\text{alloc-locals}(\langle m, a, p\rangle, [\,]) = \langle m, a, p\rangle \\ \text{alloc-locals}(\langle m, a, p\rangle, I.I^*) = \text{alloc-locals}(\text{alloc}^l(\langle m, a, p\rangle, I), I^*).\end{array}$$

Similarly, alloc-params$(\langle m, a, p\rangle, I^*, Z^*)$ and alloc-params$(\langle m, a, p\rangle, I^*, Z^*)$ are maps

[1] Environment a also does not change, but it is used for evaluating expressions.

of alloc-init^l and alloc-init^g, respectively.

The stack deallocation in BL does not change the stack pointer; the functions that deallocate space simply return their input p:

$$\begin{aligned}&\text{dealloc-locals}(p, D^*) = p\\&\text{dealloc-params}(p, I^*) = p.\end{aligned}$$

We explain later why we do not decrease the value of p.

We next present how a program starts and finishes its execution. We also present what is input and output data for a program. To start a program execution, we need to give two sequences of values: one for the global variables of the program and one for the actual parameters of the starting procedure. The input function $\mathcal{I}$ maps a program and two sequences to an initial configuration:

```
$\mathcal{I}(Q, Z_g^*, Z_p^*) =$
  $\textbf{matching } Q \rhd \texttt{ prog } D^* \ P^+ \ \rrbracket$
   $\textbf{let } \langle m_s, p_s \rangle \textbf{ be } \text{random-memory}() \textbf{ in}$
    $\textbf{let } a_s \textbf{ be } \text{empty-environment}() \textbf{ in}$
     $\textbf{let } \langle m_g, a_g, p_g \rangle \textbf{ be } \text{alloc-globals}(\langle m_s, a_s, p_s \rangle, D^*, Z_g^*) \textbf{ in}$
      $\textbf{let } P \textbf{ be } \text{head}(P^+) \textbf{ in}$
       $\textbf{matching } P \rhd \texttt{ proc } I\texttt{(}I^*\texttt{)} \ D^* \ \{N^+\} \ \rrbracket$
        $\textbf{let } \langle m_p, a_p, p_p \rangle \textbf{ be } \text{alloc-params}(\langle m_g, a_g, p_g \rangle, I^*, Z_p^*) \textbf{ in}$
         $\textbf{let } \langle m, a, p \rangle \textbf{ be } \text{alloc-locals}(\langle m_g, a_g, p_g \rangle, D^*) \textbf{ in}$
          $\langle P^0, m, a, p, [\,], P \rangle$
       $\textbf{endmatching}$
  $\textbf{endmatching}.$
```

The initial state of the memory, before the execution starts, is completely arbitrary, and the environment is empty. The function alloc-globals first creates an environment for the global variables and initializes the memory locations for those variables. Next, the starting procedure P of the program is obtained. Space for the parameters of P is allocated in memory, and those locations are initialized to the input values. Space is also allocated for the local variables of P, but those locations are not initialized. Execution starts with the first node in P and an empty history.

The program finishes its execution when the starting procedure gets to the return node. At that point, the history is empty. The output of the program is only the sequence of values of its global variables, not the state of the whole memory:

$$\begin{aligned}&\mathcal{F} = \{\langle L, m, a, p, [\,], P \rangle | P(L) \equiv L\texttt{:ret}\}\\&\mathcal{O}(\langle L, m, a, p, [\,], P \rangle, Q) = \textbf{matching } Q \rhd \texttt{ prog } D^* \ P^+ \ \rrbracket\\&\qquad\qquad\qquad\qquad\qquad\quad \text{extract-output}(m, \langle a^g, \text{empty-environment}() \rangle, D^*)\\&\qquad\qquad\qquad\qquad\qquad\textbf{endmatching},\end{aligned}$$

where

$$\begin{aligned}&\text{extract-output}(m, a, [\,]) = [\,]\\&\text{extract-output}(m, a, I.I^*) = m(a(I)).\text{extract-output}(m, a, I^*).\end{aligned}$$

The value of variable I in memory m and environment a is denoted $m(a(I))$. We use $a(I)$ to denote the address of the variable I in the environment a: if I is in the domain of a^l, then $a(I) = a^l(I)$; otherwise, $a(I) = a^g(I)$.

We use notation $m(a(E))$ to denote the value of expression E in memory m and environment a. Figure 4-2 defines the value of BL expressions. The definition uses the helper function calc-value. This function takes a syntactic representation of the operator and operands, and returns an integer literal which is the result of applying that operator to those operands. The operations give the same results as in C, except for division and modulo operations. We define that they evaluate to 0 if their second operand is 0. This way expression evaluation cannot end up in an error state. We discuss the absence of error states after presenting completely the semantics of BL programs.

$$
\begin{array}{l}
m(a(Z)) = Z \\
m(a(\texttt{TRUE})) = \texttt{1} \\
m(a(\texttt{FALSE})) = \texttt{0} \\
m(a(E_1\ O\ E_2)) = \text{calc-value}(O, m(a(E_1)), m(a(E_2))) \\
m(a(U\ E)) = \text{calc-value}(U, m(a(E)))
\end{array}
$$

Figure 4-2: BL Expression Evaluation

Figure 4-3 defines the rewrite rules for BL. We present a high-level operational semantics [4], without specifying the details of expression evaluation[2] on a machine that is executing the programs. Instead, in each step the machine evaluates all the expressions occurring in the current node and makes a transition to the next node.

The rule for I`=`E evaluates the expression, updates the memory, and the execution continues at the next node. The rules for `br(`E`)`L' evaluate the condition, and depending on its truth value, the execution continues at the next node or the node with label L'. The execution of I`(`E^*`)` first evaluates the values of parameters. Next, it allocates space in memory for the parameters and initializes them with the values. Finally, it allocates space for local variables of I and the execution continues from the first node of I. The rule for `ret` deallocates the space for local variables and parameters, and the execution continues in the caller, at the node after the call.

We now discuss the stack deallocation. We use deallocate functions that do not change the stack pointer p; they simply return p' that has the same value as p. This may look surprising, since the expected behavior would be that the deallocate functions decrease p (opposite of the allocate functions). We do not change p because we model local variables that have arbitrary initial values for *every* call. If we decreased p, the initial values of local variables would be the final values of local variables in previous calls. We want to eliminate such dependencies in the language for two reasons: they do not formalize properly the intuitive notion of uninitialized local variables and they make compiler transformations more difficult.

[2] This is easy to do because expressions have no side effects.

$$\begin{gathered}
\langle L, m, a, p, h, P\rangle \rightarrow \langle L +_P 1, m[V \mapsto V'], a, p, h, P\rangle \qquad [\mathit{assign}]\\
\text{where } P(L) \equiv L\,\texttt{:}\,I\texttt{=}E\\
\text{and } V \equiv a(I) \text{ and } V' \equiv m(a(E))
\end{gathered}$$

$$\begin{gathered}
\langle L, m, a, p, h, P\rangle \rightarrow \langle L', m, a, p, h, P\rangle \qquad [\mathit{branch\text{-}true}]\\
\text{where } P(L) \equiv L\,\texttt{:}\,\texttt{br(}E\texttt{)}\,L'\\
\text{and } m(a(E)) \not\equiv \texttt{0}
\end{gathered}$$

$$\begin{gathered}
\langle L, m, a, p, h, P\rangle \rightarrow \langle L +_P 1, m, a, p, h, P\rangle \qquad [\mathit{branch\text{-}false}]\\
\text{where } P(L) \equiv L\,\texttt{:}\,\texttt{br(}E\texttt{)}\,L'\\
\text{and } m(a(E)) \equiv \texttt{0}
\end{gathered}$$

$$\begin{gathered}
\langle L, m, a, p, h, P\rangle \rightarrow \langle \text{start-label}(P'), m', a', p', \langle L +_P 1, P, a\rangle.h, P'\rangle \qquad [\mathit{call}]\\
\text{where } P(L) \equiv L\,\texttt{:}\,I\texttt{(}E^*\texttt{)}\\
\text{and } P' \equiv \texttt{proc}\ I\texttt{(}I^*\texttt{)}\ D^*\ \{N^+\}\\
\text{and } V^* \equiv m(a(E^*))\\
\text{and } \langle m', a', p'\rangle = \text{alloc-locals}(\text{alloc-params}(\langle m, a, p\rangle, I^*, V^*), D^*)
\end{gathered}$$

$$\begin{gathered}
\langle L, m, a, p, \langle L', P', a'\rangle.h, P\rangle \rightarrow \langle L', m, a', p', h, P'\rangle \qquad [\mathit{return}]\\
\text{where } P(L) \equiv L\,\texttt{:}\,\texttt{ret}\\
\text{and } P \equiv \texttt{proc}\ I\texttt{(}I^*\texttt{)}\ D^*\ \{N^+\}\\
\text{and } p' = \text{dealloc-params}(\text{dealloc-locals}(p, D^*), I^*)
\end{gathered}$$

Figure 4-3: BL Operational Semantics Rewrite Rules

We allow programs that read uninitialized local variables.[3] However, such a program can generate different results in different executions, depending on the state of the memory at the beginning of the execution. As we pointed out, the execution starts with a completely arbitrary memory. That is why we call BL programs *almost* deterministic—the output is determined uniquely by the input *and* the initial state of the memory, but that state is arbitrary. For example, consider a simple program that has only one global variable and only one parameterless procedure. Further, let the procedure set the global variable to an unknown value of a local variable. This program can generate any output, no matter what the value of the global variable at the beginning is.

We next argue that not decreasing the stack pointer p in the deallocate functions does not affect "well-behaved" programs. We call a program "well-behaved" if it does not read uninitialized variables, and its output is therefore determined solely by its input. Consider the execution of BL programs on a realistic machine that would decrease p on a return from a callee. The execution of a BL program on such machine would generate one of the results that the executions of the same program can generate on a machine that does not decrease p. If a program can generate only one result (for a given input) on a machine that does not decrease p, then the program generates the same result on the machine that decreases p. Therefore, not decreasing p does not affect the result of the "well-behaved" programs as they can generate only one result.

For the original programs that are not "well-behaved" and read uninitialized variables, we could define the result to be "undefined." We could then allow the compiler to generate any transformed program; however, we do not do that. We require, instead, the compiler to generate a transformed program that can generate only the results that the original program can generate. We present details later in the text.

Finally, we point out that BL semantics has no error states. There are no stuck configurations: each configuration is either final and the execution finishes, or the execution can make a transition from the configuration to its successor. BL programs thus either generate a regular output or do not terminate. This is a simplification that we make in the basic framework. We consider extending the semantics with error states in Section 6.1.3.

Partial Executions

We next define partial executions of BL programs and procedures using the rewrite rules for BL. We also define quantification of partial executions, which we use to specify the compiler requirements.

Definition 1 *A partial execution of a program Q is a sequence of configurations $\langle P^0, m, a, p, [\,], P\rangle \to \ldots \to \langle L, m', a', p', h, P'\rangle$ such that:*

- *the first configuration is the initial configuration for an execution of Q: the current procedure P is the starting procedure of Q, the current label P^0 is the label*

[3] It is undecidable in general to check if a BL program reads an uninitialized local variable.

of the starting node of P, the history is empty, and m, a, and p are, respectively, the memory, the environment, and the stack pointer at the beginning of P; and

- *each successor configuration is obtained from the preceding configuration by a rewrite rule.*

The configurations in a partial execution of a program can have different current procedures. The current procedure changes between two consecutive configurations when a call node or a return node is executed. More precisely, a procedure can call itself, and it is the *activation* of the procedure that changes. Each configuration in a partial execution of a program belongs to some procedure activation. All configurations that belong to a fixed procedure activation form a sequence of configurations, which is a subsequence (not necessarily consecutive) of the sequence of configurations for the partial execution of the program. We usually refer to a sequence of configurations that belong to a fixed procedure activation as a partial execution of a procedure.

Definition 2 *A partial execution of (an activation) of a procedure P is a sequence of configurations $\langle P^0, m^0, a, p^0, h, P\rangle \overset{+}{\rightarrow} \ldots \overset{+}{\rightarrow} \langle L, m, a, p, h, P\rangle$ such that:*

- *the first configuration consists of the label of the starting node of P, the starting memory m^0 at the entry of the procedure, the environment a for the particular procedure activation, and the stack pointer p^0 and the history h at the entry of the procedure; and*
- *each successor configuration is obtained from the preceding configuration:*
  - *if the preceding configuration is not a call node, the successor is obtained by one rewrite rule, and*
  - *if the preceding configuration is a call node, the successor is obtained by several rewrite rules, none of which is a return from a configuration with history h; and*
- *all configurations have the same environment a, history h, and procedure P.*

We usually refer only to the first and the last configuration in a sequence representing a partial execution. Therefore, we denote a partial execution as $\langle P^0, m^0, a, p^0, h, P\rangle \overset{+}{\dashrightarrow} \langle L, m, a, p, h, P\rangle$. We next describe another abbreviation that we use.

Procedure P can call other procedures during an execution. The environment, a history, and current procedure temporarily change at a call site, but are restored after the called procedure returns. (Therefore, h and P are the same for all configurations in a partial execution of an activation, but h and P can be the same even for different activations.) On the other hand, the memory is not restored, and for BL programs, the stack pointer is also not restored: p is increased at call sites, but not decreased on returns. The expression evaluation, however, does not depend on the stack pointer. Therefore, during the execution of a fixed procedure activation, we abbreviate the configurations $\langle L, m, a, p, h, P\rangle$ to triples $\langle L, m, a\rangle$, and we represent *a partial execution of an activation of a procedure* P as $\langle P^0, m^0, a\rangle \overset{+}{\dashrightarrow} \langle L, m, a\rangle$.

We next explain quantification of partial executions. We use the terms "for all partial executions" and "there exists a partial execution" to specify the correctness requirements for compiler analyses (Section 4.3.2) and compiler transformations (Section 4.4.2). We first introduce quantification of starting configurations $\langle P^0, m^0, a, p^0, h, P\rangle$ for an activation of a procedure $P \equiv \texttt{proc}\ I\texttt{(}I^*\texttt{)}\ D^*\ \{N^+\}$. Any p^0 is possible at the beginning of any P. Let $p' = p^0 - |\text{locals}(P)|$ and $p'' = p' - |\text{params}(P)|$, i.e., p' is the value of the stack pointer before the allocation of the local variables of P and p'' is the value of the stack pointer before the allocation of the parameters of P. We say that environment a and history h are *possible* for p^0 and P if:

- local environment a^l maps I^* to addresses from p'' to $p' - 1$ and a^l maps D^* to addresses from p' to $p^0 - 1$, and
- global environment a^g is the same for all environments in $h = \langle L', P', a'\rangle^*$ and a^g maps the global variables of the program that P is in to addresses less than some p^g, and
- if $h = \langle L', P', a'\rangle.h'$, then there exists some $p'^0 \leq p''$ such that a' and h' are possible for p'^0 and P'; otherwise if $h = [\,]$, then $p^g \leq p''$.

Basically, an environment and a history are possible for a procedure if they represent a possible stack state.

We define the quantification of starting configurations based on memory m^0:

- "for all starting configurations of P" means: for all p^0, and for all a and h possible for that p^0, and for all m^0; and
- "there exists a starting configuration of P" means: for all p^0, and for all a and h possible for that p^0, and for all values of m^0 locations with addresses less than p' (the locations "below" the local variables of P), there exist some values of other m^0 locations (the locations for the local variables of P and "above" the local variables).

Finally, we define the quantification of partial executions:

- "for all partial executions $\langle P^0, m^0, a\rangle \overset{+}{\dashrightarrow} \langle L, m, a\rangle$" means:
  - for all starting configurations of P, and
  - for all sequences of configurations from $\langle P^0, m^0, a\rangle$ to $\langle L, m, a\rangle$ such that each configuration is the successor of the preceding configuration;
- "there exists a partial execution $\langle P^0, m^0, a\rangle \overset{+}{\dashrightarrow} \langle L, m, a\rangle$" means:
  - there exists a starting configuration of P, and
  - there exists a sequence of configurations from $\langle P^0, m^0, a\rangle$ to $\langle L, m, a\rangle$ such that each configuration is the successor of the preceding configuration.

4.2 Basic Logic

This section presents the logic for credible compilation for the basic language (BL) introduced in the previous section. We use first-order predicate logic with simple integer variables [4]. In logic formulas, there needs to be a way to refer to the variables from both the original program and the compiler transformed program. To this end, we add special constructors to logic expressions. We describe in detail the relationship between the program and logic expressions. We finally define semantics of the logic formulas, and discuss briefly proof rules for those formulas.

4.2.1 Syntax of Logic Formulas

Figure 4-4 shows the syntax of the logic formulas.[4] We use the meta-variable F for the elements of the syntactic domain of the logic formulas. The formulas consists of boolean expressions G^b and logical operators connecting them. The formulas can also be universally or existentially quantified; in the basic logic, only integer logic variables can be quantified.

Boolean expressions are constructed from boolean constants, operators that take either boolean or integer expressions and produce boolean expressions, and the integer to boolean conversion function. $\texttt{i2b(}G^i\texttt{)}$ could be also written as $G^i \neq 0$. Integer expressions are constructed from integer constants, integer variables, operators that produce integer results, and the boolean to integer conversion function. $\texttt{b2i(}G^b\texttt{)}$ represents the function that takes a boolean value, and if it is true, returns 1, otherwise 0.

There are two groups of variables in the logic formulas: logic variables, for which we use the meta-variable x, and program variables, for which we use the meta-variable I. We use the integer expression constructors $H\texttt{(}I\texttt{)}$ to denote values of program variables. We introduced these special constructors for two reasons.

First, $H\texttt{(}I\texttt{)}$ constructors provide a way to refer, within a logic formula, to the program variables not visible in the lexical scope of the part of the program enclosing the formula. For example, consider a program Q that has a global variable named `v` and a procedure P that itself has a local variable named `v`. Referring to `v` within P accesses the local variable; there is no way to access the global `v`. In the formulas in P, the compiler might also need to refer to the global `v`. That is the intended meaning of the constructors `glob` (to denote the global variables) and `loc` (to denote the local variables).[5] (We also use `loc` to denote the procedure parameters. Unless noted otherwise, we always treat the procedure parameters in the same way as the local variables.) For example, the formula $\texttt{loc(v)} = \texttt{glob(v)}$ denotes that the local `v` has the same value as the global `v`; depending on the values of the two different variables (with the same name), the formula can be true or false.

[4] We use different notation for logic expressions (common mathematical symbols in a proportional font) than for program expressions (common programming languages symbols in a fixed-width font).

[5] Another way to make a distinction between the variables would be to require disjoint sets of names for the global and local variables.

Syntactic Domains :

$F \in$ Formula
$G^b \in$ Boolean-expression
$G^i \in$ Integer-expression
$H \in$ Program-variable $= \{\texttt{loc}, \texttt{glob}\} \cup \{\texttt{loc}_1, \texttt{glob}_1, \texttt{loc}_2, \texttt{glob}_2\}$
$I \in$ Identifier
$x \in$ Logic-identifier
$O^a \in$ Arithmetic-operator $= \{+, -, *, /, \%\}$
$O^l \in$ Logical-operator $= \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$
$O^r \in$ Relational-operator $= \{=, \neq, >, <, \geq, \leq\}$
$Z \in$ Integer-constant $= \{\ldots, -2, -1, 0, 1, 2, \ldots\}$

Production Rules :

$$
\begin{array}{lll}
G^i & \equiv Z & \text{[Integer Constant]} \\
 & \mid\ x & \text{[Logic Variable]} \\
 & \mid\ H(I) & \text{[Program Variable]} \\
 & \mid\ G^i_1\ O^a\ G^i_2 & \text{[Arithmetic Operator]} \\
 & \mid\ -G^i_1 & \text{[Unary Minus]} \\
 & \mid\ \texttt{b2i}(G^b) & \text{[Boolean Conversion]} \\
 & & \\
G^b & \equiv \texttt{true} & \text{[Constant True]} \\
 & \mid\ \texttt{false} & \text{[Constant False]} \\
 & \mid\ G^b_1\ O^l\ G^b_2 & \text{[Logical Operator]} \\
 & \mid\ \neg G^b_1 & \text{[Negation]} \\
 & \mid\ G^i_1\ O^r\ G^i_2 & \text{[Relational Operator]} \\
 & \mid\ \texttt{i2b}(G^i) & \text{[Integer Conversion]} \\
 & & \\
F & \equiv G^b & \text{[Boolean Expressions]} \\
 & \mid\ F_1\ O^l\ F_2 & \text{[Logical Operator]} \\
 & \mid\ \neg F' & \text{[Negation]} \\
 & \mid\ \forall x.\ F' & \text{[Universal Quantification]} \\
 & \mid\ \exists x.\ F' & \text{[Existential Quantification]}
\end{array}
$$

Figure 4-4: Abstract Syntax of the Logic Formulas

The second reason for introducing $H(I)$ constructors is for the formulas that describe a correspondence between two programs. Such a formula refers to the variables from both programs.[6] We use the constructors with indices to denote the program variables: $\texttt{loc}_1$ and $\texttt{glob}_1$ for the variables from program 1 and $\texttt{loc}_2$ and $\texttt{glob}_2$ for the variables from program 2. Although the syntax for formulas allows combining constructors `loc` and `glob` with $\texttt{loc}_1$, $\texttt{glob}_1$, $\texttt{loc}_2$, and $\texttt{glob}_2$, we use those two groups exclusively. There are two groups of formulas: formulas that describe properties of only one program and formulas that describe correspondences between two programs. The formulas from the first group can contain only the constructors `loc` and `glob` without indices, and the formulas from the second group can contain only the constructors with indices.

The majority of the formulas that describe correspondences between two programs are conjunctions stating that some variables from one procedure/program have the same values as the corresponding variables from the other procedure/program. We introduce a special form for such formulas. We use meta-variable J for the pairs consisting of a logic formula and a sequence of pairs of program variables: $F\texttt{,}(H_1\texttt{(}I_1\texttt{)}\texttt{,}H_2\texttt{(}I_2\texttt{)})^*$. Such a pair represents conjunction $F \wedge \bigwedge H_1\texttt{(}I_1\texttt{)} = H_2\texttt{(}I_2\texttt{)}$, where $\bigwedge$ ranges over the pairs of variables in the sequence. We require that a variable can appear in only one pair of variables, i.e., in the sequence of the first components $H_1\texttt{(}I_1\texttt{)}^*$, all the variables have to be different, and in the sequence of the second components $H_2\texttt{(}I_2\texttt{)}^*$, all the variables have to be different. We write var-pairs(J) for the set of pairs of variables from formula J.

We introduce one notational convention for referring to the entities from two programs Q_1 and Q_2. We use index $_{1/2}$ to mean "1 and 2, respectively." For example, we say "variables $H_{1/2}\texttt{(}I_{1/2}\texttt{)}$ from programs $Q_{1/2}$" to mean "variables $H_1\texttt{(}I_1\texttt{)}$ and $H_2\texttt{(}I_2\texttt{)}$ from program Q_1 and program Q_2, respectively." We also write only "variables $H_1\texttt{(}I_1\texttt{)}$ and $H_2\texttt{(}I_2\texttt{)}$ from programs Q_1 and Q_2" referring to the respective entities, without mentioning it explicitly.

4.2.2 Relationship between Program and Logic Expressions

We next describe the relationship between expressions in BL programs and expressions in the logic formulas. The program and logic expressions have a similar structure. The main difference is in the way of referring to the program variables. In programs, the variables are referred to simply by their name. In logic expressions, program variables are referred to by using expression constructors from H. In logic expressions, there are also logic variables introduced by quantification of logic formulas. These variables are referred to simply by their name. Another difference between program and logic expressions is typing. Whereas program expressions are untyped (all expressions have integer type), logic expressions have types and they can be either integer or boolean.

[6] Again, one way to make a distinction between the variables would be to require that the two programs use different names for the variables. Although this might be acceptable for global and local variables within one program, it is less acceptable for variables from two programs.

We introduce functions for translating expressions from program form to logic form. Figure 4-5 shows the definition of the translation functions. These functions use a symbolic environment e in translation. The symbolic environment e is similar to the environment a used in the operational semantics. However, e does not map variable names to the memory addresses, but to the appropriate logic expressions for those variables. For example, a symbolic environment for one program maps the variables in the following way: for each local variable I_l, $e(I_l) = \mathtt{loc(}I_l\mathtt{)}$, and for each global variable I_g, $e(I_g) = \mathtt{glob(}I_g\mathtt{)}$.

```
$\text{translate}(E, e) = \text{to-type}(\text{translate-type}(E, e), \mathtt{int})$
$\text{translate-bool}(E, e) = \text{to-type}(\text{translate-type}(E, e), \mathtt{bool})$
$\text{translate-seq}(E.E^*, e) = \text{translate}(E, e).\text{translate-seq}(E^*, e)$
$\text{translate-seq}([\,], e) = [\,]$

$\text{translate-type}(Z, e) = \langle Z, \mathtt{int}\rangle$
$\text{translate-type}(\mathtt{TRUE}, e) = \langle \mathtt{true}, \mathtt{bool}\rangle$
$\text{translate-type}(\mathtt{FALSE}, e) = \langle \mathtt{false}, \mathtt{bool}\rangle$
$\text{translate-type}(I, e) = \langle e(I), \mathtt{int}\rangle$
$\text{translate-type}(\mathtt{-}E_1, e) =$
  $\textbf{let } G \textbf{ be } \text{to-type}(\text{translate-type}(E_1, e), \mathtt{int}) \textbf{ in } \langle -G, \mathtt{int}\rangle$
$\text{translate-type}(\mathtt{!}E_1, e) =$
  $\textbf{let } G \textbf{ be } \text{to-type}(\text{translate-type}(E_1, e), \mathtt{bool}) \textbf{ in } \langle \neg G, \mathtt{bool}\rangle$
$\text{translate-type}(E_1\ O\ E_2, e) =$
  $\textbf{let } O^x \textbf{ be } \text{translate-op}(O) \textbf{ in}$
    $\textbf{let } G_1 \textbf{ be } \text{to-type}(\text{translate-type}(E_1, e), \text{op1-type}(O^x)) \textbf{ in}$
      $\textbf{let } G_2 \textbf{ be } \text{to-type}(\text{translate-type}(E_2, e), \text{op2-type}(O^x)) \textbf{ in}$
        $\langle G_1\ O^x\ G_2, \text{ret-type}(O^x)\rangle$

$\text{to-type}(\langle G, \mathtt{int}\rangle, \mathtt{int}) = G$
$\text{to-type}(\langle G, \mathtt{int}\rangle, \mathtt{bool}) = \mathtt{b2i(}G\mathtt{)}$
$\text{to-type}(\langle G, \mathtt{bool}\rangle, \mathtt{int}) = \mathtt{i2b(}G\mathtt{)}$
$\text{to-type}(\langle G, \mathtt{bool}\rangle, \mathtt{bool}) = G$
```

Figure 4-5: Functions for Translating Program Expressions to Logic Expressions

When translating a program expression, we need to obtain either an integer logic expression or a boolean logic expression. The function translate, given program expression E and symbolic environment e, returns an integer logic expression G^i representing E. For example, if the variable `v` is local in the environment e, then the expression `v+1` would be translated to $\mathtt{loc(v)} + 1$. The function translate-bool produces a boolean logic expression G^b. In the example, the result would be $\text{translate-bool}(\mathtt{v+1}, e) = \mathtt{i2b(loc(v)} + 1\mathtt{)}$. Finally, translate-seq is used for translating a sequence of program expressions into a sequence of integer logic expressions.

We next describe substitutions for the defined logic formulas. The special constructors for program variables do not change the way free and bound variables are defined for the logic expressions. In particular, program variables are always free, whereas logic variables follow the standard rules for bound variables. We use the common notation $F[G/x]$ to denote the substitution of the expression G for the logic variable x in formula F. This substitution follows the usual rules of renaming bound variables. We use $F[G/H(I)]$ to denote the substitution of the expression G for the program variable I in formula F. For example, $\texttt{glob(v)} = \texttt{loc(v)}[0/\texttt{glob(v)}] \equiv 0 = \texttt{loc(v)}$ and $\texttt{glob(v)} = \texttt{loc(v)}[0/\texttt{loc(v)}] \equiv \texttt{glob(v)} = 0$.

We also define a multiple substitution of logic expressions for program variables in a formula. Verification-condition generators symbolically execute BL programs and perform multiple substitutions on invariants to produce verification conditions. A symbolic execution uses a symbolic state, which is a mapping from program variables (a set of $H(I)$) to logic expressions (a set of G^i). For a symbolic state s, we denote by $\text{subst}(F, s)$ the logic formula obtained by substituting the logic expressions from s for the appropriate program variables in F. For example, if s maps `glob(v)` to 0, and `loc(v)` to 1, then $\text{subst}(\texttt{glob(v)} = \texttt{loc(v)}, s)$ gives $0 = 1$.

4.2.3 Semantics of Logic Formulas

We next define the semantics of the logic formulas. The semantics consists of a set of semantic domains and a set of valuation functions. Figure 4-6 presents the semantic domains that we use. The basic semantic domains *Int* and *Bool* are the usual integer numbers and truth values. We use the domain *One-Context* to represent a pair of an environment and a memory. These pairs are used to define the meaning of program variables. As explained, there are two groups of logic formulas: formulas with variables from only one program and formulas with variables from two programs. For the first group, we use a context that consists of one environment-memory pair, and for the second group, we use two such pairs. The same meta-variable c ranges over both groups of contexts. When we want to specify a context, we abbreviate $\langle m, a\rangle$ to m, a and $\langle\langle m_1, a_1\rangle, \langle m_2, a_2\rangle\rangle$ to m_1, a_1, m_2, a_2.

$$\begin{array}{l}
z \in \mathit{Int} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
b \in \mathit{Bool} = \{\mathit{true}, \mathit{false}\} \\
\quad \mathit{One\text{-}Context} = \text{Memory} \times \text{Environment} \\
c \in \mathit{Context} = \mathit{One\text{-}Context} + \mathit{One\text{-}Context} \times \mathit{One\text{-}Context}
\end{array}$$

Figure 4-6: Semantic Domains for Logic Formulas

Figure 4-7 presents the signatures of the valuation functions used in the semantics. $\mathcal{Z}$ maps integer constants used in the syntactic representation of the logic formulas to the integer numbers used in the semantic domain. The valuation functions $\mathcal{O}^a$, $\mathcal{O}^r$, and $\mathcal{O}^l$ map the syntactic representation of operators to their semantic equivalents. The functions $\mathcal{G}^i$, $\mathcal{G}^b$, and $\mathcal{F}$ are used for the meaning of the expressions and formulas in the logic. We define the meaning only of the expressions and formulas with no free logic variables. (The program variables are always free, and they get their meaning from the context.) We write $\mathcal{G}^i\llbracket G^i \rrbracket\ c$ to denote the value of integer expression G^i in context c. Similarly, we write $\mathcal{F}\llbracket F \rrbracket\ c$ to denote the value of formula F in context c. We are mostly interested in the valid formulas, and we abbreviate $\mathcal{F}\llbracket F \rrbracket\ c = \mathit{true}$ to $c \models F$ and say that formula F holds in context c.

$\mathcal{Z}$: Integer-constant $\rightarrow \mathit{Int}$
$\mathcal{O}^a$: Arithmetic-operator $\rightarrow \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int}$
$\mathcal{O}^r$: Relational-operator $\rightarrow \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Bool}$
$\mathcal{O}^l$: Logical-operator $\rightarrow \mathit{Bool} \rightarrow \mathit{Bool} \rightarrow \mathit{Bool}$
$\mathcal{G}^i$: Integer-expression $\rightarrow \mathit{Context} \rightarrow \mathit{Int}$
$\mathcal{G}^b$: Boolean-expression $\rightarrow \mathit{Context} \rightarrow \mathit{Bool}$
$\mathcal{F}$: Formula $\rightarrow \mathit{Context} \rightarrow \mathit{Bool}$

Figure 4-7: Signatures of the Valuation Functions

Figure 4-8 presents the valuation functions for integer and boolean expressions and validity of logic formulas. These functions define the meaning for all expressions and formulas with no free logic variables. (To obtain total meaning functions, we assign 0 as the meaning of operations not defined on integers, such as division by 0.) Bound logic variables are substituted with integers, as shown in the meaning of quantified formulas.[7] Program variables get their meaning from the context. There are two groups of contexts and two groups of formulas. Formulas have a meaning only for the appropriate contexts. Figure 4-8 shows the valuation functions for all meaningful combinations. If a formula F holds in all meaningful contexts, we write $\models F$.

4.2.4 Proof Rules for Logic Formulas

We need a set of proof rules for proving the validity of logic formulas. We do not specify the exact set of rules, but we assume the existence of rules for proving the formulas of the presented first-order predicate logic with integer variables. This set includes the standard rules for introduction and elimination of logical operators in the natural deduction style, the reflexivity and the congruence rules for the equality, and a group of rules for integer arithmetic. We write $\vdash F$ to denote that formula F with no free (logic) variables is provable using those rules. The proof system is required to be sound, namely for all F, if $\vdash F$, then $\models F$.

4.3 Compiler Analyses

In this section we present the verification of the results generated by a credible compiler analysis. The compiler expresses the analysis results in the form of *standard*

[7] The substitution is used in a slightly informal way; we would actually need to substitute Z, or use a context for logic formulas, which we want to avoid for simplicity of presentation.

$$
\begin{array}{l}
\mathcal{G}^i\llbracket Z\rrbracket\ c = \mathcal{Z}\llbracket Z\rrbracket \\
\mathcal{G}^i\llbracket G^i_1\ O^a\ G^i_2\rrbracket\ c = \mathcal{G}^i\llbracket G^i_1\rrbracket\ c\ \mathcal{O}^a\llbracket O^a\rrbracket\ \mathcal{G}^i\llbracket G^i_2\rrbracket\ c \\
\mathcal{G}^i\llbracket -G^i_1\rrbracket\ c = -\mathcal{G}^i\llbracket G^i_1\rrbracket\ c \\
\mathcal{G}^i\llbracket \texttt{b2i(}G^b\texttt{)}\rrbracket\ c = \textbf{if } \mathcal{G}^b\llbracket G^b\rrbracket\ c \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{G}^b\llbracket \texttt{true}\rrbracket\ c = \mathit{true} \\
\mathcal{G}^b\llbracket \texttt{false}\rrbracket\ c = \mathit{false} \\
\mathcal{G}^b\llbracket G^b_1\ O^l\ G^b_2\rrbracket\ c = \mathcal{G}^b\llbracket G^b_1\rrbracket\ c\ \mathcal{O}^l\llbracket O^l\rrbracket\ \mathcal{G}^b\llbracket G^b_2\rrbracket\ c \\
\mathcal{G}^b\llbracket \neg G^b_1\rrbracket\ c = \neg\mathcal{G}^b\llbracket G^b_1\rrbracket\ c \\
\mathcal{G}^b\llbracket G^i_1\ O^r\ G^i_2\rrbracket\ c = \mathcal{G}^i\llbracket G^i_1\rrbracket\ c\ \mathcal{O}^r\llbracket O^r\rrbracket\ \mathcal{G}^i\llbracket G^i_2\rrbracket\ c \\
\mathcal{G}^b\llbracket \texttt{i2b(}G^i\texttt{)}\rrbracket\ c = \mathcal{G}^b\llbracket G^i \neq 0\rrbracket\ c
\end{array}
$$

$$
\begin{array}{lll}
c \models G^b & \text{iff} & \mathcal{G}^b\llbracket G^b\rrbracket\ c = \mathit{true} \\
c \models F_1 \wedge F_2 & \text{iff} & c \models F_1 \text{ and } c \models F_2 \\
c \models F_1 \vee F_2 & \text{iff} & c \models F_1 \text{ or } c \models F_2 \\
c \models F_1 \Rightarrow F_2 & \text{iff} & c \models F_1 \text{ implies } c \models F_2 \\
c \models F_1 \Leftrightarrow F_2 & \text{iff} & c \models F_1 \Rightarrow F_2 \text{ and } c \models F_2 \Rightarrow F_1 \\
c \models \neg F_1 & \text{iff} & \text{not } c \models F_1 \\
c \models \forall x.\ F' & \text{iff} & c \models F'[z/x] \text{ for all } z \in \mathit{Int} \\
c \models \exists x.\ F' & \text{iff} & c \models F'[z/x] \text{ for some } z \in \mathit{Int}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{G}^i\llbracket \texttt{loc(}I\texttt{)}\rrbracket\ \langle m, a\rangle = m(a^l(I)) \\
\mathcal{G}^i\llbracket \texttt{glob(}I\texttt{)}\rrbracket\ \langle m, a\rangle = m(a^g(I)) \\
\mathcal{G}^i\llbracket \texttt{loc}_1\texttt{(}I\texttt{)}\rrbracket\ \langle m_1, a_1, m_2, a_2\rangle = m_1(a^l_1(I)) \\
\mathcal{G}^i\llbracket \texttt{glob}_1\texttt{(}I\texttt{)}\rrbracket\ \langle m_1, a_1, m_2, a_2\rangle = m_1(a^g_1(I)) \\
\mathcal{G}^i\llbracket \texttt{loc}_2\texttt{(}I\texttt{)}\rrbracket\ \langle m_1, a_1, m_2, a_2\rangle = m_2(a^l_2(I)) \\
\mathcal{G}^i\llbracket \texttt{glob}_2\texttt{(}I\texttt{)}\rrbracket\ \langle m_1, a_1, m_2, a_2\rangle = m_2(a^g_2(I))
\end{array}
$$

Figure 4-8: Valuation Functions for Expressions and Validity of Formulas

contexts. We first describe standard contexts and then formally define when they are correct. We finally present the standard verification-condition generator (StdVCG). The verifier for the compiler uses the StdVCG to generate a standard verification condition (StdVC) for a set of standard contexts. To prove that the analysis results are indeed correct, the compiler needs to prove that the StdVC holds.

4.3.1 Standard Contexts and Standard Invariants

We next describe the form in which the compiler presents analysis results. To allow separate compilation of procedures, we need support for modularity. It is usually obtained using procedure specifications, each of which consists of two formulas: a precondition formula and a postcondition formula. A precondition for a procedure P describes what P is allowed to assume at its entry, and a postcondition for a procedure P describes what P must preserve at its exit. We call a formula at the entry of a procedure a *standard input context*. It describes the context in which the procedure is called; we write F^{in} for such formulas. We call a formula at the exit of a procedure a *standard output context*. It describes the return context of the procedure; we write F^{out} for such formulas. We refer to a pair of a standard input context and a standard output context as a standard context.

Both groups of formulas F^{in} and F^{out} represent properties of only one program. Therefore, they can contain only `loc` and `glob` logic expression constructors. Further, the variables that appear in F^{in} for procedure P can be only the global variables of the program that contains P and the formal parameters of P, i.e., $\text{vars}(P) - \text{locals}(P)$. In F^{out}, only the global variables of the program, i.e., $\text{globals}(P)$, can appear.

The compiler may generate several contexts for the same procedure. This allows the compiler to express the results of context-sensitive interprocedural analyses. For each context of a procedure, the compiler generates a set of *standard invariants*. A standard invariant consists of a logic formula and a label. The formula represents an expression that the compiler claims to be true whenever the execution reaches the program point represented by the label. There should be at least one invariant in each loop to ensure that the StdVCG, which symbolically executes the procedure, terminates. One way to guarantee this is to place an invariant at every backward branch in the procedure. We do not explicitly require this placement, but we require that there be enough invariants.

We represent a standard invariant syntactically as L:`inv` F; meta-variable T ranges over standard invariants. For each context the compiler generates at least F^{in}, F^{out}, and a sequence T^*. These are the analysis results that the compiler needs to prove. The compiler also needs to generate more information to guide the StdVCG in generating the StdVC. For each context, the compiler generates a sequence K^*. This sequence represents the indices of the callee contexts that the compiler used at each call site in the analysis of the current context. We present more details after introducing some additional notation.

For procedure P for which the compiler generates n contexts, we write:

$$\begin{array}{ll} P \equiv \texttt{proc}\ I\texttt{(}I^*\texttt{)}\ D^*\ \{N^+\} & \\ \quad \texttt{std-invariants} & F_1^{in}\ F_1^{out}\ T_1^*\ K_1^* \\ & F_2^{in}\ F_2^{out}\ T_2^*\ K_2^* \\ & \vdots \\ & F_n^{in}\ F_n^{out}\ T_n^*\ K_n^*. \end{array}$$

We define several functions for standard contexts: contexts(P) returns the set of context indices for procedure P (in the general case, it is $\{1\ldots n\}$), in-context(P,k) returns the formula for the input context k of P (F_k^{in}), out-context(P,k) returns the formula for the output context k of P (F_k^{out}), context(P,k) returns a pair of input and output contexts. Also, the function std-invariant(P,k,L) returns the formula F from the standard invariant[8] $T \equiv L\texttt{:inv}\ F$ from context k of P. (We often use, instead of procedures, procedure identifiers as function arguments, e.g., contexts(I) returns a set of context indices for procedure named I.)

We next explain why we require the compiler to generate the sequence K^*. Each $K \equiv L\texttt{:}Z$ consists of the label L of a call node $L\texttt{:}I\texttt{(}E^*\texttt{)}$ from P and an integer literal Z that represents the index of the callee context for that call site. We write context-index(L,K^*) for the context index Z of label L in K^*. The StdVCG uses context(I,Z) at call site L to generate the StdVC. In general, the StdVCG cannot determine which callee context the compiler used at the call site (if the compiler generated several callee contexts). The StdVCG could generate a StdVC that includes all callee contexts (either context 1 is used, or context 2, or up to the total number of callee contexts), but the resulting StdVC would be prohibitively long. Therefore, we simply require the compiler to generate more additional information which represents which context the analysis used at each call site.

The compiler may use different contexts of the same callee procedure at different call sites. We illustrate this situation using an example. Suppose that the compiler analyzes some procedure `p` that has two calls to another procedure `q`. Suppose that `q` has one formal parameter `i`, and that the actual parameter of `q` is `0` for the first call, and `1` for the second call. Additionally, the compiler performs a context-sensitive interprocedural analysis and separately analyzes `q` for these two input contexts. Further, assume that these input contexts have different output contexts. For instance, a global variable `g` is `3` at the end of an execution of `q` in the first context, and it is `8` in the second context. In this example, the compiler would generate two contexts for procedure `q`: $F_1^{in} \equiv \texttt{loc(i)} = 0$, $F_1^{out} \equiv \texttt{glob(g)} = 3$, and $F_2^{in} \equiv \texttt{loc(i)} = 1$, $F_2^{out} \equiv \texttt{glob(g)} = 8$. In K^* for `p`, the compiler would represent that it used the first context for the first call, and the second context for the second call.

The compiler need not perform a context-sensitive analysis. It can perform a context-insensitive analysis and generate only one context for each procedure. For instance, in the previous example of procedure `q`, the compiler could generate the

[8] If there are many invariants with the same label, std-invariant returns the conjunction of all the formulas.

following context: $F^{in} \equiv$ `loc(i)` $= 0 \vee$ `loc(i)` $= 1$ and $F^{out} \equiv$ `glob(g)` $= 3 \vee$ `glob(g)` $= 8$. K^* would in this case represent that at both call sites the same context of `q` is used. Finally, the compiler need not perform an interprocedural analysis at all. It can always use for any procedure a context that represents that for all possible inputs to the procedure ($F^{in} \equiv$ `true`), the output can be anything ($F^{out} \equiv$ `true`). This corresponds to an intraprocedural analysis, which (in terms of dataflow analyses) kills all the information at call sites.

4.3.2 Analysis Correctness

We next discuss the notion of correct compiler analysis results. Most compiler analyses generate results that satisfy only *partial correctness*—a result is guaranteed to be correct if the program execution reaches the point of the result, but the program execution is not guaranteed to reach that point. Therefore, we require the compiler to prove that the generated standard invariants are only partially correct.

Formally, we first define when the standard invariants are correct for a standard context.

Definition 3 *A standard context k holds for a procedure P in a program Q, in notation $\models$* `std-invs(`k, P, Q`)`*, iff for all partial executions $\langle P^0, m^0, a\rangle \overset{+}{\dashrightarrow} \langle L, m, a\rangle$ of P for which $m^0, a \models F_k^{in}$, the following is satisfied:*

- *for all L'*:`inv` *F from T_k^*, if $L \equiv L'$, then $m, a \models F$; and*
- *for all L'*:`ret` *from P, if $L \equiv L'$, then $m, a \models F_k^{out}$.*

In other words, if the input context holds at the beginning of an execution, then each invariant should hold when the execution reaches it and the output context should hold when the execution reaches a return node. We extend the definition to procedures and programs.

Definition 4 *Standard invariants are correct (hold) for a procedure $P \in Q$, in notation $\models$* `std-invs(`P, Q`)`*, iff all contexts k of P hold.*

Definition 5 *Standard invariants are correct (hold) for a program Q, in notation $\models$* `std-invs(`Q`)`*, iff standard invariants hold for all procedures $P \in Q$.*

The compiler does not prove directly that $\models$ `std-invs(`Q`)`. Instead, the verifier uses the StdVCG to generate the StdVC for the invariants of all contexts of all procedures in program Q. We write F_Q^{vc} for the logic formula representing the StdVC of program Q. We present a sound StdVCG such that the validity of F_Q^{vc} implies that the invariants of Q are correct, i.e., if $\models F_Q^{vc}$, then $\models$ `std-invs(`Q`)`. (We show in Section 5.1 that the StdVCG is sound.) The compiler generates a proof $\vdash F_Q^{vc}$ using the proof rules from the logic. By the soundness of the proof rules, if $\vdash F_Q^{vc}$, then $\models F_Q^{vc}$. Therefore, the compiler actually proves that the StdVC holds for program Q, which then implies that all the standard invariants for program Q hold, and the compiler analysis results are thus correct.

4.3.3 Standard Verification-Condition Generator

We next present the algorithm for generating standard verification conditions. The StdVCG generates the StdVC for a program Q and a set of standard contexts for procedures in Q by symbolically executing each of those contexts. We first explain how those parts of the StdVC for each context combine in the whole StdVC. We then describe the algorithm that consists of two phases: the initial phase and the main phase.

Figure 4-9 shows the algorithm that generates $F^{vc}_{k,P,Q}$, a part of the StdVC for one context $k \in \text{contexts}(P) = \{1 \ldots n\}$ of procedure $P \in Q$. The conjunction of the verification conditions for all contexts of procedure P is the verification condition for that procedure, and the conjunction of the verification conditions for all procedures of program Q is the verification condition for that program:

$$F^{vc}_{P,Q} = \bigwedge_{k \in \{1 \ldots n\}} F^{vc}_{k,P,Q} \qquad \text{and} \qquad F^{vc}_{Q} = \bigwedge_{P \in Q} F^{vc}_{P,Q}.$$

The StdVC for a program is the whole F^{vc}_{Q}; we also refer to $F^{vc}_{k,P,Q}$ as StdVC.

The StdVCG generates $F^{vc}_{k,P,Q}$ by symbolically executing the context k of procedure P. We first describe the initial phase of the StdVCG, which prepares the procedure for the execution, and then the main phase of the StdVCG, which performs the execution using the main function Std.

In the initial phase, the StdVCG first uses the helper function merge-invariants to merge the invariants T^*_k into the procedure P, generating procedure P'. The invariants in T^*_k have the same labels as nodes in P; merge-invariants makes the labels unique and inserts the invariants in front of the appropriate nodes.[9] We assume that the function merge-invariants also checks that there are enough invariants so that the symbolic execution terminates.[10] If there are not enough invariants, the results are marked as incorrect. The StdVCG next creates a symbolic environment e for procedure $P \in Q$. This environment maps each local variable I_l of P to the logic expression $\texttt{loc(}I_l\texttt{)}$ and each global variable I_g of Q to the logic expression $\texttt{glob(}I_g\texttt{)}$. The StdVCG then creates a fresh symbolic state s^0 that maps all variables from the environment e to fresh logic variables. The sequence of all these fresh logic variables is in x^*, and $F^{vc}_{k,P,Q}$ is universally quantified over x^*.

Standard Verification-Condition Generator Main Function

The function Std performs the symbolic execution of procedure P'. This function takes three arguments: the label L of the current node to execute, the symbolic state s, and the set i of already (symbolically) executed standard invariants.

The execution starts from the first node of procedure P with a fresh state and

[9] The change is done so that the branches to a node in P are now branches to the first invariant in front of that node in P'.

[10] We use this organization only for an easier explanation of the StdVCG; in practice, the checks are done during the symbolic execution. The StdVCG keeps track of the nodes already symbolically executed on each path, and if a node is reached twice, there is a loop without invariant.

```
$P \equiv$ proc $I(I^*)$ $D^*$ $\{N^+\}$
    std-invariants $(F^{in}\ F^{out}\ T^*\ K^*)^*$

$F^{vc}_{k,P,Q} =$
 let $P'$ be merge-invariants$(T^*_k, P)$ in
  let $e$ be sym-environment$(P, Q)$ in
   let $\langle s^0, x^* \rangle$ be fresh-sym-state$(e)$ in
    letrec Std be $\lambda L\ s\ i.$
     matching $P'(L)$
      $\triangleright$ $L$:$I$=$E$ $\|$
       let $s'$ be translate-assign$(I, E, s, e)$ in
        Std$(L +_{P'} 1, s', i)$
      $\triangleright$ $L$:br($E$)$L'$ $\|$
       let $G$ be translate-branch$(E, s, e)$ in
        $(G \Rightarrow$ Std$(L', s, i)) \wedge$
        $(\neg G \Rightarrow$ Std$(L +_{P'} 1, s, i))$
      $\triangleright$ $L$:ret $\|$
       subst$(F^{out}_k, s)$
      $\triangleright$ $L$:$I$($E^*$) $\|$
       let $G^*$ be translate-call$(E^*, s, e)$ in
        let $k'$ be context-index$(L, K^*_k)$ in
         let $\langle F^{in}, F^{out} \rangle$ be context$(I, k')$ in
          let $\langle s', x^* \rangle$ be fresh-globals$(s)$ in
           subst$(F^{in}$, set-params$(I, G^*, s)) \wedge$
           $\forall x^*.$ subst$(F^{out}, s') \Rightarrow$ Std$(L +_{P'} 1, s', i)$
      $\triangleright$ $L$:inv $F$ $\|$
       if member-first$(L, i)$ then
        subst$(F, s)$
       else
        let $\langle s', x^* \rangle$ be fresh-sym-state$(e)$ in
         subst$(F, s) \wedge$
         $\forall x^*.$ subst$(F, s') \Rightarrow$ Std$(L +_{P'} 1, s'$, union$(\langle L, s' \rangle, i))$
       fi
     endmatching in
      $\forall x^*.$ subst$(F^{in}_k, s^0) \Rightarrow$ Std(start-label$(P'), s^0, \{\})$
```

Figure 4-9: Verification-Condition Generator for Standard Invariants

the empty set of invariants. This execution generates a part of $F^{vc}_{k,P,Q}$ that captures the correctness of all the invariants and the output context of context k. Since the invariants and the output context are required to be correct only for the appropriate input context, the whole $F^{vc}_{k,P,Q}$ is an implication—input context, substituted in the initial state, implies the result of the execution starting from that initial state.

We next describe how the algorithm executes each group of nodes. (Figure 4-10 shows the helper functions that the algorithm uses during the execution.)

N1. The execution of an assignment node I=E changes the symbolic state using the function translate-assign. This function changes only the symbolic expression representing the value of variable I; the rest of the state is unchanged. The execution proceeds from the assignment node to the next node.

N2. The execution of a branch node `br(`E`)`L' first translates the branch condition E using the function translate-branch. After that, the execution splits into two branches generating the appropriate condition on each branch.

N3. The execution of a return node `ret` generates the formula representing the output context in the current state and finishes this branch of the execution.

N4. The execution of a call node I`(`E^*`)` is more involved. The StdVCG first creates expressions G^* that symbolically represent the values of the actual parameters at the call site. These expressions will be replaced with the formal parameters in the callee input context. The StdVCG next decides, based on the sequence K^*_k, which callee context k' to use for this call. The StdVCG generates the part of $F^{vc}_{k,P,Q}$ that requires the input context F^{in} for context k' of I to hold for this call. This is done using the function set-params, which extends s with the mapping from the formal parameters of procedure I to the expressions G^*.

The call to procedure I can change the global variables in an arbitrary way. Therefore, the StdVCG creates a state s' in which all global variables from s are mapped to fresh logic variables x^*, while all local variables from s remain the same as before the call. The StdVCG next generates the part of $F^{vc}_{k,P,Q}$ that requires the output context F^{out} for context k' of I to hold in state s'. The symbolic execution continues with the state s' from the node after the call node.

N5. The execution of a standard invariant L`:inv` F depends on whether the invariant has been already executed or not.

N5.1. If the label L is in i (more precisely, in the first component of one of the pairs in i), the invariant is reached for the second time during this branch of the execution. The StdVCG substitutes the current symbolic state in the invariant, generates the resulting formula as a part of $F^{vc}_{k,P,Q}$ that needs to be proven, and the execution of this branch finishes.

N5.2. If label L is not in i, then the invariant is reached for the first time. The StdVCG similarly substitutes the current symbolic state in the invariant and generates the resulting formula as a part of $F^{vc}_{k,P,Q}$ that needs to be proven, but continues the execution. The execution continues from the node after the invariant with a fresh symbolic state s' and the pair $\langle L, s' \rangle$ added to the set of executed invariants. (For this StdVCG, i can be a set of labels only; we add the states technically to prove the soundness of the StdVCG.) The rest of this execution can assume that the invariant holds in state s'.

$$\begin{array}{l}
\text{translate-assign}(I, E, s, e) = s[\text{translate}(I, e) \mapsto \text{subst}(\text{translate}(E, e), s)] \\
\text{translate-branch}(E, s, e) = \text{subst}(\text{translate-bool}(E, e), s) \\
\text{translate-call}(E^*, s, e) = \text{subst-seq}(\text{translate-seq}(E^*, e), s)
\end{array}$$

Figure 4-10: Translation Functions for Verification-Condition Generators

4.4 Compiler Transformations

In this section we present the verification of the results generated by a credible compiler transformation. After performing the transformation on an original program, the compiler generates a transformed program and additional information in the form of *simulation contexts*. We first describe simulation contexts and then formally define the simulation of BL programs. We call the two programs Q_1 and Q_2, and we specify when Q_1 simulates Q_2. (Depending on the required (bi-)simulation correspondence, programs Q_1 and Q_2 can be the transformed and original programs and/or the original and transformed programs.) We finally present the simulation verification-condition generator (SimVCG). The verifier uses SimVCG to generate a simulation verification condition (StdVC) for a set of simulation contexts. To prove that Q_1 simulates Q_2, the compiler needs to prove that the SimVC holds.

4.4.1 Simulation Contexts and Simulation Invariants

We next describe the additional information that the compiler transformation generates beside the transformed program. Similarly as the compiler generates a standard context to summarize the analysis results for a procedure, the compiler generates a simulation context to represent a simulation relationship between a pair of procedures. A simulation context consists of two formulas that we call a *simulation input context* and a *simulation output context*. Simulation input contexts represent the correspondence between the states of the two programs at the entries of the two procedures. Simulation output contexts represent the correspondence between the states of the two programs at the exits of the two procedures.

Both simulation input and output contexts are formulas representing correspondence between two programs. Therefore, they can contain only the indexed constructors H for accessing program variables in logic expressions ($\mathtt{loc}_1$, $\mathtt{glob}_1$, $\mathtt{loc}_2$,

$\texttt{glob}_2$). We write J^{in} and J^{out} for the simulation input and output context, respectively. (Recall that we use J for formulas that explicitly represent pairs of variables that have the same value.) For procedures P_1 and P_2, only the variables from globals(P_1) $\cup$ params(P_1) $\cup$ globals(P_2) $\cup$ params(P_2) can appear in J^{in}, and only the variables from globals(P_1) $\cup$ globals(P_2) can appear in J^{out}.

The compiler may generate several simulation contexts involving the same procedure or pair of procedures. (This allows, for instance, the compiler to express the results of procedure specializations.) For each simulation context, the compiler generates a set of *simulation invariants* that represent relationships between the two programs. A simulation invariant consists of a logic formula and two labels, one from each program. We represent a simulation invariant syntactically as $L_1\texttt{,}L_2\texttt{:sim-inv}\ J$, where the first label L_1 is a label from P_1 and the second label L_2 is a label from P_2; meta-variable S ranges over simulation invariants.

The set of simulation invariants S^* for one simulation context may contain several invariants with the same first label, e.g., $L_1\texttt{,}L'_2\texttt{:sim-inv}\ J'$, or the same second label, e.g., $L'_1\texttt{,}L_2\texttt{:sim-inv}\ J'$.[11] We denote by set-sim-inv(L_1, S^*) the set of the simulation invariants whose first label is L_1. Informally, a set of simulation invariants for P_1 and P_2 holds if *for all* partial executions of P_1 that reach label L_1 of one of the invariants from the set, *there exists* a partial execution of P_2 that reaches label L_2 of one of the invariants $L_1\texttt{,}L_2\texttt{:sim-inv}\ J \in$ set-sim-inv(L_1, S^*), such that formula J holds for the states of the two procedures. We formalize this in Section 4.4.2.

Similar to the placement of standard invariants for the StdVCG, there should be enough simulation invariants so that the SimVCG can execute both procedures to generate the SimVC. These executions require that for each path in P_1, there exist an appropriate path in P_2. Therefore, there should be at least one simulation invariant in each loop in P_1. We do not require any particular placement of these invariants. Additionally, for each path from one invariant to another in P_1, there should be a path between the corresponding points in P_2. The placement of these invariants depends on the change that the transformation performs on the control flow graph of the procedure.

We have presented so far the formulas J^{in} and J^{out} and a sequence S^* that the compiler needs to generate for each simulation context. Analogous to the standard contexts, the compiler also needs to generate which simulation contexts the SimVCG should use for calls. We represent this as a sequence K^* where each $K \equiv L_1\texttt{,}L_2\texttt{:}Z$ consists of the labels L_1 and L_2 (of call nodes $L_1\texttt{:}I_1\texttt{(}E_1^*\texttt{)}$ from P_1 and $L_2\texttt{:}I_2\texttt{(}E_2^*\texttt{)}$ from P_2) and an integer literal Z that represents the *simulation* context index for procedures I_1 and I_2. We write sim-context-index(L_1, L_2, K^*) for the context index Z of labels L_1 and L_2 in K^*. We next present the other additional information that the verifier requires the compiler to generate for each simulation context.

The compiler may use the analysis results to perform the transformation. For different simulation contexts, the compiler may use different analysis results. The compiler represents the analysis results as standard contexts. Since there can be

[11] In general, there can be even many invariants with both labels being the same, e.g., $L_1\texttt{,}L_2\texttt{:sim-inv}\ J'$, but they can be replaced with: $L_1\texttt{,}L_2\texttt{:sim-inv}\ J \wedge J'$.

many standard contexts, the compiler needs to represent, for each simulation context, which standard contexts it uses for the two procedures. The compiler represents these contexts by their indices, as integer literals Z^1 and Z^2.

The compiler also generates the expressions for the initial values of local variables of P_2. The StdVCG and the SimVCG introduce fresh logic variables during the symbolic executions that generate verification-condition formulas. The StdVCG always universally quantifies the formulas over the new logic variables. We showed in Section 3.2.1 that the SimVCG needs to existentially quantify over the logic variables that represent the initial values of the variables from locals(P_2). To avoid existential quantification in the SimVC, we require the compiler to generate, for each simulation context, a sequence of integer logic expressions G^* that represent the initial values of local variables of P_2. These expressions can contain global variables from both programs, procedure parameters from both procedures, and local variables from P_1. Usually, the expression for a local variable from P_2 is that the initial value is the same as the initial value of the corresponding local variable from P_1, or that the initial value can be anything, e.g., the constant 0.

Finally, the compiler generates, for each context, a sequence of actions A^* to guide the SimVCG in generating the SimVC. We present the actions in detail later in the text. In summary, for procedures P_1 and P_2 for which the compiler generates n simulation contexts, we write:

$$\begin{array}{l}
P_1 \equiv \texttt{proc}\ I_1(I_1^*)\ D_1^*\ \{N_1^+\} \\
P_2 \equiv \texttt{proc}\ I_2(I_2^*)\ D_2^*\ \{N_2^+\} \\
\qquad \texttt{sim-invariants}\ \begin{array}[t]{l}
J_1^{in}\ J_1^{out}\ S_1^*\ K_1^*\ Z_1^1\ Z_1^2\ G_1^*\ A_1^* \\
J_2^{in}\ J_2^{out}\ S_2^*\ K_2^*\ Z_2^1\ Z_2^2\ G_2^*\ A_2^* \\
\vdots \\
J_n^{in}\ J_n^{out}\ S_n^*\ K_n^*\ Z_n^1\ Z_n^2\ G_n^*\ A_n^*.
\end{array}
\end{array}$$

We define several functions for simulation contexts: sim-contexts(P_1, P_2) returns a set of simulation context indices for procedures P_1 and P_2 (in the general case, it is $\{1 \ldots n\}$), sim-in-context(P_1, P_2, k) returns the formula for the simulation input context k of procedures P_1 and P_2 (J_k^{in}), sim-out-context(P_1, P_2, k) returns the formula for the simulation output context k of procedures P_1 and P_2 (J_k^{out}), and sim-context(P_1, P_2, k) returns a pair of simulation input and output contexts.

4.4.2 Transformation Correctness

We discussed the notion of correct compiler transformations in Section 2.1. We require the compiler to generate a transformed program that simulates the original program. This means that the transformed program is Q_1, and the original program is Q_2. Informally, Q_1 simulates Q_2 if Q_1 can generate only the results that Q_2 can generate. The result of a BL program execution is the values of the global variables at the end of the starting procedure. Therefore, we require the two programs to have the same number of global variables, and we additionally require corresponding global variables to have the same name.

We next define formally when one BL program simulates another. We first define the notion of simulation for a simulation context.

Definition 6 *A procedure $P_1 \in Q_1$ simulates a procedure $P_2 \in Q_2$ for a simulation context k, in notation $P_1, Q_1 \rhd_k P_2, Q_2$, iff for all partial executions $\langle P_1^0, m_1^0, a_1 \rangle \overset{+}{\dashrightarrow} \langle L_1, m_1, a_1 \rangle$ of P_1, there exists a partial execution*[12] *$\langle P_2^0, m_2^0, a_2 \rangle \overset{+}{\dashrightarrow} \langle L_2, m_2, a_2 \rangle$ of P_2 such that if $m_1^0, a_1, m_2^0, a_2 \models J_k^{in}$, then the following is satisfied:*

- *for all L_1', L_2'*`:sim-inv` *J' from S_k^*, if $L_1 \equiv L_1'$, then there exists L_1, L_2''*`:sim-inv` *J from* set-sim-inv(L_1, S_k^*) *such that $L_2 \equiv L_2''$ and $m_1, a_1, m_2, a_2 \models J$; and*
- *if $P_1(L_1) \equiv L_1$*`:ret`*, then $P_2(L_2) \equiv L_2$*`:ret` *and $m_1, a_1, m_2, a_2 \models J_k^{out}$; and*
- *if the partial execution of P_1 does not terminate, then the partial execution of P_2 also does not terminate.*

We extend the definition to procedures and programs.

Definition 7 *A procedure $P_1 \in Q_1$ simulates a procedure $P_2 \in Q_2$, in notation $P_1, Q_1 \rhd P_2, Q_2$, iff for all simulation contexts $k \in$* sim-contexts(P_1, P_2)*, $P_1, Q_1 \rhd_k P_2, Q_2$.*

Definition 8 *A program Q_1 simulates a program Q_2, in notation $Q_1 \rhd Q_2$ iff:*

- *for all pairs of procedures $P_1 \in Q_1$ and $P_2 \in Q_2$ for which there are simulation contexts, $P_1, Q_1 \rhd P_2, Q_2$; and*
- *one of the simulation contexts for the starting procedures for programs Q_1 and Q_2 is the following:*
  - *the simulation input context states that the two programs start with the same input at the beginning:*

    $$J^{in} \equiv \bigwedge \mathtt{glob}_1(I^g) = \mathtt{glob}_2(I^g) \land \bigwedge \mathtt{loc}_1(I_1^p) = \mathtt{loc}_2(I_2^p)$$

    for all global variables I^g from Q_1 (and Q_2) and for all parameters I_1^p of the starting procedure of Q_1 and their corresponding parameters[13] *I_2^p of the starting procedure of Q_2,*[14] *and*

[12] The exact order of quantifications in the definition is: for all a_1 possible at the start of P_1, for all m_1^0, for all a_2 possible at the start of P_2, and for all values of m_2^0 locations “below” the local variables of P_2, there exist some values of the other m_2^0 locations such that if $m_1^0, a_1, m_2^0, a_2 \models J_k^{in}$, then for all $\langle L_1, m_1, a_1 \rangle$ in the execution sequence of P_1, there exists a corresponding $\langle L_2, m_2, a_2 \rangle$ in the execution sequence of P_2.

[13] Since the values for the parameters of the starting procedures are supplied to the programs, we require the starting procedures to have the same number of parameters, but corresponding parameters need not have the same name. In general, two procedures need not even have the same number of parameters when there is a simulation context for those procedures.

[14] Additionally, the requirement for the simulation input context of the starting procedures implies that the simulation holds for all possible input values because the standard contexts Z^1 and Z^2, for this simulation context, need to have their input contexts `true`.

- *the simulation output context states that the two programs generate the same output at the end:*

$$J^{out} \equiv \bigwedge \texttt{glob}_1\texttt{(}I^g\texttt{)} = \texttt{glob}_2\texttt{(}I^g\texttt{)}$$

  for all global variables I^g from Q_1 (and Q_2).

This definition for $Q_1 \rhd Q_2$ formalizes the intuitive notion of the simulation—program Q_1 simulates program Q_2 if Q_2 can generate the same output data as Q_1 generates, provided that the two programs have the same input data. The simulation also requires *termination simulation*—Q_2 does not terminate if Q_1 does not terminate. The definition for $Q_1 \rhd Q_2$ (more specifically, $P_1, Q_1 \rhd_k P_2, Q_2$) additionally requires the simulation invariants to hold. Finally, the way that the SimVCG generates the SimVC also requires that the execution of Q_2 reaches a call site whenever the execution of Q_1 reaches a call site.

Analogous to the compiler analysis results and $\models$ `std-invs(`Q`)`, the compiler does not prove directly that $Q_1 \rhd Q_2$. Instead, the verifier uses the SimVCG to generate the SimVC for the simulation invariants of all simulation contexts of all procedures in programs Q_1 and Q_2. We write $F^{vc}_{Q_1,Q_2}$ for the logic formula representing the SimVC of programs Q_1 and Q_2. We design a sound SimVCG such that the validity of $F^{vc}_{Q_1,Q_2}$ implies that Q_1 simulates Q_2, i.e., if $\models F^{vc}_{Q_1,Q_2}$, then $Q_1 \rhd Q_2$. (More precisely, the standard verification conditions for Q_1 and Q_2 also need to hold: if $\models F^{vc}_{Q_1,Q_2}$ and $\models F^{vc}_{Q_1}$ and $\models F^{vc}_{Q_2}$, then $Q_1 \rhd Q_2$. We show in Section 5.2 that the SimVCG is sound.) The compiler generates a proof $\vdash F^{vc}_{Q_1,Q_2}$ using the proof rules from the logic. By the soundness of the proof rules, if $\vdash F^{vc}_{Q_1,Q_2}$, then $\models F^{vc}_{Q_1,Q_2}$. Therefore, the compiler actually proves that the SimVC holds for programs Q_1 and Q_2; that implies that program Q_1 simulates program Q_2, and therefore Q_1 can generate only the results that Q_2 can generate.

4.4.3 Simulation Verification-Condition Generator

We next present the algorithm for generating simulation verification conditions. The SimVCG generates the SimVC for two programs Q_1 and Q_2 and a set of simulation contexts for procedures in Q_1 and Q_2 by symbolically executing each of those simulation contexts. We first explain how those parts of the SimVC for each context combine in the whole SimVC. We then describe the algorithm that consists of two phases: the initial phase and the main phase.

Figures 4-11 and 4-12 show the algorithm that generates $F^{vc}_{k,P_1,Q_2,P_2,Q_2}$, a part of SimVC, for one simulation context $k \in \text{sim-contexts}(P_1, P_2) = \{1 \ldots n\}$ of procedures $P_1 \in Q_1$ and $P_2 \in Q_2$. Similar to the standard contexts, the conjunction of the verification conditions for all simulation contexts of procedures P_1 and P_2 is the verification condition for those procedures, and the conjunction of the verification conditions for all pairs of procedures (for which there is a simulation context) of programs Q_1 and Q_2 is the verification condition for those programs:

$$F^{vc}_{P_1,Q_1,P_2,Q_2} = \textstyle\bigwedge_{k \in \{1 \ldots n\}} F^{vc}_{k,P_1,Q_1,P_2,Q_2} \qquad \text{and} \qquad F^{vc}_{Q_1,Q_2} = \textstyle\bigwedge_{P_1 \in Q_1, P_2 \in Q_2} F^{vc}_{P_1,Q_1,P_2,Q_2}.$$

The SimVC for two programs is the whole $F^{vc}_{Q_1,Q_2}$; we also refer to $F^{vc}_{k,P_1,Q_2,P_2,Q_2}$ as SimVC.

The SimVCG generates $F^{vc}_{k,P_1,Q_1,P_2,Q_2}$ by symbolically executing the simulation context k for procedures P_1 and P_2. We first describe the initial phase of the SimVCG, which prepares the procedures for the executions, and then the main phase of the SimVCG, which performs the executions using the main function Sim.

In the initial phase, the SimVCG first uses the function merge-sim-invariants to merge the simulation invariants S^*_k into procedures $P_{1/2}$ generating procedures $P'_{1/2}$. This is similar to the way in which the StdVCG merges the standard invariants, but there are several differences. Merging the simulation invariant L_1,L_2:sim-inv J in P_1 only adds the node[15] L_1:sim-inv that represents that there is a simulation invariant at L_1. In P_2, merging adds the node L_2:sim-inv J,L_1 to record the actual formula of the invariant. The reason for this is that there can be many invariants for the same node L_1 in P_1, and they all share the same L_1:sim-inv node. We call each of the nodes L_1:sim-inv and L_2:sim-inv J,L_1 a half of the simulation invariant.

We explained in Section 3.2.1 the most notable difference between the StdVCG and the SimVCG—the SimVCG uses a sequence of actions to guide the symbolic executions, whereas the StdVCG has a fixed structure of the symbolic execution. After the function merge-sim-invariants generates $P'_{1/2}$, the SimVCG applies the function action-tree to A^*_k to obtain an action tree t^0. The action tree contains the whole step-by-step description for the interleaving of the symbolic executions of $P'_{1/2}$. We describe action trees and all actions later in the text.

The SimVCG next applies the function check-std-contexts. This function returns the indices Z^1_k of P_1 and Z^2_k of P_2 after checking that those indices are correct for the standard contexts of P_1 and P_2. The standard invariants from standard contexts k_1 and k_2 can be used in the simulation context k.

The SimVCG next creates symbolic environments $e_{1/2}$ for procedures $P_{1/2} \in Q_{1/2}$. These environments map local variables I_l of their respective procedures to logic expressions loc$_{1/2}$(I_l) and global variables I_g of their respective programs to logic expressions glob$_{1/2}$(I_g). The SimVCG next creates the initial symbolic states s^0_1 and s^0_2 for the two procedures using the following algorithm:

- first create s^0_1 that maps all variables from vars(P_1) to fresh logic variables and put those logic variables in x^*;
- then create s^0_2 that maps all variables from vars(P_2)$-$locals(P_2) to the following logic variables:
  - if variable I_2 appears in some pair H_1(I_1),H_2(I_2) in var-pairs(J^{in}_k), map H_2(I_2) in s^0_2 to the logic variable that H_1(I_1) is mapped to in s^0_1, and
  - if variable I_2 does not appear in any pair in var-pairs(J^{in}_k), map H_2(I_2) in s^0_2 to a fresh logic variable, and add that logic variable to x^*;

[15] A label renaming is also performed to make all the labels unique.

```
$P_1 \equiv \texttt{proc}\ I_1\texttt{(}I_1^*\texttt{)}\ D_1^*\ \{N_1^+\}$
$P_2 \equiv \texttt{proc}\ I_2\texttt{(}I_2^*\texttt{)}\ D_2^*\ \{N_2^+\}$
    $\texttt{sim-invariants}\ (J^{in}\ J^{out}\ S^*\ K^*\ Z^1\ Z^2\ G^*\ A^*)^*$

$F^{vc}_{k,P_1,Q_1,P_2,Q_2} =$
 let $\langle P_1', P_2' \rangle$ be merge-sim-invariants$(S_k^*, P_1, P_2)$ in
  let $t^0$ be action-tree$(A_k^*, P_1', P_2')$ in
   let $\langle k_1, k_2 \rangle$ be check-std-contexts$(P_1, Z_k^1, P_2, Z_k^2)$ in
    let $\langle e_1, e_2 \rangle$ be sim-sym-environments$(P_1, Q_1, P_2, Q_2)$ in
     let $\langle s_1^0, s_2^0, x^* \rangle$ be initial-sim-sym-states$(e_1, e_2, J_k^{in}, P_2, G_k^*)$ in
      letrec Sim be $\lambda L_1\ s_1\ L_2\ s_2\ i\ t.$
       matching root$(t)$
        $\triangleright$ $\texttt{execute}_1$ $[\![$
         matching $P_1'(L_1)$
          $\triangleright$ $L_1\texttt{:}I\texttt{=}E$ $[\![$
           let $s'$ be translate-assign$(I, E, s_1, e_1)$ in
            Sim$(L_1 +_{P_1'} 1, s', L_2, s_2, i, \mathrm{left}(t))$
          $\triangleright$ $L_1\texttt{:br(}E\texttt{)}L'$ $[\![$
           let $G$ be translate-branch$(E, s_1, e_1)$ in
            $(G \Rightarrow \mathrm{Sim}(L', s_1, L_2, s_2, i, \mathrm{left}(t))) \wedge$
            $(\neg G \Rightarrow \mathrm{Sim}(L_1 +_{P_1'} 1, s_1, L_2, s_2, i, \mathrm{right}(t)))$
         endmatching
        $\triangleright$ $\texttt{execute}_2\ B$ $[\![$
         matching $P_2'(L_2)$
          $\triangleright$ $L_2\texttt{:}I\texttt{=}E$ $[\![$
           let $s'$ be translate-assign$(I, E, s_2, e_2)$ in
            Sim$(L_2, s_1, L_2 +_{P_2'} 1, s', i, \mathrm{left}(t))$
          $\triangleright$ $L_2\texttt{:br(}E\texttt{)}L'$ $[\![$
           let $G$ be translate-branch$(E, s_2, e_2)$ in
            if $B \equiv \texttt{true}$ then
             $G \wedge \mathrm{Sim}(L_1, s_1, L', s_2, i, \mathrm{left}(t))$
            else
             $\neg G \wedge \mathrm{Sim}(L_1, s_1, L_2 +_{P_2'} 1, s_2, i, \mathrm{left}(t))$
            fi
          $\triangleright$ $L_2\texttt{:sim-inv}\ J\texttt{,}L_1$ $[\![$
           Sim$(L_1, s_1, L_2 +_{P_2'} 1, s_2, i, \mathrm{left}(t))$
         endmatching
        $\triangleright$ ... continued in Figure 4-12
       endmatching in
        $\forall x^*.$ subst-sim$(J_k^{in}, s_1^0, s_2^0) \Rightarrow$
         Sim(start-label$(P_1'), s_1^0,$ start-label$(P_2'), s_2^0, \{\}, t^0) \wedge$
         subst(in-context$(P_1, k_1), s_1^0) \wedge$ subst(in-context$(P_2, k_2), s_2^0)$
```

Figure 4-11: Simulation Verification-Condition Generator, Part 1

```
letrec Sim be $\lambda L_1\ s_1\ L_2\ s_2\ i\ t$.
  matching $\mathrm{root}(t)$
    $\triangleright$ ... continued from Figure 4-11
    $\triangleright$ stop $\|$
      false
    $\triangleright$ split $F$ $\|$
      let $F'$ be $\mathrm{subst}(F, s_1)$ in
        $(F' \Rightarrow \mathrm{Sim}(L_1, s_1, L_2, s_2, i, \mathrm{left}(t))) \wedge$
        $(\neg F' \Rightarrow \mathrm{Sim}(L_1, s_1, L_2, s_2, i, \mathrm{right}(t)))$
    $\triangleright$ use-analysis$_1$ $\|$
      $\mathrm{subst}(\text{std-invariant}(P_1, k_1, L_1), s_1) \Rightarrow \mathrm{Sim}(L_1, s_1, L_2, s_2, i, \mathrm{left}(t))$
    $\triangleright$ use-analysis$_2$ $\|$
      $\mathrm{subst}(\text{std-invariant}(P_2, k_2, L_2), s_2) \Rightarrow \mathrm{Sim}(L_1, s_1, L_2, s_2, i, \mathrm{left}(t))$
    $\triangleright$ execute-both $\|$
      matching $P'_2(L_2)$
        $\triangleright$ $L_2$:ret $\|$
          $\text{subst-sim}(J_k^{out}, s_1, s_2)$
        $\triangleright$ $L_2$:$I_2$($E_2^*$) $\|$
          let $G_2^*$ be $\text{translate-call}(E_2^*, s_2, e_2)$ in
            matching $P'_1(L_1)$
              $\triangleright$ $L_1$:$I_1$($E_1^*$) $\|$
                let $G_1^*$ be $\text{translate-call}(E_1^*, s_1, e_1)$ in
                  let $k'$ be $\text{sim-context-index}(L_1, L_2, K_k^*)$ in
                    let $\langle J^{in}, J^{out}\rangle$ be $\text{sim-context}(I_1, I_2, k')$ in
                      let $\langle s'_1, s'_2, x^*\rangle$ be $\text{fresh-sim-globals}(s_1, s_2, J^{out})$ in
                        $\text{subst-sim}(J^{in}, \text{set-params}(I_1, G_1^*, s_1),$
                                            $\text{set-params}(I_2, G_2^*, s_2)) \wedge$
                        $\forall x^*.\ \text{subst-sim}(J^{out}, s'_1, s'_2) \Rightarrow$
                              $\mathrm{Sim}(L_1 +_{P'_1} 1, s'_1, L_2 +_{P'_2} 1, s'_2, i, \mathrm{left}(t))$
            endmatching
        $\triangleright$ $L_2$:sim-inv $J$,$L_1$ $\|$
          if $\text{member-first}(\langle L_1, L_2\rangle, i)$ then
            $\text{subst-sim}(J, s_1, s_2)$
          else
            let $\langle s'_1, s'_2, x^*\rangle$ be $\text{fresh-sim-sym-states}(s_1, s_2, J)$ in
              $\text{subst-sim}(J, s_1, s_2) \wedge$
              $\forall x^*.\ \text{subst-sim}(J, s'_1, s'_2) \Rightarrow$
                    $\mathrm{Sim}(L_1 +_{P'_1} 1, s'_1, L_2 +_{P'_2} 1, s'_2,$
                          $\mathrm{union}(\langle\langle L_1, L_2\rangle, \langle s_1, s_2\rangle, \mathrm{left}(t)\rangle, i), \mathrm{left}(t))$
          fi
      endmatching
endmatching in
```

Figure 4-12: Simulation Verification-Condition Generator, Part 2

- finally map the local variables of P_2 in s_2^0 to the logic expressions obtained by substituting the program variables appearing in G_k^* with the appropriate logic variables from s_1^0 and s_2^0.

The sequence of all fresh logic variables is in x^*, and $F^{vc}_{k,P_1,Q_1,P_2,Q_2}$ is universally quantified over all those variables.

The algorithm for the SimVCG uses the function subst-sim to perform substitutions. For a formula $J \equiv F\texttt{,}(H_1\texttt{(}I_1\texttt{)}\texttt{,}H_2\texttt{(}I_2\texttt{)})^*$ and symbolic states s_1 and s_2, subst-sim(J, s_1, s_2) does the following:

- substitute all program variables in formula F using the mappings $s_{1/2}$; call the result $F' \equiv \text{subst}(\text{subst}(F, s_1), s_2)$, and
- for all pairs $H_1\texttt{(}I_1\texttt{)}\texttt{,}H_2\texttt{(}I_2\texttt{)}$ from var-pairs(J), check if the logic expressions $G_{1/2}$ to which $s_{1/2}$ map $H_{1/2}\texttt{(}I_{1/2}\texttt{)}$ are syntactically identical, and:
  - if the expressions are identical, do not add anything to F' for this pair; or
  - if the expressions are not identical, add $G_1 = G_2$ as a conjunct to F', and
- finally return the whole conjunction F'.

This way the SimVCG performs several rules from the logic, most notably the congruence rule for equality, which results in a much shorter SimVC.

Simulation Verification-Condition Generator Main Function

The function Sim performs the symbolic executions of procedures P_1' and P_2'. This function takes six arguments: the label L_1 of the current node to execute in P_1', the symbolic state s_1 of P_1', the label L_2 of the current node to execute in P_2', the symbolic state s_2 of P_2', the set i of already (symbolically) executed *simulation* invariants, and the action tree t.

The executions start from the first nodes of procedures P_1 and P_2 with the created initial symbolic states, the empty set of invariants, and t^0 obtained from A_k^*. The executions generate a part of $F^{vc}_{k,P_1,Q_1,P_2,Q_2}$ that captures the correctness of all simulation invariants and the simulation output context of context k. Since the invariants and the output context are required to be correct only for the appropriate input context, the whole $F^{vc}_{k,P_1,Q_1,P_2,Q_2}$ is an implication—the input context, substituted in the initial states[16], implies the result of the executions starting from the initial states. Additionally, we require the simulation input context to imply the input contexts for standard contexts k_1 for P_1 and k_2 for P_2.

We next describe how Sim uses the action tree t. At each step, Sim performs the action from the root of the action tree. We write root(t) for the action from the root. At the branch nodes in P_1', Sim splits the execution into two paths. That is where

[16] The input context J_k^{in} is substituted in s_1^0 and s_2^0 using the function subst-sim. For the initial states, all pairs of variables that appear in J_k^{in} have the same value, and their equality is not added to $F^{vc}_{k,P_1,Q_1,P_2,Q_2}$, but it is still represented within the symbolic states.

Sim uses the subtrees of the action tree to continue the executions of the two paths. We write left(t) and right(t) for the left and right subtree, respectively. For most other actions, Sim does not require t to be a tree, but only a list. In those cases, Sim continues the execution with all the actions but the root. We also write left(t) for that list of actions. All the functions that operate on a tree t generate an error if t does not have the appropriate form.[17]

Finally, we describe how Sim performs each group of actions from the root of the action tree t.

A1. The action `execute`$_1$ executes the current node in P_1'. This action can be used only when the current node is an assignment node or a conditional branch node. (Otherwise, **matching** fails and the SimVCG signals an error.) The SimVCG executes assignment and branch nodes similarly as the StdVCG does.

A1.1. The execution of an assignment node changes the state s_1 and the execution continues from the next node in P_1'.

A1.2. The execution of a branch node splits the execution of P_1' into two branches and each of them generates an implication with the appropriate condition. These conditions represent formulas that hold when the particular branch is taken.

A2. The action `execute`$_2$ executes the current node in P_2'. This action can be used when the current node in P_2' is an assignment node, a conditional branch node, or a half of a simulation invariant.

A2.1. The execution of an assignment node changes the state s_2 and proceeds from the next node in P_2', analogously as the execution for `execute`$_1$.

A2.2. The execution of a branch node in P_2' is different than the execution in P_1'. Since the simulation condition requires for all paths in P_1' that there exist a corresponding path in P_2', only one branch is taken in P_2'. The action represents with B the branch that the SimVCG should take. Further, the branch conditions of P_1' are used as assumptions in the SimVC, whereas the branch conditions of P_2' are used as conclusions—the compiler needs to prove, when P_1' takes some branches, that P_2' indeed takes the branches that the compiler claims P_2' takes.

A2.3. The execution of half of a simulation invariant only moves past the invariant. This is sound because the execution of P_2' has to reach *any* half of a simulation invariant in P_2' (corresponding to the half in P_1'); it need not be the first half that the execution gets to.

[17] We use action trees only technically to prove that the SimVCG is sound. In practice, the SimVCG uses a mutable list l that is initialized to A_k^*. At each step, instead of root(t), the SimVCG applies head!(l) that returns the head of the list and sets the list to the tail. If the list is empty, head!() generates an error; the SimVCG always terminates because the list is finite.

A3. The action `stop` can be used at any point; it generates `false` as a part of SimVC that needs to be proven and finishes the execution. In general, `false` can be proven only if it is implied by `false`. Therefore, the compiler uses this action only when it knows that the path taken by the symbolic execution of P_1' is not possible during the concrete execution of P_1. For example, the path that branches from $L_1\texttt{:br(FALSE)}L'$ to L' during the symbolic execution is actually never taken. (The compiler usually does not need to use `stop` to prove that the transformed program simulates original program. However, if we also required the other simulation direction, the compiler would have to use `stop` when the compiler, for example, eliminated a branch that cannot be taken.)

A4. The action `split` can also be used at any point. The compiler uses `split` F to instruct the SimVCG to split the execution of P_1' into two paths, although there are no conditional branch nodes. (The SimVCG otherwise splits the execution of P_1' only at conditional branch nodes.) Both paths continue from the next node, but one of them has the condition F and the other has the negation of F (with proper substitutions). For example, the compiler uses this to prove that $L_1\texttt{:z=x*y;}L\texttt{:}$ simulates $L_2\texttt{:br(y==0)}L'\texttt{;z=x*y;br(TRUE)}L\texttt{;}L'\texttt{:z=0;}L\texttt{:}$. If `x` and `y` are the same in both programs before these sequences, then `z` is the same after the sequences. The compiler would use $\texttt{split loc}_1\texttt{(y)} = 0$ to create two paths of execution of P_1', each of which implies the corresponding path in the longer program sequence P_2'. (The compiler uses `split`, in general, when it merges two branches into one.)

A5. The actions $\texttt{use-analysis}_1$ and $\texttt{use-analysis}_2$ include the results of compiler analyses of procedures P_1 and P_2 in the SimVC. This can be done at any point at which there is a standard invariant in context k_1 (for P_1) or k_2 (for P_2). (Note that the program variables in the standard invariants are represented with H constructors without indices, whereas states s_1 and s_2 map variables with indices. Therefore, to be precise, we should replace $H_{1/2}$ for H in the invariants before the substitution.)

A6. The action `execute-both` simultaneously executes a node from each procedure. The two nodes can be both return nodes, both call nodes, or both halves of some simulation invariant.

A6.1. If both nodes are return nodes, the SimVCG adds to the SimVC the simulation output context substituted into the current symbolic states of the two procedures.

A6.2. If both nodes are call nodes, the process is more involved, but similar to the generation of the StdVC. The SimVCG first creates G_1^* and G_2^* that symbolically represent the actual parameters of callees at the call sites. These expressions will be replaced for the formal parameters in the simulation input context of the two callees. The SimVCG next decides, based on the sequence K_k^*, which simulation context k' to use for these call nodes. Next, the SimVCG generates a part of SimVC that requires the simulation input context J^{in} for context k' of I_1 and I_2 to hold for these calls. The function set-params extends the mappings $s_{1/2}$ with the mappings from the formal parameters of procedures $I_{1/2}$ to expressions $G^*_{1/2}$.

Callees can arbitrarily change the global variables of the programs. Therefore, the SimVCG creates new states s'_1 and s'_2 to represent the states of the programs after the calls. This is done using the function fresh-sim-sym-states, which first maps all global variables from s_1 to fresh logic variables, and then, based on the pairs of variables in J^{out}, maps global variables from s_2 to either the appropriate logic variables in s_1 or to fresh logic variables. The sequence of all the introduced logic variables is returned in x^*. All the local variables in s_1 and s_2 remain the same as they were before the call. Finally, the SimVCG generates a part of the SimVC that requires the simulation output context J^{out} for context k' of procedures $I_{1/2}$ to hold in states $s'_{1/2}$, and the symbolic execution continues from the nodes after the call nodes.

Observe that the part of the SimVC generated for call nodes requires only that the *simulation* input context k' for procedures $I_{1/2}$ hold. That simulation context is valid only for some standard contexts for $I_{1/2}$. However, the SimVC does not explicitly require those standard contexts to hold at every call site. Instead, the SimVC requires the simulation input context to imply the standard contexts in which the simulation context is valid. This is included in the SimVC only once for each simulation context, as shown at the bottom of Figure 4-11. Similarly, the SimVC for call nodes does not include the output contexts of the standard contexts, but those relationships are represented in the simulation output context.

A6.3. If both nodes are halves of a simulation invariant, the execution depends on whether the invariant has been already executed during this branch of the execution.

A6.3.1. If the pair of labels $\langle L_1, L_2 \rangle$ is in i (more precisely, in the first component of one of the triples in i), the simulation invariant has been already executed. The SimVCG substitutes the current symbolic states s_1 and s_2 in the invariant formula J. The resulting formula is generated as the part of SimVC that needs to be proven, and the executions of these paths finish.

A6.3.2. If the pair of labels $\langle L_1, L_2 \rangle$ is not in i, then the simulation invariant is reached for the first time. The SimVCG similarly substitutes the current symbolic states in the invariant and generates the resulting formula as the part of SimVC that needs to be proven. The executions do not finish, though, but move past the invariant halves in P'_1 (the node L_1`:sim-inv`) and P'_2 (the node L_2`:sim-inv` J`,`L_1). The executions continue with fresh symbolic states $s'_{1/2}$, created with respect to J, and the triple $\langle\langle L_1, L_2 \rangle, \langle s'_1, s'_2 \rangle, t\rangle$ added to the set of executed invariants. (For this SimVCG, i can be a set of label pairs $\langle L_1, L_2 \rangle$

only; we add the states and the action tree technically to prove the soundness of the SimVCG.) The rest of the executions can assume that the invariant holds in states $s'_{1/2}$.

Chapter 5

Soundness Proofs

In this chapter we prove the soundness of the standard verification-condition generator (StdVCG) and the simulation verification-condition generator (SimVCG) presented in Chapter 4. The StdVCG generates a standard verification condition (StdVC) for a program Q and a set of standard contexts for Q; the StdVCG is sound if the validity of the StdVC implies that those contexts indeed hold for program Q. The SimVCG generates a simulation verification condition (SimVC) for a pair of programs Q_1 and Q_2 and a set of simulation contexts for those programs; the SimVCG is sound if the validity of the SimVC implies that Q_1 simulates Q_2. Our proof of the soundness of the StdVCG follows a proof by Necula [35, Appendix A], and our proof of the soundness of the SimVCG combines the techniques from the proof by Necula and a proof by Rinard [41].

Before presenting the soundness proofs, we introduce some additional notation and present lemmas that we will use in the proofs. For the brevity of the presentation, we will consider logic expressions without the constructors H. Each program variable I is represented in logic expressions simply by its name I. We assume that local and global variables have different names. Symbolic execution of a procedure P uses a symbolic state s that maps all program variables from P to logic expressions. We represent a symbolic state as $\{I^* \mapsto G^*\}$; usually, the logic expressions are just logic variables and $s = \{I^* \mapsto x^*\}$. The notation with sequences represents a mapping between corresponding elements: $s = \{I^{(1)} \mapsto x^{(1)}, \ldots, I^{(n)} \mapsto x^{(n)}\}$, where $x^{(n)}$ denotes the n-th element of sequence x^*. We denote by $\mathrm{subst}(F, s)$ the logic formula obtained by substituting the logic expressions from s for the appropriate program variables in formula F. For a variable I that occurs in s, we write $s(I)$ for the logical expression corresponding to I.

In the course of the proofs, we show properties of partial executions of procedures. Recall that $\langle P^0, m^0, a\rangle \overset{+}{\dashrightarrow} \langle L, m, a\rangle$ represents a partial execution of an *activation* of procedure P, and we therefore use the abbreviated form $\langle L, m, a\rangle$ of configurations $\langle L, m, a, p, h, P\rangle$. We refer to the execution of BL programs on a machine with configurations $\langle L, m, a\rangle$ ($\langle L, m, a, p, h, P\rangle$) as the *concrete execution*, to distinguish it from the *symbolic execution*. The concrete execution operates on the *concrete state*, which consists of the memory m and the environment a, whereas the symbolic execution operates on the symbolic state s.

Each symbolic state s corresponds to a set of the concrete states m, a. A symbolic state $s = \{I^* \mapsto G^*\}$ maps program variables to logic expressions, whereas an environment a maps the variables to the addresses, and a memory m maps the addresses to the concrete values from the Value domain so that $m(a(I)) = Z$. The logic expressions G^* in the symbolic state s include logic variables. Let all those variables be from a set/sequence x^*. In the proofs, we use substitutions that map logic variables to integer values. We represent a substitution that maps the logic variables x^* to some concrete values Z^* as $\{x^* \mapsto Z^*\}$; each such substitution specifies one particular mapping from s to m, a.

We use the meta-variable τ to represent substitutions. We denote by $\tau' \cup \{x^* \mapsto Z^*\}$ the union of a substitution τ' (which does not map any variable from some x^*) and a mapping from x^* to Z^*. We denote with $\tau(F)$ the formula obtained by substituting the integer values from τ for the appropriate logic variables in F. We usually apply a substitution τ for logic variables after substituting the program variables in a formula F with the logic expressions from a symbolic state s: $\tau(\text{subst}(F, s))$. We say that a symbolic state s and a concrete state m, a *coincide* (with respect to a substitution τ) for some variables I^* if $\models \tau(s(I)) = m(a(I))$ for all variables I from I^*.

We use several lemmas to prove the main soundness theorems. The first two lemmas (for one program or for two programs) assert that if the symbolic states coincide with the concrete states for all program variables in a formula, then the formula is valid in the symbolic states if and only if it is valid in the concrete states. (We break each equivalence into two implications for easier referral later in the text.) We omit the details of the proofs of the lemmas.

Lemma 1 *(Standard Congruence) Let P be any procedure and F be any formula with program variables from $I^* \subseteq \text{vars}(P)$. Let m and a be any memory and environment for P. Let the symbolic state be $s \supseteq \{I^* \mapsto G^*\}$, and let x^* be all logic variables that occur in G^*. If a substitution $\tau \supseteq \{x^* \mapsto Z^*\}$ is such that $\models \tau(s(I)) = m(a(I))$ for all I that occur in F, then:*

$$\textit{if } m, a \models F, \textit{ then } \models \tau(\text{subst}(F, s)) \tag{5.1}$$

and

$$\textit{if } \models \tau(\text{subst}(F, s)), \textit{ then } m, a \models F. \tag{5.2}$$

Proof: Structural induction on the formula F (actually the formula $\text{subst}(F, s)$).

Lemma 2 *(Simulation Congruence) Let P_1 and P_2 be any pair of procedures and J be any formula with program variables from $I_1^* \subseteq \text{vars}(P_1)$ and $I_2^* \subseteq \text{vars}(P_2)$. Let $m_{1/2}$ and $a_{1/2}$ be any memory and environment for $P_{1/2}$. Let the symbolic states be $s_{1/2} \supseteq \{I_{1/2}^* \mapsto G_{1/2}^*\}$, and let x^* be all logic variables that occur in G_1^* or G_2^*. If a substitution $\tau \supseteq \{x^* \mapsto Z^*\}$ is such that $\models \tau(s_{1/2}(I_{1/2})) = m_{1/2}(a_{1/2}(I_{1/2}))$ for all $I_{1/2}$ that occur in J, then:*

$$\textit{if } m_1, a_1, m_2, a_2 \models J, \textit{ then } \models \tau(\text{subst-sim}(J, s_1, s_2)) \tag{5.3}$$

and

$$\textit{if } \models \tau(\text{subst-sim}(J, s_1, s_2)), \textit{ then } m_1, a_1, m_2, a_2 \models J. \tag{5.4}$$

Proof: Structural induction on the formula J (actually subst-sim(J, s_1, s_2)).

The next three lemmas (for assignment, branch, and call nodes) assert that the translation functions for program expressions (Figure 4-5) are correct with respect to expression evaluation (Figure 4-2). The proofs of all three lemmas are by structural induction on the particular expressions. We omit the details of these proofs.

Lemma 3 *(Assignment Translation) Let P be any procedure and E be any expression that can occur in an assignment node in that procedure. Let e and s be, respectively, the symbolic environment and a symbolic state for a symbolic execution of P. Let a and m be, respectively, an environment and a memory for a concrete execution of P. If a substitution τ is such that $\models \tau(s(I)) = m(a(I))$ for all $I \in \mathrm{vars}(P)$, then $\models \tau(\mathrm{subst}(\mathrm{translate}(E, e), s)) = m(a(E))$.*

Proof: Structural induction on the expression E.

Lemma 4 *(Branch Translation) Let P be any procedure and E be any expression that can occur in a branch node in that procedure. Let e and s be, respectively, the symbolic environment and a symbolic state for a symbolic execution of P. Let a and m be, respectively, an environment and a memory for a concrete execution of P. If a substitution τ is such that $\models \tau(s(I)) = m(a(I))$ for all $I \in \mathrm{vars}(P)$, then $m(a(E)) \not\equiv \mathtt{0}$ if and only if $\models \tau(\mathrm{translate\text{-}branch}(E, s, e))$.*

Proof: Structural induction on the expression E.

Lemma 5 *(Call Translation) Let P be any procedure and E^* be any sequence of expressions that can occur in a call node in that procedure. Let e and s be, respectively, the symbolic environment and a symbolic state for a symbolic execution of P. Let a and m be, respectively, an environment and a memory for a concrete execution of P. If a substitution τ is such that $\models \tau(s(I)) = m(a(I))$ for all $I \in \mathrm{vars}(P)$, then $\models \tau(G^{(n)}) = m(a(E^{(n)}))$ for each $E^{(n)}$ from the sequence E^* and the respective expression $G^{(n)}$ from the sequence $G^* = \mathrm{translate\text{-}call}(E^*, s, e)$.*

Proof: Induction on the length of the sequence E^* and application of Lemma 3 for each of those expressions.

5.1 Soundness of Standard Verification-Condition Generator

In this section we prove that the standard verification-condition generator (presented in Section 4.3) is sound: for every program Q and every set of standard contexts (invariants) for Q, if the standard verification condition for that program and those contexts is valid, then those standard contexts indeed hold for that program; in notation: if $\models F_Q^{vc}$, then $\models$ `std-invs(`Q`)`. By the definition (page 58), $\models$ `std-invs(`Q`)` if $\models$ `std-invs(`P, Q`)` for all procedures $P \in Q$. Further, $\models$ `std-invs(`P, Q`)` if $\models$ `std-invs(`k, P, Q`)` for all standard contexts k of P. We therefore prove the standard soundness theorem as follows.

Theorem 1 *(Standard Soundness) If* $\models F^{vc}_Q$*, then* $\models$ `std-invs`(k, P, Q) *for all contexts* k *of all procedures* $P \in Q$*.*

Proof: Pick any procedure $P \in Q$ and any context k of P:

$$\begin{array}{l} P \equiv \texttt{proc } I\texttt{(}I^*\texttt{)}\ D^*\ \{N^+\} \\ \qquad \texttt{std-invariants}\ (F^{in}\ F^{out}\ T^*\ K^*)^*. \end{array}$$

Since $\models F^{vc}_Q$ and $F^{vc}_Q = \bigwedge_{P \in Q} F^{vc}_{P,Q}$, by the definition of conjunction (Figure 4-8), we get $\models F^{vc}_{P,Q}$. Further, $F^{vc}_{P,Q} = \bigwedge_{k \in \{1 \ldots n\}} F^{vc}_{k,P,Q}$, for contexts$(P) = \{1 \ldots n\}$, and therefore $\models F^{vc}_{k,P,Q}$, i.e., the standard verification condition for the context k of P holds. From the algorithm for $F^{vc}_{k,P,Q}$ and Std (Figure 4-9), we have:

$$\models \forall x^*.\ \mathrm{subst}(F^{in}_k, s^0) \Rightarrow \mathrm{Std}(\text{start-label}(P'), s^0, \{\}), \tag{5.5}$$

where $s^0 = \{I^* \mapsto x^*\}$ for all $I \in \mathrm{vars}(P)$ and s^0 is the starting symbolic state for the symbolic execution of $P' = \text{merge-invariants}(k, P)$. The symbolic execution uses a symbolic environment e that maps program variables to logic expressions representing those variables.

We need to show that for all partial executions $\langle P^0, m^0, a\rangle \overset{+}{\dashrightarrow} \langle L, m, a\rangle$ of P for which $m^0, a \models F^{in}_k$, the following is satisfied:

- for all L'`:inv` F from T^*_k, if $L \equiv L'$, then $m, a \models F$; and
- for all L'`:ret` from P, if $L \equiv L'$, then $m, a \models F^{out}_k$.

The proof is by induction on the length of the partial execution of P. We prove the induction using an induction hypothesis that we call the standard induction hypothesis (StdIH). The StdIH relates a configuration $\langle L, m, a\rangle$ of the concrete execution of P to the parameters of the symbolic execution $\mathrm{Std}(L', s, i)$ of P'. We first informally describe a relationship between each program point L of the concrete execution and a corresponding program point L' of the symbolic execution. We then formally state the StdIH and prove the base case and the induction step. The proof proceeds by an induction on the structure of the symbolic execution of P' corresponding to the concrete execution of P.

The correspondence between the symbolic execution of P' and the concrete execution of P is as follows. Each node (with label) L' from P' has a unique corresponding node (with label) L from P. The nodes in P' are obtained by adding the standard invariants to the nodes from P. Each node from P' that is not an invariant corresponds to the appropriate original node from P. Each standard invariant L`:inv` F from T^*_k is added to P' in front of the node with label L, with a proper label renaming. We say that an invariant L`:inv` F corresponds to the node with label L from P. (The merging of the invariants only adds new nodes, and the program variables do not change: $\mathrm{vars}(P) = \mathrm{vars}(P')$.) Conversely, each node from P has one or more corresponding nodes from P': the copy of the original node and, in general, a set of invariant nodes[1] that precede the copy of the original node. Therefore, each concrete

[1] In practice, there is at most one standard invariant for any node.

execution of a node from P has a corresponding sequence in the symbolic execution of P'—this sequence consists of the symbolic execution of the corresponding nodes from P'.

5.1.1 Standard Induction Hypothesis

The standard induction hypothesis (StdIH) relates a configuration $\langle L, m, a\rangle$ of the partial execution $\langle P^0, m^0, a\rangle \overset{+}{\dashrightarrow} \langle L, m, a\rangle$ of procedure P to the parameters of the symbolic execution $\mathrm{Std}(L', s, i)$ of P' started with the symbolic state s^0. Formally, the StdIH is a relation with ten arguments $\mathrm{StdIH}(L, m, a, L', s, i, \tau, P, m^0, s^0)$, where τ is a mapping from logic variables to values. We will abbreviate the StdIH to seven arguments $\mathrm{StdIH}(L, m, a, L', s, i, \tau)$, because the procedure and the starting states do not change for a fixed procedure activation. We define that the StdIH holds if the following is satisfied:

StdIH1. the standard verification condition is valid: $\models \tau(\mathrm{Std}(L', s, i))$, and

StdIH2. the symbolic state and the concrete state coincide for all $I \in \mathrm{vars}(P)$: $\models \tau(s(I)) = m(a(I))$, and

StdIH3. the substitution τ is correct with respect to i: either

StdIH3.1. $i = \{\}$ and for the initial symbolic state $s^0 = \{I^* \mapsto x^*\}$:

StdIH3.1.1. all logic variables are in the substitution, i.e., $\tau \supseteq \{x^* \mapsto Z^*\}$, and

StdIH3.1.2. $\models \tau(s^0(I)) = m^0(a(I))$ for all $I \in \mathrm{vars}(P)$; or

StdIH3.2. $i = i_1 \cup \{\langle L'', s'\rangle\}$, where $s' = \{I^* \mapsto x^*\}$ for some x^*, and

StdIH3.2.1. $P'(L'') \equiv L''\texttt{:inv}\ F$, and

StdIH3.2.2. $\tau = \tau_1 \cup \{x^* \mapsto Z^*\}$, and

StdIH3.2.3. x^* are fresh variables, i.e., for all x from x^*, $x \notin \tau_1 \cup i_1$, and

StdIH3.2.4. $\models \tau_1(\forall x^*.\ \mathrm{subst}(F, s') \Rightarrow \mathrm{Std}(L'' +_{P'} 1, s', i))$, and

StdIH3.2.5. τ_1 is correct with respect to i_1, as defined by StdIH3.

We now prove that for all $\langle L, m, a\rangle$ in $\langle P^0, m^0, a\rangle \overset{+}{\dashrightarrow} \langle L, m, a\rangle$ of P, where $m^0, a \models F_k^{in}$, and for all $L' \in P'$ corresponding to $L \in P$, there exist a symbolic state s, a set of symbolically executed invariants i, and a substitution τ such that $\mathrm{StdIH}(L, m, a, L', s, i, \tau)$ holds. We also show that this implies that the standard invariants and the output context of the context k hold:

- if $P'(L') \equiv L'\texttt{:inv}\ F$, then $m, a \models F$; and
- if $P'(L') \equiv L'\texttt{:ret}$, then $m, a \models F_k^{out}$.

5.1.2 Standard Base Case

The initial configuration for the concrete execution of P is $\langle P^0, m^0, a\rangle$, where $m^0, a \models F_k^{in}$. The symbolic execution starts with $L' \equiv$ start-label(P'), $s = s^0 = \{I^* \mapsto x^*\}$, where x^* are fresh variables, and $i = \{\}$. We need to show that there exists a substitution τ such that StdIH$(P^0, m^0, a, L', s^0, \{\}, \tau)$ holds. Let $\tau = \bigcup\{s(I) \mapsto m^0(a(I))\}$, where $\bigcup$ ranges over all $I \in$ vars(P), i.e., $\tau = \{x^* \mapsto m^0(a(I^*))\}$. This choice immediately implies StdIH2 and StdIH3 (part StdIH3.1). We next prove that StdIH1 also holds.

From 5.5, we have $\models \forall x^*.\ \text{subst}(F_k^{in}, s^0) \Rightarrow \text{Std}(L', s^0, \{\})$. Using the definition of universal quantification (Figure 4-8), we obtain $\models \tau(\text{subst}(F_k^{in}, s^0) \Rightarrow \text{Std}(L', s^0, \{\}))$. By the definition of implication, this simplifies to: if $\models \tau(\text{subst}(F_k^{in}, s^0))$, then $\models \tau(\text{Std}(L', s^0, \{\}))$. Since StdIH2 holds, we can apply the Standard Congruence Lemma, implication 5.1, and from $m^0, a \models F_k^{in}$, we have $\models \tau(\text{subst}(F_k^{in}, s^0))$. Therefore, $\models \tau(\text{Std}(L', s^0, \{\}))$, i.e., $\models \tau(\text{Std}(L', s, i))$, which is StdIH1.

5.1.3 Standard Induction Step

The concrete execution of the node at label L in P has a corresponding sequence in the symbolic execution of P'. The node at L' in P' corresponds to the node at L in P. We do a case analysis of the last node, at L', executed during the symbolic execution of P'. We show that if the StdIH holds before the last node is executed, then the StdIH also holds after the node is executed.

N1. The last node is an assignment node: $P'(L') \equiv L'$:I=E. Before this node is executed, from the induction hypothesis, there exist s, i, and τ such that StdIH$(L, m, a, L', s, i, \tau)$ holds. When the node is executed, the concrete execution makes a transition $\langle L, m, a\rangle \rightarrow \langle L +_P 1, m', a\rangle$, where $m' = m[a(I) \mapsto m(a(E))]$. The symbolic execution continues at the next node Std$(L' +_{P'} 1, s', i)$ with the new symbolic state $s' = s[I \mapsto \text{subst}(\text{translate}(E, e), s)]$. We need to show that StdIH holds in the new states. We use the same substitution τ to prove that StdIH$(L +_P 1, m', a, L' +_{P'} 1, s', i, \tau)$ holds.

StdIH3 holds in the new states because StdIH3 holds in the previous states, and τ and i do not change. That StdIH1 holds in the new states, namely $\models \tau(\text{Std}(L' +_{P'} 1, s', i))$, is also easy to prove: it follows from StdIH1 of the induction hypothesis, namely $\models \tau(\text{Std}(L', s, i))$, because τ does not change and the symbolic executions before and after the assignment node generate the same verification condition. To prove StdIH2 in the new states, namely $\models \tau(s'(I')) = m'(a(I'))$ for all $I' \in$ vars(P), we analyze two cases.

First, for all variables I' different than I, the symbolic state and the memory do not change, i.e., for all $I' \not\equiv I$, $s'(I') \equiv s(I)$ and $m'(a(I')) \equiv m(a(I))$. Therefore, for those variables, $\models \tau(s'(I')) = m'(a(I'))$ follows from StdIH2 of the induction hypothesis. Second, for $I' \equiv I$, to show $\models \tau(s'(I')) = m'(a(I'))$, we need to show $\models \tau(\text{subst}(\text{translate}(E, e), s)) = m(a(E))$. The equality holds by Lemma 3 because $\models \tau(s(I)) = m(a(I))$ for all $I \in$ vars(P), by StdIH2 of the induction hypothesis. Therefore, StdIH2 also holds in the new states for all variables.

N2. The last node is a conditional branch node: $P'(L') \equiv L'\texttt{:br(}E\texttt{)}L''$. There are two paths from this node, and the concrete execution takes only one of them depending on the value of the branch condition. However, the symbolic execution takes both paths, and from StdIH1 of the induction hypothesis, we have:

$$\models \tau(\texttt{(}G \Rightarrow \mathrm{Std}(L'', s, i)\texttt{)} \wedge \texttt{(}\neg G \Rightarrow \mathrm{Std}(L' +_{P'} 1, s, i)\texttt{)}), \tag{5.6}$$

where $G = \text{translate-branch}(E, s, e)$.

We next show that StdIH holds after the branch node when the branch is taken during the concrete execution; the case when the branch is not taken is analogous. The branch is taken, and the concrete execution makes a transition $\langle L, m, a\rangle \rightarrow \langle L'', m, a\rangle$, if $m(a(E)) \not\equiv \texttt{0}$. We use the same substitution τ from the induction hypothesis to show that StdIH holds after the branch is taken. Since $m(a(E)) \not\equiv \texttt{0}$ and, by StdIH2 of the induction hypothesis, $\models \tau(s(I)) = m(a(I))$ for all $I \in \mathrm{vars}(P)$, we have, by Lemma 4, $\models \tau(G)$.

From 5.6 and the definition of conjunction (Figure 4-8), we obtain that both $\models \tau(G \Rightarrow \mathrm{Std}(L'', s, i))$ and $\models \tau(\neg G \Rightarrow \mathrm{Std}(L' +_{P'} 1, s, i))$. From the former we further obtain: if $\models \tau(G)$, then $\models \tau(\mathrm{Std}(L'', s, i))$. Therefore, when the branch is taken, then $\models \tau(\mathrm{Std}(L'', s, i))$ which is StdIH1 in the new states. StdIH2 and StdIH3 in the new states trivially follow from the induction hypothesis because m, s, i, and τ remain the same.

N3. The last node is a return node: $P'(L') \equiv L'\texttt{:ret}$. It is the final node in the concrete execution of a procedure activation, and therefore we do not show that StdIH holds after the return node. We still need to show that the standard output context holds at the return node. From StdIH1 of the induction hypothesis, there exists τ such that $\models \tau(\mathrm{Std}(L', s, i))$, i.e., $\models \tau(\mathrm{subst}(F_k^{out}, s))$. Since StdIH2 of the induction hypothesis holds, by the Congruence Lemma 5.2, we obtain $m, a \models F_k^{out}$.

N4. The last node is a procedure call node: $P'(L') \equiv L'\texttt{:}I\texttt{(}E^*\texttt{)}$. Let the callee procedure be $P'' \equiv \texttt{proc } I\texttt{(}I^*\texttt{) } D^* \ \{N^+\}$. The concrete execution makes a transition to the first node of P'', allocating memory for the parameters and local variables of P'':

$$\langle m^{in}, a^{in}, p^{in}\rangle = \text{alloc-locals}(\text{alloc-params}(\langle m, a, p\rangle, I^*, m(a(E^*))), D^*).$$

When and if the execution of P'' returns, the concrete execution continues from $\langle P(L +_P 1), m', a\rangle$, where m' is the memory after the call. For all non-global variables of P, the values remain the same: $m'(a(I)) = m(a(I))$ for all $I \in \mathrm{locals}(P) \cup \mathrm{params}(P)$.

The symbolic execution continues, after the return node, with the new symbolic state $s' = s[I^* \mapsto x^*]$ for all $I^* \in \mathrm{globals}(P)$, where x^* are fresh logic variables.

From StdIH1 of the induction hypothesis, there is a substitution τ such that:

$$\models \tau(\text{subst}(F^{in}, \text{set-params}(I, G^*, s))) \tag{5.7}$$

and

$$\models \tau(\forall x^*.\ \text{subst}(F^{out}, s') \Rightarrow \text{Std}(L' +_{P'} 1, s', i)), \tag{5.8}$$

where $G^* = \text{translate-call}(E^*, s, e)$, and $\langle F^{in}, F^{out}\rangle$ is one of the contexts of procedure P''. We use s^{in} to denote the symbolic state for the formula F^{in}: $s^{in} = \text{set-params}(I, G^*, s) = s[I^* \mapsto G^*]$ for parameters I^* of procedure I/P''. We need to show that the StdIH holds after the call. We use the substitution $\tau' = \tau \cup \{x^* \mapsto m'(a(I^*))\}$ to show that $\text{StdIH}(L +_P 1, m', a, L' +_{P'} 1, s', i, \tau')$ holds.

StdIH3 is easy to prove: from StdIH3 of the induction hypothesis, τ is correct with respect to i. After the symbolic execution of the call node, i remains the same and $\tau \subseteq \tau'$. Therefore, τ' is correct with respect to i. To show StdIH2, we analyze two cases. First, for all non-global variables $I \in \text{locals}(P) \cup \text{params}(P)$, the following holds: $\models \tau'(s'(I)) = \tau'(s(I)) = \tau(s'(I)) = m(a(I)) = m'(a(I))$. Second, for all global variables $I \in \text{globals}(P)$, from the choice of τ' immediately follows: $\models \tau'(s'(I)) = m'(a(I))$. Therefore, $\models \tau'(s'(I)) = m'(a(I))$ for all $I \in \text{vars}(P)$. We still need to prove that StdIH1 holds in the new states.

We first show that $\models \tau(s^{in}(I)) = m^{in}(a^{in}(I))$ for all variables $I \in \text{globals}(P'') \cup \text{params}(P'')$ that can occur in F^{in}. We analyze two cases: $I \in \text{globals}(P'')$ and $I \in \text{params}(P'')$. First, for all global variables I, $s^{in}(I) \equiv s(I)$ and $m^{in}(a^{in}(I)) = m(a(I))$. From StdIH2 of the induction hypothesis, $\models \tau(s(I)) = m(a(I))$ for all $I \in \text{vars}(P)$. Since P and P'' have the same global variables, $\models \tau(s^{in}(I)) = m^{in}(a^{in}(I))$ for all those global variables. Second, for each parameter $I^{(n)}$ from the sequence I^*, $s^{in}(I^{(n)}) \equiv G^{(n)}$, where $G^{(n)}$ is the respective expression from the sequence $G^* = \text{translate-call}(E^*, s, e)$. By Lemma 5, $\models \tau(G^{(n)}) = m(a(E^{(n)}))$, since $\models \tau(s(I)) = m(a(I))$ for all $I \in \text{vars}(P)$, by StdIH2 of the induction hypothesis. Further, $m^{in}(a^{in}(I^{(n)})) \equiv m(a(E^{(n)}))$ by the definition of the alloc-params function (Section 4.1.2). Therefore, $\models \tau(s^{in}(I^{(n)})) = m^{in}(a^{in}(I^{(n)}))$ for all parameters of P'', and thus $\models \tau(s^{in}(I)) = m^{in}(a^{in}(I))$ for all variables I that can occur in F^{in}.

We next show that $m', a \models F^{out}$. From 5.7, we have $\models \tau(\text{subst}(F^{in}, s^{in}))$. Since $\models \tau(s^{in}(I)) = m^{in}(a^{in}(I))$ for all variables I that can occur in F^{in}, by the Congruence Lemma 5.2, we obtain $m^{in}, a^{in} \models F^{in}$. This means that the input context F^{in} holds at the beginning of P''. For every context k'' of every procedure P'' in Q, if the input context holds at the beginning of an activation of P'', then the output context holds at the end of that activation. (Note that it appears that we assume the actual statement that we try to establish, namely $\models$ `std-invs(`Q`)`. We show later how to correct this apparent error.) Therefore, $m^{out}, a^{out} \models F^{out}$, where m^{out} is the memory at the end of the activation of P'', and $a^{out} = a^{in}$ is the environment for the particular activation of P''. Memory $m^{out} = m'$ because the concrete execution of the return node in P'' does not change the memory. Additionally, the environments a^{in} for P'' and a for P map the global variables to the same addresses. Since F^{out} can contain only global variables, we have $m', a \models F^{out}$.

Finally, we show that $\models \tau'(\mathrm{Std}(L' +_{P'} 1, s', i))$. From 5.8 and the choice of τ', we have $\models \tau'(\mathrm{subst}(F^{out}, s') \Rightarrow \mathrm{Std}(L' +_{P'} 1, s', i))$. We have already proven that StdIH2 holds in the new states: $\models \tau'(s'(I)) = m'(a(I))$ for all $I \in \mathrm{vars}(P)$. From $m', a \models F^{out}$, we obtain, by the Congruence Lemma 5.1, that $\models \tau'(\mathrm{subst}(F^{out}, s'))$. Therefore, $\models \tau'(\mathrm{Std}(L' +_{P'} 1, s', i))$, which is StdIH1 in the new states. This concludes the proof that the execution of any call node preserves the StdIH.

We next show how to correct the apparent error that we used $\models$ `std-invs(`Q`)` to prove $\models$ `std-invs(`Q`)`. More precisely, while proving $\models$ `std-invs(`k, P, Q`)`, we assumed, at call sites in P, that $\models$ `std-invs(`k'', P'', Q`)` for all contexts k'' of all procedures P''. If the procedure P is recursive, we cannot make such an assumption. The correct proof of call nodes requires, besides the induction on the length of the partial execution of P, an additional induction on the height of the procedure call tree. A procedure call tree is a directed tree whose nodes represent activations of procedures called during a (concrete) program execution, and whose edges represent call relationships—there is an edge from one activation to another if the former calls the latter. The root of the whole call tree for a program execution is the initial activation of the starting procedure of the program. Each activation is also the root of a subtree of calls made starting from that activation.

We next describe only informally how the induction on the height of the procedure call tree would proceed. The induction hypothesis would state that $\models$ `std-invs(`k, P, Q`)` for all activations (of all contexts k of all procedures P) whose subtrees have height n. The base case considers activations which make no calls. (The proof for this case is by induction on the length of the partial execution without call nodes.) The induction step assumes that $\models$ `std-invs(`k, P, Q`)` for all activations whose subtrees have height at most n, and derives that $\models$ `std-invs(`k, P, Q`)` for all activations whose subtrees have height at most $n + 1$. (The proof for this case is again by induction on the length of the partial execution, and the hypothesis for height at most n is used for call nodes.) This induction would correct the proof.

Observe that this induction proceeds from the leaves of the tree toward the root. Therefore, if a tree has finite height, then clearly $\models$ `std-invs(`k, P, Q`)` for all activations. However, even if a tree has infinite height, still $\models$ `std-invs(`k, P, Q`)` for all activations. The reason is that we require *partial correctness*—the output contexts should hold only *if* the execution reaches a return node. When a subtree of the call tree has infinite height, it means that the activation of the root of that subtree never terminates. Since the execution of that activation does not reach a return node, any result for the output context is allowed. For example, consider a parameterless procedure that has only a recursive call to the same procedure and a return node. For this procedure, the context with $F^{in} \equiv$ `true` and $F^{out} \equiv$ `false` holds because no execution can reach the return node. Otherwise, the output context `false` does not hold for any concrete state, and would not be correct/provable if an execution could reach the return node.

N5. The last node is an invariant: $P'(L') \equiv L'$`:inv` F. There are two cases depending on whether the invariant has been already symbolically executed or not:

N5.1. The invariant L'`:inv` F is executed for the first time, i.e., $\langle L', s'\rangle \notin i$ for any s'. From StdIH1 of the induction hypothesis, there is a substitution τ such that:

$$\models \tau(\text{subst}(F, s)) \tag{5.9}$$

and

$$\models \tau(\forall x^*.\ \text{subst}(F, s') \Rightarrow \text{Std}(L' +_{P'} 1, s', i')), \tag{5.10}$$

where $s' = \{I^* \mapsto x^*\}$ is a fresh symbolic state with fresh logic variables x^* for all $I \in \text{vars}(P)$ and $i' = \langle L', s'\rangle \cup i$.

From StdIH2 of the induction hypothesis, $\models \tau(s(I)) = m(a(I))$ for all $I \in \text{vars}(P)$. We can therefore apply the Congruence Lemma 5.2, and from 5.9, we obtain $m, a \models F$, which is one of the requirements for $\models$ `std-invs(`k, P, Q`)`. We need additionally to show that StdIH holds after the symbolic execution of the invariant. (There is no concrete execution of the invariants—the invariants only correspond to certain program points and describe the program state at those points.) We use the substitution $\tau' = \tau \cup \{x^* \mapsto m'(a(I^*))\}$ to show that $\text{StdIH}(L, m, a, L' +_{P'} 1, s', i', \tau')$ holds after the invariant.

StdIH2 is trivial to prove; from the choice of τ': $\models \tau'(s'(I)) = m(a(I))$ for all $I \in \text{vars}(P)$. Since StdIH2 holds, and also $m, a \models F$, by the Congruence Lemma 5.1, we have $\models \tau'(\text{subst}(F, s'))$. Further, from 5.10 and the choice of τ', we have:

$$\models \tau'(\text{subst}(F, s') \Rightarrow \text{Std}(L' +_{P'} 1, s', i')), \tag{5.11}$$

which means that $\models \tau'(\text{Std}(L' +_{P'} 1, s', i'))$ if $\models \tau'(\text{subst}(F, s'))$. Therefore, $\models \tau'(\text{Std}(L' +_{P'} 1, s', i'))$, which is StdIH1. We need still to prove StdIH3, i.e., that τ' is correct with respect to i'. We prove that StdIH3.2 holds. StdIH3.2.1–StdIH3.2.3 follow from the choice of τ'. StdIH3.2.4 holds because of 5.10 (τ_1 is τ). StdIH3.2.5 follows from StdIH3 of the induction hypothesis.

N5.2. The invariant L'`:inv` F has been previously executed, i.e., $\langle L', s'\rangle \in i$ for some symbolic state s'. From StdIH1 of the induction hypothesis, there is a substitution τ such that $\models \tau(\text{subst}(F, s))$. From StdIH2 of the induction hypothesis, $\models \tau(s(I)) = m(a(I))$ for all $I \in \text{vars}(P)$. We can therefore apply the Congruence Lemma 5.2 to obtain $m, a \models F$, which is one of the requirements for $\models$ `std-invs(`k, P, Q`)`. We need additionally to show that StdIH holds after the invariant. We show that there exists τ' such that StdIH$(L, m, a, L' +_{P'} 1, s', i, \tau')$ holds.

From $\langle L', s' \rangle \in i$, we have that $i = \{\langle L', s' \rangle\} \cup i_1$, where $s' = \{I^* \mapsto x^*\}$ for all $I \in \text{vars}(P)$. From StdIH3.2.2 of the induction hypothesis, we know that $\tau = \tau_1 \cup \{x^* \mapsto Z^*\}$ for some τ_1 such that by StdIH3.2.3, none of x from x^* is in τ_1. Therefore, we can use $\tau' = \tau_1 \cup \{x^* \mapsto m(a(I^*))\}$, and we prove that StdIH holds for that τ'.

StdIH2, namely $\models \tau'(s'(I)) = m(a(I))$ for all $I \in \text{vars}(P)$, follows directly from the choice of τ'. StdIH3 follows from StdIH3 of the induction hypothesis. We need still to show that StdIH1 holds: $\models \tau'(\text{Std}(L' +_{P'} 1, s', i))$.

From StdIH3.2.4 of the induction hypothesis, we have:

$$\models \tau_1(\forall x^*. \,\text{subst}(F, s') \Rightarrow \text{Std}(L' +_{P'} 1, s', i)). \tag{5.12}$$

Further, by the definition of universal quantification, we get:

$$\models \tau'(\text{subst}(F, s') \Rightarrow \text{Std}(L' +_{P'} 1, s', i)). \tag{5.13}$$

As we have already shown that StdIH2 holds and $m, a \models F$, by the Congruence Lemma 5.1, we obtain $\models \tau'(\text{subst}(F, s'))$. Therefore, from 5.13, we finally have that $\models \tau'(\text{Std}(L' +_{P'} 1, s', i))$.

5.2 Soundness of Simulation Verification-Condition Generator

In this section we prove that the simulation verification-condition generator (presented in Section 4.4) is sound: for every pair of programs Q_1 and Q_2 and every set of simulation contexts for those programs, if the simulation verification condition for those programs and those contexts is valid, then Q_1 simulates Q_2; in notation: if $\models F^{vc}_{Q_1,Q_2}$, then $Q_1 \rhd Q_2$. (More precisely, we also need the standard verification conditions for programs Q_1 and Q_2 to be valid: if $\models F^{vc}_{Q_1,Q_2}$ and $\models F^{vc}_{Q_1}$ and $\models F^{vc}_{Q_2}$, then $Q_1 \rhd Q_2$.) By the definition (page 65), $Q_1 \rhd Q_2$ if:

- $P_1, Q_1 \rhd P_2, Q_2$ for all pairs of procedures $P_1 \in Q_1$ and $P_2 \in Q_2$ for which there are simulation contexts, and

- one of the simulation contexts for the starting procedures for programs Q_1 and Q_2 requires that the two programs generate the same output given that they start with the same input.

Further, $P_1, Q_1 \rhd P_2, Q_2$ if $P_1, Q_1 \rhd_k P_2, Q_2$ for all contexts $k \in \text{sim-contexts}(P_1, P_2)$. We therefore prove the simulation soundness theorem as follows.

Theorem 2 *(Simulation Soundness) If* $\models F^{vc}_{Q_1,Q_2}$ *and* $\models F^{vc}_{Q_1}$ *and* $\models F^{vc}_{Q_2}$*, then* $P_1, Q_1 \rhd_k P_2, Q_2$ *for all simulation contexts* k *of all pairs of procedures* $P_1 \in Q_1$ *and* $P_2 \in Q_2$*.*

Proof: Pick any pair of procedures $P_1 \in Q_1$ and $P_2 \in Q_2$, for which there is a simulation context, and pick any simulation context k of P_1 and P_2:

$$\begin{array}{l} P_1 \equiv \texttt{proc}\ I_1\texttt{(}I_1^*\texttt{)}\ D_1^*\ \{N_1^+\} \\ P_2 \equiv \texttt{proc}\ I_2\texttt{(}I_2^*\texttt{)}\ D_2^*\ \{N_2^+\} \\ \qquad \texttt{sim-invariants}\ (J^{in}\ J^{out}\ S^*\ K^*\ Z^1\ Z^2\ G^*\ A^*)^*. \end{array}$$

Since $\models F^{vc}_{Q_1,Q_2}$ and $F^{vc}_{Q_1,Q_2} = \bigwedge_{P_1 \in Q_1, P_2 \in Q_2} F^{vc}_{P_1,Q_1,P_2,Q_2}$, by the definition of conjunction (Figure 4-8), we get $\models F^{vc}_{P_1,Q_1,P_2,Q_2}$. Further, by the definition, $F^{vc}_{P_1,Q_1,P_2,Q_2} = \bigwedge_{k \in \{1 \ldots n\}} F^{vc}_{k,P_1,Q_1,P_2,Q_2}$, for sim-contexts$(P_1, P_2) = \{1 \ldots n\}$, and thus $\models F^{vc}_{k,P_1,Q_1,P_2,Q_2}$, i.e., the simulation verification condition for the context k of P_1 and P_2 holds. From the algorithm for $F^{vc}_{k,P_1,Q_1,P_2,Q_2}$ and Sim (Figure 4-11), we have:

$$\begin{array}{ll} \models \forall x^*.\ \text{subst-sim}(J^{in}_k, s^0_1, s^0_2) \Rightarrow & \\ \qquad \text{Sim}(\text{start-label}(P'_1), s^0_1, \text{start-label}(P'_2), s^0_2, \{\}, t^0) \wedge & \qquad (5.14) \\ \qquad \text{subst}(\text{in-context}(P_1, k_1), s^0_1) \wedge \text{subst}(\text{in-context}(P_2, k_2), s^0_2), & \end{array}$$

where: $s^0_1 = \{I^*_1 \mapsto x^*_1\}$ for all $I_1 \in \text{vars}(P_1)$, $s^0_2 = \{I^*_2 \mapsto G^*\}$ for all $I_2 \in \text{vars}(P_2)$, s^0_1 and s^0_2 are the starting symbolic states for the symbolic executions of procedures $\langle P'_1, P'_2 \rangle = \text{merge-sim-invariants}(k, P_1, P_2)$, and $t^0 = \text{action-tree}(A^*_k, P'_1, P'_2)$ is the action tree for the interleaving of the symbolic executions of P'_1 and P'_2. The symbolic executions use symbolic environments $e_{1/2}$ that map program variables from procedures $P_{1/2}$ to logic expressions representing those variables.[2] We use a variable name with an index to represent which program the variable is from.

The starting symbolic expressions G^* for s^0_2 are created in the following way:

- for each $I_2 \in \text{vars}(P_2) - \text{locals}(P_2)$:
  - if $\langle I_1, I_2 \rangle \in \text{var-pairs}(J^{in}_k)$ for some I_1, then I_2 is mapped to $s_1(I_1)$; otherwise,
  - I_2 is mapped to a fresh variable x_2, and the sequence of all such fresh variables, x^*_2, concatenated to x^*_1 gives x^*; and
- for each $I_2 \in \text{locals}(P_2)$, the symbolic expression for I_2 is obtained by substituting x^* for the appropriate program variables in expressions G^*_k provided by the compiler transformation.

[2] In general, $e_{1/2}$ maps each $I_l \in \text{locals}(P_{1/2}) \cup \text{params}(P_{1/2})$ to logic expression $\texttt{loc}_{1/2}\texttt{(}I_l\texttt{)}$ and each $I_g \in \text{globals}(P_{1/2})$ to logic expression $\texttt{glob}_{1/2}\texttt{(}I_g\texttt{)}$. For the brevity of the proof presentation, we use logic expressions without the special constructors H. Therefore, $e_{1/2}$ is the identity: it maps each variable $I_{1/2} \in \text{vars}(P_{1/2})$ to itself. We assume that the variable names in two procedures are different: $\text{vars}(P_1) \cap \text{vars}(P_2) = \{\}$.

The compiler transformation also provides indices for standard contexts of procedures: k_1 of P_1 and k_2 of P_2. By assumption, the standard verification conditions for programs Q_1 and Q_2 are valid: $\models F^{vc}_{Q_1}$ and $\models F^{vc}_{Q_2}$. Therefore, by Theorem 1, all standard contexts of all procedures from programs Q_1 and Q_2 hold. In particular, contexts k_1 of P_1 and k_2 of P_2 hold:

$$\models \texttt{std-invs(}k_1, P_1, Q_1\texttt{)} \tag{5.15}$$

and

$$\models \texttt{std-invs(}k_2, P_2, Q_2\texttt{)}. \tag{5.16}$$

We need to show that for all partial executions $\langle P_1^0, m_1^0, a_1\rangle \overset{+}{\dashrightarrow} \langle L_1, m_1, a_1\rangle$ of P_1, there exists a partial execution $\langle P_2^0, m_2^0, a_2\rangle \overset{+}{\dashrightarrow} \langle L_2, m_2, a_2\rangle$ of P_2 such that if $m_1^0, a_1, m_2^0, a_2 \models J_k^{in}$, then the following is satisfied:

- for all L_1',L_2':sim-inv J' from S_k^*, if $L_1 \equiv L_1'$, then there exists L_1,L_2'':sim-inv J from set-sim-inv(L_1, S_k^*) such that $L_2 \equiv L_2''$ and $m_1, a_1, m_2, a_2 \models J$; and
- if $P_1(L_1) \equiv L_1$:ret, then $P_2(L_2) \equiv L_2$:ret and $m_1, a_1, m_2, a_2 \models J_k^{out}$; and
- if the partial execution of P_1 does not terminate, then the partial execution of P_2 also does not terminate.

The proof is by induction on the length of the partial execution of P_1. We prove the induction using an induction hypothesis that we call the simulation induction hypothesis (SimIH). The SimIH relates configurations $\langle L_1, m_1, a_1\rangle$ and $\langle L_2, m_2, a_2\rangle$ of the concrete executions of P_1 and P_2 to the parameters of the symbolic executions $\text{Sim}(L_1', s_1, L_2', s_2, i, t)$ of P_1' and P_2'. Similar to the standard induction hypothesis, the simulation induction hypothesis relates the program points $L_{1/2}$ of the concrete executions to the program points $L_{1/2}'$ of the symbolic executions. Additionally, the SimIH includes an action tree t that guides the symbolic executions. We first describe the correspondence between the actions of the tree and the concrete executions. We next describe how the SimIH uses a substitution τ to relate the concrete states $m_{1/2}, a_{1/2}$ to the symbolic states $s_{1/2}$. We then present a lemma that shows certain substitutions to be well-defined for symbolic states that are related by a formula J. Finally, we state the SimIH and prove the base case and the induction step.

The correspondence between the nodes of $P_{1/2}'$ and the nodes of $P_{1/2}$ is as follows. Each node (with label) L' from either P_1' or P_2' has a unique corresponding node (with label) L from the respective P_1 or P_2. The nodes in $P_{1/2}'$ are obtained by merging the halves of simulation invariants to the nodes from $P_{1/2}$. Each node from $P_{1/2}'$ that is not a half of an invariant corresponds to the appropriate original node from $P_{1/2}$. For each simulation invariant L_1,L_2:sim-inv J from S_k^*, the node L_1:sim-inv is added to P_1' in front of the node with label L_1 and the node L_2:sim-inv J,L_1 is added to P_2' in front of the node with label L_2, as explained in the initial phase of the SimVCG (page 67). The nodes L_1:sim-inv and L_2:sim-inv J,L_1 represent two halves of a simulation invariant.

Similar to the standard invariants, each node from $P_{1/2}$ has one or more corresponding nodes from $P'_{1/2}$: the copy of the original node and, in general, a set of halves of simulation invariants that precede the copy of the original node.[3] The soundness proof for the standard invariants is, conceptually, by an induction on the length of the concrete execution of the procedure. Technically, we do an induction on all possible symbolic executions (corresponding to the concrete execution) that implies the induction on the length of the concrete execution. Similarly, the soundness proof for the simulation invariants is, conceptually, by an induction on the length of the concrete execution of P_1. We actually do an induction on all possible symbolic executions of P'_1 and P'_2 corresponding to the concrete execution of P_1.

The action tree t guides the interleaving of the symbolic executions of P'_1 and P'_2. Each concrete execution of a node L_1 from P_1 has a corresponding sequence of actions from the tree. This sequence ends with the action that executes in P'_1 the copy of the node L_1; depending on the node, the action can be either $\texttt{execute}_1$ or `execute-both`. The sequence starts with the action that immediately follows the end of the previous sequence, except that the sequence for the first node starts with the first action of the tree. We show in the induction step that any sequence corresponding to the concrete execution of a node preserves the simulation induction hypothesis.

More precisely, we consider only correct sequences of actions, i.e., correct action trees. An action tree t is *correct* with respect to L'_1, L'_2, and i if the application $\mathrm{Sim}(L'_1, s_1, L'_2, s_2, i, t)$ does not generate an error. In general, an error occurs if there are not enough actions in the tree (i.e., t is empty when Sim performs $\mathrm{root}(t)$), or the root action is not allowed for the nodes L'_1 and/or L'_2 (e.g., `execute-both` for a return and a branch node). The SimVCG invokes the function Sim with the initial tree t^0 and we know that it generates a SimVC, and not an error, since we also know that the SimVC is valid. Thus, t^0 is correct for the respective starting labels. In the base case of the induction, we also use the tree t^0. If a tree is correct before some action, then the appropriate subtrees are correct for the executions after the action. In the induction step, we consider the same subtrees as the function Sim, and therefore the subtrees are correct for the respective executions. Whenever we write $\mathrm{Sim}(L'_1, s_1, L'_2, s_2, i, t)$ in the rest of the proof, we understand that it is for a correct t.

We next show the correspondence between the symbolic states $s_{1/2}$ and the concrete states $m_{1/2}, a_{1/2}$. Each of the symbolic states $s_{1/2} = \{I^*_{1/2} \mapsto G^*_{1/2}\}$ corresponds to a set of the concrete states $m_{1/2}, a_{1/2}$. We can specify one such correspondence with a substitution $\tau = \{x^* \mapsto Z^*\}$, where x^* are all logic variables in the expressions $G^*_{1/2}$. The SimIH uses τ such that the symbolic states and the concrete states coincide: $\models \tau(s_{1/2}(I_{1/2})) = m_{1/2}(a_{1/2}(I_{1/2}))$ for all $I_{1/2} \in \mathrm{vars}(P_{1/2})$. In general, there are symbolic and concrete states for which no such τ exists, even if $s_1 = \{I^*_1 \mapsto x^*_1\}$ for all $I_1 \in \mathrm{vars}(P_1)$ and some sequence x^*_1 of different logic variables. (If s_1 can be arbitrary, then there is no τ, e.g., for $s_1 = \{I_1 \mapsto x_1, I' \mapsto x_1\}$ and m_1, a_1 such that $m_1(a_1(I_1)) \neq m_1(a_1(I'))$.) We next show that a substitution τ exists for all states that satisfy certain conditions.

[3] Recall that in P'_1, there is at most one L_1:`sim-inv` before any original node because the simulation invariants with the same first label share that node.

Lemma 6 *(Well-Defined Substitution) Let J be any formula for two programs (a simulation invariant, a simulation input context, or a simulation output context), and let m_1, a_1, m_2, a_2 be any concrete states such that $m_1, a_1, m_2, a_2 \models J$. Let $s_1 = \{I_1^* \mapsto x_1^*\}$ for all $I_1 \in \mathrm{vars}(P_1)$ and some sequence x_1^* of different logic variables. Let $s_2 = \{I_2^* \mapsto G^*\}$ for all $I_1 \in \mathrm{vars}(P_1)$, where G^* are any expressions such that: for any variable I and any variable $I' \not\equiv I$ that* can *occur in J (all variables, non-local variables, or global variables, depending on J), $s_2(I)$ is a logic variable and $s_2(I) \not\equiv s_2(I')$. Let x_2^* be all logic variables that are images, in s_2, of variables that can occur in J. (Note that some variables from x_2^* may be identical to some variables from x_1^*.) Let $x^* = x_1^* \cup x_2^*$ be a sequence of all unique variables from x_1^* and x_2^*. If $s_1(I_1) \equiv s_2(I_2)$ for all $\langle I_1, I_2 \rangle \in \mathrm{var\text{-}pairs}(J)$, i.e., the symbolic states $s_{1/2}$ are related by (the pairs of variables in) J, then there exists a substitution $\tau = \{x_1^* \mapsto m_1(a_1(I_1^*))\} \cup \{x_2^* \mapsto m_2(a_2(I_2^*))\}$, denoted as $\{x^* \mapsto m_{1/2}(a_{1/2}(I_{1/2}^*))\}$, such that: $\models \tau(s_{1/2}(I_{1/2})) = m_{1/2}(a_{1/2}(I_{1/2}))$ for all $I_{1/2}$ that can occur in J.*

Proof: We give only an outline of the proof. The main result to show is that the substitution τ is well-defined: for each x from x^*, there exists a unique Z such that $\tau(x) = Z$. Let $\tau_1 = \{x_1^* \mapsto m_1(a_1(I_1^*))\}$ and $\tau_2 = \{x_2^* \mapsto m_2(a_2(I_2^*))\}$. We have that $\tau = \tau_1 \cup \tau_2$, and we need to show that $\tau_1(x) = \tau_2(x)$ for all variables x that are both in x_1 *and* x_2. It is easy to show that those logic variables are images, in $s_{1/2}$, of the program variables that appear in var-pairs(J). Further, it is easy to show, by the definition of conjunction, that $m_1(a_1(I_1)) = m_2(a_2(I_2))$ for all $\langle I_1, I_2 \rangle \in \mathrm{var\text{-}pairs}(J)$ follows from $m_1, a_1, m_2, a_2 \models J$. This concludes the proof of the lemma.

5.2.1 Simulation Induction Hypothesis

The simulation induction hypothesis (SimIH) relates configurations $\langle L_1, m_1, a_1 \rangle$ and $\langle L_2, m_2, a_2 \rangle$ of the partial executions $\langle P_1^0, m_1^0, a_1 \rangle \overset{+}{\dashrightarrow} \langle L_1, m_1, a_1 \rangle$ and $\langle P_2^0, m_2^0, a_2 \rangle \overset{+}{\dashrightarrow} \langle L_2, m_2, a_2 \rangle$ of procedures P_1 and P_2 to the parameters of the symbolic executions $\mathrm{Sim}(L_1', s_1, L_2', s_2, i, t)$ of P_1' and P_2' started with the symbolic states s_1^0 and s_2^0. Formally, the SimIH is a relation with many arguments:

$$\mathrm{SimIH}(L_1, m_1, a_1, L_2, m_2, a_2, L_1', s_1, L_2', s_2, i, t, \tau, P_1, m_1^0, s_1^0, P_2, m_2^0, s_2^0),$$

where τ is a mapping from logic variables to values. We will abbreviate the SimIH to $\mathrm{SimIH}(L_1, m_1, a_1, L_2, m_2, a_2, L_1', s_1, L_2', s_2, i, t, \tau)$, because the procedures and the starting states do not change for a fixed pair of procedure activations. We define that the SimIH holds if the following is satisfied:

SimIH1. the simulation verification condition holds: $\models \tau(\mathrm{Sim}(L_1', s_1, L_2', s_2, i, t))$, and

SimIH2. the symbolic states and the concrete states coincide:

SimIH2.1. for all $I_1 \in \mathrm{vars}(P_1)$: $\models \tau(s_1(I_1)) = m_1(a_1(I_1))$, and

SimIH2.2. for all $I_2 \in \mathrm{vars}(P_2)$: $\models \tau(s_2(I_2)) = m_2(a_2(I_2))$; and

SimIH3. the substitution τ is correct with respect to i: either

SimIH3.1. $i = \{\}$ and for the initial symbolic states $s_1^0 = \{I_1^* \mapsto x_1^*\}$ and $s_2^0 = \{I_2^* \mapsto G^*\}$, where x^* are all variables that occur in x_1^* or G^*:

SimIH3.1.1. all logic variables are in the substitution, i.e., $\tau \supseteq \{x^* \mapsto Z^*\}$, and

SimIH3.1.2. $\models \tau(s_1^0(I_1)) = m_1^0(a_1(I_1))$ for all $I_1 \in \text{vars}(P_1)$, and

SimIH3.1.3. $\models \tau(s_2^0(I_2)) = m_2^0(a_2(I_2))$ for all $I_2 \in \text{vars}(P_2)$; or

SimIH3.2. $i = i_1 \cup \{\langle\langle L_1'', L_2''\rangle, \langle s_1', s_2'\rangle, t'\rangle\}$, where $s_1' = \{I_1^* \mapsto x_1^*\}$ and $s_2' = \{I_2^* \mapsto x_2^*\}$ for some $x^* = x_1^* \cup x_2^*$, and

SimIH3.2.1. $P_1'(L_1'') \equiv L_1''$`:sim-inv` and $P_2'(L_2'') \equiv L_2''$`:sim-inv` J`,`L_1'', and

SimIH3.2.2. $\tau = \tau_1 \cup \{x^* \mapsto Z^*\}$, and

SimIH3.2.3. x^* are fresh variables, i.e., for all x from x^*, $x \notin \tau_1 \cup i_1$, and

SimIH3.2.4. $\models \tau_1(\forall x^*.\ \text{subst-sim}(J, s_1', s_2') \Rightarrow \text{Sim}(L_1'' +_{P_1'} 1, s_1', L_2'' +_{P_2'} 1, s_2', i, t'))$, and

SimIH3.2.5. τ_1 is correct with respect to i_1, as defined by SimIH3.

We now prove that for all configurations $\langle L_1, m_1, a_1\rangle$ in sequences $\langle P_1^0, m_1^0, a_1\rangle \overset{+}{\dashrightarrow} \langle L_1, m_1, a_1\rangle$ of P_1, there exists a configuration $\langle L_2, m_2, a_2\rangle$ in a sequence $\langle P_2^0, m_2^0, a_2\rangle \overset{+}{\dashrightarrow} \langle L_2, m_2, a_2\rangle$ of P_2, where $m_1^0, a_1, m_2^0, a_2 \models J_k^{in}$, such that for all $L_1' \in P_1'$ corresponding to $L_1 \in P_1$, there exists a label $L_2' \in P_2'$ corresponding to $L_2 \in P_2$, and there exist symbolic states s_1 and s_2, a set of symbolically executed simulation invariants i, an action tree t, and a substitution τ for which $\text{SimIH}(L_1, m_1, a_1, L_2, m_2, a_2, L_1', s_1, L_2', s_2, i, t, \tau)$ holds. We also show that this implies that the simulation invariants and the simulation output context of the context k hold, as well as that the termination simulation is satisfied:

- if $P_1'(L_1') \equiv L_1'$`:sim-inv`, then $P_2'(L_2') \equiv L_2'$`:sim-inv` J`,`L_1' and $m_1, a_1, m_2, a_2 \models J$; and
- if $P_1'(L_1') \equiv L_1'$`:ret`, then $P_2'(L_2') \equiv L_2'$`:ret` and $m_1, a_1, m_2, a_2 \models J_k^{out}$; and
- if the execution of P_1 does not terminate, then the execution of P_2 also does not terminate.

5.2.2 Simulation Base Case

The initial configurations for the concrete executions of P_1 and P_2 are $\langle P_1^0, m_1^0, a_1\rangle$ and $\langle P_2^0, m_2^0, a_2\rangle$, where $m_1^0, a_1, m_2^0, a_2 \models J_k^{in}$. (Recall that by the simulation requirement, the memory m_1^0 for the execution of P_1 is universally quantified, whereas the memory m_2^0 for the execution of P_2 is existentially quantified.) The symbolic executions start with $L_1' \equiv \text{start-label}(P_1')$, $s_1 = s_1^0 = \{I_1^* \mapsto x_1^*\}$, $L_2' \equiv \text{start-label}(P_2')$, $s_2 = s_2^0 = \{I_2^* \mapsto G^*\}$, $i = \{\}$, and $t = t^0$. Let x^* be all logic variables that occur in x_1^* or G^*. We need to show that there exists a substitution τ for all variables in x^* such that $\text{SimIH}(P_1^0, m_1^0, a_1, P_2^0, m_2^0, a_2, L_1', s_1^0, L_2', s_2^0, \{\}, t^0, \tau)$ holds.

By the Well-Defined Substitution Lemma, we know that there exists $\tau = \{x^* \mapsto m^0_{1/2}(a_{1/2}(I^*_{1/2}))\}$ such that $\models \tau(s^0_{1/2}(I_{1/2})) = m^0_{1/2}(a_{1/2}(I_{1/2}))$ for all $I_{1/2} \in \text{vars}(P_{1/2}) - \text{locals}(P_{1/2})$. We first show that for this τ, SimIH2 holds. SimIH3 (part SimIH3.1) then trivially follows from SimIH2. To conclude the proof that SimIH holds, we then show that SimIH1 holds. Finally, we show also that the standard invariants of standard contexts $k_{1/2}$ of $P_{1/2}$ hold during the concrete executions started with $m^0_{1/2}, a_{1/2}$.

SimIH2.1 holds from the choice of τ. To prove SimIH2.2, we still need to prove that $\models \tau(s^0_2(I_2)) = m^0_2(a_2(I_2))$ for all $I_2 \in \text{locals}(P_2)$. More precisely, since m^0_2 is existentially quantified, we need to show that there exist values $m^0_2(a_2(I_2))$ for all $I_2 \in \text{locals}(P_2)$ such that $\models \tau(s^0_2(I_2)) = m^0_2(a_2(I_2))$. From the algorithm that creates the starting symbolic expressions G^* (page 86), we have that for all $I_2 \in \text{locals}(P_2)$, $s^0_2(I_2) \equiv \text{subst-sim}(G_k, s^0_1, s^0_2)$ for the appropriate G_k, which cannot contain local variables of P_2. Therefore, we can choose $m^0_2(a_2(I_2)) = \mathcal{G}^i[\![\tau(s^0_2(I_2))]\!]\ m^0_1, a_1, m^0_2, a_2$ for all $I_2 \in \text{locals}(P_2)$, since the evaluation $\mathcal{G}^i$ does not require the values of local variables of P_2. This choice implies directly that SimIH2.2 holds. Note that at this point, we specify the values of memory locations of m^0_2 only for the local variables of P_2. Because of the existential quantification of m^0_2, we can still specify the values of the locations "above" the locations for the local variables; we use this for call nodes.

We next prove that SimIH1 holds. From 5.14, we obtain:

$$\begin{aligned}&\models \tau(\text{subst-sim}(J^{in}_k, s^0_1, s^0_2) \Rightarrow\\ &\qquad \text{Sim}(L'_1, s^0_1, L'_2, s^0_2, \{\}, t^0) \land\\ &\qquad \text{subst}(\text{in-context}(P_1, k_1), s^0_1) \land \text{subst}(\text{in-context}(P_2, k_2), s^0_2)).\end{aligned}$$

This further simplifies to: if $\models \tau(\text{subst-sim}(J^{in}_k, s^0_1, s^0_2))$, then

$$\models \tau(\text{Sim}(L'_1, s^0_1, L'_2, s^0_2, \{\}, t^0)) \tag{5.17}$$

and

$$\models \tau(\text{subst}(\text{in-context}(P_1, k_1), s^0_1)) \tag{5.18}$$

and

$$\models \tau(\text{subst}(\text{in-context}(P_2, k_2), s^0_2)). \tag{5.19}$$

Since SimIH2 holds, we can apply the Simulation Congruence Lemma, direction 5.3, and from $m^0_1, a_1, m^0_2, a_2 \models J^{in}_k$, we have $\models \tau(\text{subst-sim}(J^{in}_k, s^0_1, s^0_2))$. Therefore, 5.17 also holds, i.e., $\models \tau(\text{Sim}(L'_1, s_1, L'_2, s_2, i, t))$ which is SimIH1.

Additionally, 5.18 and 5.19 hold. We next prove that the standard invariants of context k_1 hold during the concrete execution of P_1 started with m^0_1, a_1; by analogy, the standard invariants of context k_2 also hold. From 5.15, we know that the context k_1 holds. Therefore, if we prove that the standard input context $F^{in}_{k_1} = \text{in-context}(P_1, k_1)$ holds for m^0_1, a_1, then all standard invariants of k_1 hold. From SimIH2.1, we have that $\models \tau(s^0_1(I_1)) = m^0_1(a_1(I_1))$ for all variables I_1 that can occur in $F^{in}_{k_1}$. Therefore, we can apply the Standard Congruence Lemma 5.2, and from 5.18, we obtain $m^0_1, a_1 \models F^{in}_{k_1}$, which concludes the proof.

5.2.3 Simulation Induction Step

The concrete execution of the node at label L_1 in P_1 has a corresponding sequence of actions from the action tree t. We show that any such sequence of actions preserves the SimIH. At each step, the SimVCG performs the action from the root of the tree. We do a case analysis of the last action performed during the symbolic executions of procedures $P'_{1/2}$. We show that if the SimIH holds before the root action from the tree t is performed, then the SimIH also holds after the action is performed.

A1. The last action is $\texttt{execute}_1$, which symbolically executes an assignment node or a branch node from P'_1.

A1.1. The node is an assignment node: $P'_1(L'_1) \equiv L'_1\texttt{:}I\texttt{=}E$. Before this node is executed, from the induction hypothesis, there exist s_1, s_2, i, t, and τ such that $\text{SimIH}(L_1, m_1, a_1, L_2, m_2, a_2, L'_1, s_1, L'_2, s_2, i, t, \tau)$ holds. When the node is executed, the concrete execution of P_1 makes a transition $\langle L_1, m_1, a\rangle \to \langle L_1 +_{P_1} 1, m'_1, a\rangle$, where $m'_1 = m_1[a_1(I) \mapsto m_1(a_1(E))]$. The symbolic execution continues at the next node $\text{Sim}(L'_1 +_{P'_1} 1, s'_1, L'_2, s'_2, i, t)$ with the new symbolic state $s'_1 = s_1[I \mapsto \text{subst}(\text{translate}(E, e_1), s_1)]$. We need to show that SimIH holds in the new states. We use the same τ to prove that $\text{SimIH}(L_1 +_{P_1} 1, m'_1, a_1, L_2, m_2, a_2, L'_1 +_{P'_1} 1, s'_1, L'_2, s_2, i, \text{left}(t), \tau)$ holds.

SimIH1 and SimIH3 follow directly from the induction hypothesis because τ and i do not change, and the symbolic executions before and after the assignment node generate the same verification condition. We need to prove that SimIH2 holds. SimIH2.2 follows directly from SimIH2.2 of the induction hypothesis because τ, s_2, and m_2 do not change. The proof that SimIH2.1 holds is analogous to the proof for assignment nodes for the StdVC (page 80): we analyze two cases, $I' \not\equiv I$ and $I' \equiv I$, and we use Lemma 3 in the latter case. We omit the details of the proof.

A1.2. The node is a branch node: $P'_1(L'_1) \equiv L'_1\texttt{:br(}E\texttt{)}L''$. There are two paths from this node, and the concrete execution of P_1 takes only one of them depending on the value of the branch condition. However, the symbolic execution takes both paths, and from SimIH1 of the induction hypothesis, we have:

$$\begin{aligned} \models \tau(\,&(G \Rightarrow \text{Sim}(L'', s_1, L'_2, s_2, i, \text{left}(t))) \wedge \\ &(\neg G \Rightarrow \text{Sim}(L'_1 +_{P'_1} 1, s_1, L'_2, s_2, i, \text{right}(t)))\,), \end{aligned} \tag{5.20}$$

where $G = \text{translate-branch}(E, s_1, e_1)$.

From the induction hypothesis, there exists a substitution τ such that SimIH holds before the branch node. The proof that SimIH holds after the branch node is analogous to the proof for branch nodes for the StdVC (page 81). We give an outline of the proof for the case when the branch is taken. We show that $\text{SimIH}(L'', m_1, a_1, L_2, m_2, a_2, L'', s_1, L'_2, s_2, i, \text{left}(t), \tau)$ holds after the branch is taken. SimIH2 and SimIH3 follow directly from the induction hypothesis because m_1, s_1, m_2, s_2, i, and τ remain the same. SimIH1, namely $\models \tau(\mathrm{Sim}(L'', s_1, L'_2, s_2, i, \mathrm{left}(t)))$, follows from 5.20, by the definitions of conjunction and implication and by Lemma 4.

A2. The last action is $\texttt{execute}_2\ B$, which either symbolically executes an assignment node or a branch node from P'_2 or moves past a simulation invariant from P'_2.

A2.1. The node is an assignment node: $P'_2(L'_2) \equiv L'_2 \texttt{:} I\texttt{=}E$. The proof that SimIH holds after the executions of an assignment node in P'_2 and P_2 is analogous to the proof that SimIH holds after the executions of an assignment node in P'_1 and P_1 (page 92).

A2.2. The node is a branch node: $P'_2(L'_2) \equiv L'_2\texttt{:br(}E\texttt{)}L''$. The symbolic execution of a branch node in P'_2 differs from the symbolic execution of a branch node in P'_1. In P'_1, the symbolic execution splits at a branch node and follows both paths. In P'_2, the symbolic execution follows only one path, either branch-taken or branch-not-taken, depending on B. We prove that the corresponding concrete execution also takes the same path. We give a proof that these executions preserve the SimIH for the branch-taken case; the branch-not-taken case is analogous. From the induction hypothesis, $\mathrm{SimIH}(L_1, m_1, a_1, L_2, m_2, a_2, L'_1, s_1, L'_2, s_2, i, t, \tau)$ holds (for some values of the existentially quantified arguments) before the node is executed. We show that $\mathrm{SimIH}(L_1, m_1, a_1, L'', m_2, a_2, L'_1, s_1, L'', s_2, i, \mathrm{left}(t), \tau)$ holds after the node is executed.

SimIH2 and SimIH3 are trivial to prove. We show that SimIH1 holds. From the induction hypothesis,

$$\models \tau(G \wedge \mathrm{Sim}(L_1, s_1, L'', s_2, i, \mathrm{left}(t))), \tag{5.21}$$

where $G = \text{translate-branch}(E, s_2, e_2)$. From 5.21, we have $\models \tau(G)$. Further, by Lemma 4, $m_2(a_2(E)) \not\equiv \texttt{0}$ which proves that the concrete execution of P_2 makes the branch-taken transition: $\langle L_2, m_2, a_2\rangle \rightarrow \langle L'', m_2, a_2\rangle$. From 5.21, we also obtain $\models \tau(\mathrm{Sim}(L_1, s_1, L'', s_2, i, \mathrm{left}(t)))$ which is SimIH1.

A2.3. The node is a half of a simulation invariant: $P'_2(L'_2) \equiv L'_2\texttt{:sim-inv}\ J\texttt{,}L''_1$. From the induction hypothesis, we know that there exists a symbolic execution of P'_2 such that $\mathrm{SimIH}(L_1, m_1, a_1, L_2, m_2, a_2, L'_1, s_1, L'_2, s_2, i, t, \tau)$ holds. The symbolic execution of the half of an invariant only moves past it. We prove that $\mathrm{SimIH}(L_1, m_1, a_1, L_2, m_2, a_2, L'_1, s_1, L'_2 +_{P'_2} 1, s_2, i, \mathrm{left}(t), \tau)$ holds. Since the states and the substitution do not change, SimIH2 and SimIH3 follow directly from their counterparts in the induction hypothesis. SimIH1 also follows from SimIH1 of the induction hypothesis because the symbolic executions before and after the half of an invariant generate the same simulation verification condition.

A3. The last action is `stop`, which finishes the symbolic executions. We need to prove that *if* SimIH holds before `stop`, then SimIH holds after `stop`. We show that this implication is valid by showing that SimIH cannot hold for any substitution τ before this action. By contradiction, assume that there exists a substitution τ (as well as the other arguments of the induction hypothesis relation) such that SimIH holds before `stop`. Since the execution of `stop` generates `false`, we obtain, from SimIH1, $\models \tau(\texttt{false})$. For all substitutions $\tau(\texttt{false}) \equiv \texttt{false}$, and we have $\models \texttt{false}$ which cannot hold, by the definition of the valuation function for `false` (Figure 4-8). Hence, SimIH does not hold before action `stop`. (This means that `stop` cannot be used on some path during a symbolic execution of P'_1 if that path is possible during a concrete execution of P_1. If a path is not possible during any concrete execution, `stop` can be used.)

A4. The last action is `split` F, which splits the symbolic execution of P'_1 into two paths. This action can be used at any point in the symbolic execution. Before the action, from the induction hypothesis, there exist s_1, s_2, i, t, and τ such that $\text{SimIH}(L_1, m_1, a_1, L_2, m_2, a_2, L'_1, s_1, L'_2, s_2, i, t, \tau)$ holds. From SimIH1 of the induction hypothesis, the following holds:

$$\begin{aligned}\models \tau(\,&(F' \Rightarrow \text{Sim}(L'_1, s_1, L'_2, s_2, i, \text{left}(t))) \wedge \\ &(\neg F' \Rightarrow \text{Sim}(L'_1, s_1, L'_2, s_2, i, \text{right}(t)))\,),\end{aligned} \tag{5.22}$$

where $F' = \text{subst}(F, s_1)$.

We next show that SimIH holds after the action if $m_1, a_1 \models F$; the case when $m_1, a_1 \models \neg F$ is analogous. (Note that by the definition of valuation functions, either F or $\neg F$ holds for any m_1, a_1.) The proof is similar to the proof for branch nodes. We use the same s_1, s_2, i, and τ from the induction hypothesis and $t' = \text{left}(t)$ to show that $\text{SimIH}(L_1, m_1, a_1, L_2, m_2, a_2, L'_1, s_1, L'_2, s_2, i, t', \tau)$ holds. SimIH2 and SimIH3 follow directly from their counterparts in the induction hypothesis. For SimIH1, we need to show $\models \tau(\text{Sim}(L'_1, s_1, L'_2, s_2, i, \text{left}(t)))$. From 5.22, we have that SimIH1 holds if $\models \tau(F')$. Further, $\models \tau(F')$ from the assumption $m_1, a_1 \models F$, by the Standard Congruence Lemma 5.1, since $\models \tau(s_1(I_1)) = m_1(a_1(I_1))$ for all $I_1 \in \text{vars}(P_1)$ from SimIH2.1 of the induction hypothesis. This concludes the proof of this case.

A5. The last action is $\texttt{use-analysis}_1$ or $\texttt{use-analysis}_2$, which includes a standard invariant in the simulation verification condition. From the induction hypothesis, there is t such that $\text{SimIH}(L_1, m_1, a_1, L_2, m_2, a_2, L'_1, s_1, L'_2, s_2, i, t, \tau)$ holds before the action. We prove that SimIH holds for $t' = \text{left}(t)$ after $\texttt{use-analysis}_1$; the proof for $\texttt{use-analysis}_2$ is analogous. SimIH2 and SimIH3 follow directly from the induction hypothesis. To prove SimIH1, let F be the formula from the standard invariant at L_1 in the context k_1 of P_1: $F = \text{std-invariant}(P_1, k_1, L_1)$. From SimIH1 of the induction hypothesis, we have

$$\models \tau(\text{subst}(F, s_1) \Rightarrow \text{Sim}(L_1, s_1, L_2, s_2, i, \text{left}(t))). \tag{5.23}$$

We have proven in the simulation base case (Section 5.2.2) that all standard invariants of context k_1 hold during the concrete execution of P_1, and therefore

$m_1, a_1 \models F$. Since SimIH2.1 holds, we can apply the Standard Congruence Lemma 5.1 to obtain: $\models \tau(\text{subst}(F, s_1))$. From 5.23, we then have that SimIH1 holds: $\models \tau(\text{Sim}(L_1, s_1, L_2, s_2, i, \text{left}(t)))$.

A6. The last action is `execute-both`, which symbolically executes a node from both P'_1 and P'_2. The two nodes can be both return nodes, or both call nodes, or both halves of some simulation invariant.

A6.1. The nodes are return nodes: $P'_1(L'_1) \equiv L'_1$`:ret` and $P'_2(L'_2) \equiv L'_2$`:ret`. These nodes are the last nodes in the concrete executions of procedure activations, and therefore we do not show that SimIH holds after the return nodes. We still need to show that the simulation output context holds whenever the procedures reach return nodes. From SimIH1 of the induction hypothesis, $\models \tau(\text{subst-sim}(J_k^{out}, s_1, s_2))$ for some τ. Since SimIH2 holds for the same τ, by the Simulation Congruence Lemma 5.4, we obtain $m_1, a_1, m_2, a_2 \models J_k^{out}$.

A6.2. The nodes are call nodes: $P'_1(L'_1) \equiv L'_1 \texttt{:} I_1\texttt{(}E^*_2\texttt{)}$ and $P'_2(L'_2) \equiv L'_2 \texttt{:} I_2\texttt{(}E^*_2\texttt{)}$. Let the callee procedures be $P''_1 \equiv$ `proc` $I_1\texttt{(}I^*_1\texttt{)}$ D^*_1 $\{N^+_1\}$ and $P''_2 \equiv$ `proc` $I_2\texttt{(}I^*_2\texttt{)}$ D^*_2 $\{N^+_2\}$. The concrete executions make transitions to the first nodes of $P''_{1/2}$, allocating memory for the parameters and local variables of $P''_{1/2}$:

$$\langle m^{in}_{1/2}, a^{in}_{1/2}, p^{in}_{1/2}\rangle = \\ \text{alloc-locals}(\text{alloc-params}(\langle m_{1/2}, a_{1/2}, p_{1/2}\rangle, I^*_{1/2}, m_{1/2}(a_{1/2}(E^*_{1/2}))), D^*_{1/2}).$$

When and if the executions of $P''_{1/2}$ return, the concrete executions continue from $\langle P_{1/2}(L_{1/2} +_{P_{1/2}} 1), m'_{1/2}, a_{1/2}\rangle$, where the memories after the calls satisfy: $m'_{1/2}(a_{1/2}(I_{1/2})) = m_{1/2}(a_{1/2}(I_{1/2}))$ for all $I_{1/2} \in \text{locals}(P_{1/2}) \cup \text{params}(P_{1/2})$. (We show later that the executions of $P''_{1/2}$ either both return or both do not return.)

Let $k'' = \text{sim-context-index}(L_1, L_2, K^*_k)$ be the index of the simulation context for the call nodes at $L_{1/2}$, and let $\langle J^{in}, J^{out}\rangle = \text{sim-context}(I_1, I_2, k'')$ be the input and output contexts for the simulation context k'' of procedures $I_{1/2}$, i.e., $P_{1/2}$. The symbolic executions continue, after the return nodes, with the new symbolic states: $s'_{1/2} = s_{1/2}[I^*_{1/2} \mapsto x^*_{1/2}]$ for all $I^*_{1/2} \in \text{globals}(P_{1/2})$, where $x^* = x^*_1 \cup x^*_2$ are fresh logic variables. (Some variables in x^*_1 can be the same as some variables in x^*_2 since $\langle s'_1, s'_2, x^*\rangle = \text{fresh-sim-globals}(s_1, s_2, J^{out})$.)

From SimIH1 of the induction hypothesis, there is a substitution τ such that:

$$\models \tau(\text{subst-sim}(J^{in}, \text{set-params}(I_1, G^*_1, s_1), \\ \text{set-params}(I_2, G^*_2, s_2))) \quad (5.24)$$

and

$$\models \tau(\forall x^*.\ \text{subst-sim}(J^{out}, s'_1, s'_2) \Rightarrow \\ \text{Sim}(L_1 +_{P'_1} 1, s'_1, L_2 +_{P'_2} 1, s'_2, i, \text{left}(t))), \quad (5.25)$$

where $G^*_{1/2} = \text{translate-call}(E^*_{1/2}, s_{1/2}, e_{1/2})$. We use $s^{in}_{1/2}$ to denote the symbolic states for the formula J^{in}: $s^{in}_{1/2} = \text{set-params}(I_{1/2}, G^*_{1/2}, s_{1/2}) = s_{1/2}[I^*_{1/2} \mapsto G^*_{1/2}]$ for parameters $I^*_{1/2}$ of procedures $I_{1/2}$. To prove that SimIH holds after the calls, we use the substitution $\tau' = \tau \cup \{x^* \mapsto m'_{1/2}(a_{1/2}(I^*_{1/2}))\}$. (We show later that we can make such a substitution.) We prove that the following relation holds:

$$\text{SimIH}(L_1 +_{P_1} 1, m'_1, a_1, L_2 +_{P_2} 1, m'_2, a_2, L'_1 +_{P'_1} 1, s'_1, L'_2 +_{P'_2} 1, s'_2, i, \text{left}(t), \tau').$$

The proof is similar to the proof for call nodes for the StdVC (page 81). SimIH3 follows from SimIH3 of the induction hypothesis because $\tau \subseteq \tau'$ and i does not change after call nodes. We need to prove that SimIH1 and SimIH2 also hold. We prove first that the simulation input context J^{in} holds, next that the simulation output context J^{out} holds, and then that SimIH2 holds[4] and finally that SimIH1 holds.

We first observe that $\models \tau(s^{in}_{1/2}(I_{1/2})) = m^{in}_{1/2}(a^{in}_{1/2}(I_{1/2}))$ for all variables $I_{1/2} \in \text{globals}(P''_{1/2}) \cup \text{params}(P''_{1/2})$ that can occur in J^{in}. The proof that the states $s^{in}_{1/2}$ and $m^{in}_{1/2}, a^{in}_{1/2}$ coincide for all variables in J^{in} is analogous (for each procedure) to the proof for the StdVC (page 82), and we do not repeat it here. From 5.24, we have $\models \tau(\text{subst-sim}(J^{in}, s^{in}_1, s^{in}_2)$. Since $\models \tau(s^{in}_{1/2}(I_{1/2})) = m^{in}_{1/2}(a^{in}_{1/2}(I_{1/2}))$ for all variables $I_{1/2}$ in J^{in}, by the Simulation Congruence Lemma 5.4, we obtain $m^{in}_1, a^{in}_1, m^{in}_2, a^{in}_2 \models J^{in}$. This means that the simulation input context J^{in} holds at the beginning of P''_1 and P''_2.

For every simulation context k'' of every pair of procedures $P''_{1/2}$ in $Q_{1/2}$, if the simulation input context holds at the beginning of activations of $P''_{1/2}$, then the simulation output context holds at the end of the activations. Similarly to the proof for the StdVC, the proof for SimVC is informal at this point and it appears that we assume assumption the actual statement that we try to establish, namely $Q_1 \rhd Q_2$. We do not present a full proof that would require an induction on the height of the procedure call trees of programs $Q_{1/2}$. (Note that the two programs have isomorphic call trees because of the simultaneous execution of the call and return nodes.) We show later that if the activation of P''_1 does not terminate, then there exists an activation of P''_2 that also does not terminate. We additionally discuss the existential quantification of the executions of P''_2 after we finish the proof that SimIH holds.

Since J^{in} holds at the beginning of $P''_{1/2}$, then, for all executions of P''_1,

[4] For the StdVC we can prove that SimIH2 holds even before we prove that the standard output context holds because the substitution τ' does not depend on the output context.

there is an execution of P''_2 such that $m'_1, a^{in}_1, m'_2, a^{in}_2 \models J^{out}$, where $m'_{1/2}$ are the memories at the end of the activations of $P''_{1/2}$, and $a^{in}_{1/2}$ are the environments for those activations. Additionally, the environments $a^{in}_{1/2}$ for $P''_{1/2}$ and $a_{1/2}$ for $P_{1/2}$ map the global variables to the same addresses. Since J^{out} can contain only the global variables, we have $m'_1, a_1, m'_2, a_2 \models J^{out}$. By the Well-Defined Substitution Lemma, we can then make substitution $\{x^* \mapsto m'_{1/2}(a_{1/2}(I^*_{1/2}))\}$ and, for that substitution, $\models \tau'(s_{1/2}(I_{1/2})) = m'_{1/2}(a_{1/2}(I_{1/2}))$ for all $I_{1/2} \in \text{globals}(P_{1/2})$. To prove that SimIH2 holds, we need also to prove that $\models \tau'(s_{1/2}(I_{1/2})) = m'_{1/2}(a_{1/2}(I_{1/2}))$ for all $I_{1/2} \in \text{locals}(P_{1/2}) \cup \text{params}(P_{1/2})$. It holds by the following: $\models \tau'(s'_{1/2}(I_{1/2})) = \tau'(s_{1/2}(I_{1/2})) = \tau(s_{1/2}(I_{1/2})) = m_{1/2}(a_{1/2}(I_{1/2})) = m'_{1/2}(a_{1/2}(I_{1/2}))$ because the executions of $P''_{1/2}$ cannot change local variables of $P_{1/2}$. We next prove that SimIH1 holds.

Since SimIH2 holds, and $m'_1, a_1, m'_2, a_2 \models J^{out}$, by the Simulation Congruence Lemma 5.3, we have $\models \tau'(\text{subst-sim}(J^{out}, s'_1, s'_2))$. Further, from 5.25 and the choice of τ', we have:

$$\models \tau'(\text{subst-sim}(J^{out}, s'_1, s'_2) \Rightarrow \text{Sim}(L_1 +_{P'_1} 1, s'_1, L_2 +_{P'_2} 1, s'_2, i, \text{left}(t))),$$

and therefore $\models \tau'(\text{Sim}(L_1 +_{P'_1} 1, s'_1, L_2 +_{P'_2} 1, s'_2, i, \text{left}(t)))$, which is SimIH1. This concludes the proof that the executions of call nodes preserve the SimIH.

We now discuss the quantification of the executions of $P''_{1/2}$ and the termination simulation between those executions. In the proof for call nodes, we assume that for the executions of $P''_{1/2}$, J^{out} holds at the end if J^{in} holds at the beginning. Specifically, we use that for all initial values for the local variables of P''_1, there exist some initial values for the local variables of P''_2 such that J^{out} holds at the end. We need to show that we can indeed choose these initial values for the appropriate locations in the memory m^{in}_2 at the beginning of P''_2, i.e., those initial values are not already specified or universally quantified. We next give an outline of the proof for this.

We first recall that the memory and the stack pointer at the beginning of P_2 are m^0_2 and p^0_2. From the simulation requirement, we can choose the values of memory locations for the local variables of m^0_2 and also for all locations with addresses greater than p^0_2. In the simulation base case (Section 5.2.2), we have chosen the values for the local variables of P_2. We now prove by induction that we can choose the values of local variables for all calls in the call tree starting from P_2. The induction hypothesis is: if the current memory is m_2 and the stack pointer is p_2, then we can choose the values of locations in m_2 with addresses greater than p_2. This apparently holds before the first call in P_2. For a call to any P''_2, the memory and the stack pointer after the call are:

$$\langle m^{in}_2, a^{in}_2, p^{in}_2 \rangle = \text{alloc-locals}(\text{alloc-params}(\langle m_2, a_2, p_2 \rangle, I^*_2, m(a(E^*_2))), D^*_2),$$

with the local variables of P''_2 being allocated in m_2^{in} at addresses greater than p_2 and less than p_2^{in}. By the induction hypothesis, we can choose the initial values for those variables. Further, we can choose the initial values for all locations with addresses greater than p_2^{in}. Since the value of the stack pointer does not decrease after the returns, it follows inductively that we can choose the initial values for the local variables for all calls from P''_2 and also for all calls from P_2 after P''_2 returns.

Finally, we discuss the termination simulation of the executions of $P''_{1/2}$: if P''_1 does not terminate, then P''_2 does not terminate either. A partial execution of a procedure terminates if it reaches a return node. In the above proof that call nodes preserve the SimIH, we have used the following: for every execution of P''_1 that terminates, there exists an execution of P''_2 that also terminates such that J^{out} holds at the end of the executions if J^{in} holds at the beginning. We explained informally that this can be proven by an induction on the height of the procedure call trees of programs $Q_{1/2}$. For every execution of Q_1, with an arbitrary call tree, there exists an execution of Q_2 with an isomorphic call tree because whenever the execution of Q_1 reaches a call node or a return node, there exists an execution of Q_2 that reaches, respectively, a call node or a return node. We have not shown, however, what happens if Q_1 (P''_1) does not terminate. We next show that then exists an execution of Q_2 (P''_2) that also does not terminate.

A program does not terminate if it has an infinite partial execution (a partial execution of infinite length). A partial execution of a program is infinite in one of the following two cases: the execution has a call tree of infinite height (*infinite recursion*) or the execution calls a procedure which has an infinite partial execution (*infinite loop*). We analyze the case of an infinite loop in Section 5.2.4. (We postpone the analysis until we complete the proof for the simulation induction step, including the case for simulation invariants.) We next analyze informally the case of an infinite recursion.

If the execution of Q_1 has a call tree of infinite height, then there exists an execution of Q_2 that has a call tree of infinite height. The same relation holds for the subtrees of these call trees: if the subtree (of the call tree for Q_1) whose root is the activation of P''_1 has infinite height, then there exists an activation of P''_2 which is the root of a subtree (of the call tree for Q_2) that has infinite height. Therefore, the termination simulation is satisfied in the case of an infinite recursion. In this case, any simulation output context is allowed. For example, it is possible to prove that the context with $J^{in} \equiv$ `true` and $J^{out} \equiv$ `false` holds for procedures:

```
proc I1() {g=1;I1();ret;}      proc I2() {g=2;I2();ret;}
```

It is also possible to prove that the global variables `g` have the same value at the end of the procedures $I_{1/2}$. The reason is that simulation output context is required to hold only *if* the executions of $I_{1/2}$ reach the return nodes.

A6.3. The nodes are two halves of a simulation invariant: $P'_1(L'_1) \equiv L'_1$`:sim-inv` and $P'_2(L'_2) \equiv L'_2$`:sim-inv` J`,`L'_1. There are two cases depending on whether the simulation invariant L_1`,`L_2`:sim-inv` J has been already symbolically executed or not.

A6.3.1. The simulation invariant is symbolically executed for the first time if $\langle\langle L'_1, L'_2\rangle, \langle s'_1, s'_2\rangle, t'\rangle \notin i$ for any s'_1, s'_2, and t'. From SimIH1 of the induction hypothesis, there is a substitution τ such that:

$$\models \tau(\text{subst-sim}(J, s_1, s_2)) \tag{5.26}$$

and

$$\models \tau(\forall x^*.\ \text{subst-sim}(J, s'_1, s'_2) \Rightarrow \text{Sim}(L'_1 +_{P'_1} 1, s'_1, L'_2 +_{P'_2} 1, s'_2, i', \text{left}(t))), \tag{5.27}$$

where $\langle s'_1, s'_2, x^*\rangle = \text{fresh-sim-sym-states}(s_1, s_2, J)$, i.e., for all $I_{1/2} \in \text{vars}(P_{1/2})$, the states $s'_{1/2} = \{I^*_{1/2} \mapsto x^*_{1/2}\}$ for some fresh logic variables $x^* = x^*_1 \cup x^*_2$ and $i' = \langle\langle L'_1, L'_2\rangle, \langle s'_1, s'_2\rangle, \text{left}(t)\rangle \cup i$.

From SimIH2 of the induction hypothesis, for all $I_{1/2} \in \text{vars}(P_{1/2})$: $\models \tau(s_{1/2}(I_{1/2})) = m_{1/2}(a_{1/2}(I_{1/2}))$. We can therefore apply the Simulation Congruence Lemma 5.4, and from 5.26 we obtain $m_1, a_1, m_2, a_2 \models J$, which is one of the requirements for $P_1, Q_1 \rhd_k P_2, Q_2$. We need additionally to show that SimIH holds after the symbolic execution of the invariant. By the Well-Defined Substitution Lemma, we can make the substitution $\tau' = \tau \cup \{x^* \mapsto m'_{1/2}(a_{1/2}(I^*_{1/2}))\}$. We use it to show that $\text{SimIH}(L_1, m_1, a_1, L_2, m_2, a_2, L'_1 +_{P'_1} 1, s'_1, L'_2 +_{P'_2} 1, s'_2, i', \text{left}(t), \tau')$ holds after the invariant.

SimIH2 follows from the choice of τ'. Since SimIH2 holds, and we have proven $m_1, a_1, m_2, a_2 \models J$, by the Simulation Congruence Lemma 5.3, we have $\models \tau'(\text{subst-sim}(J, s'_1, s'_2))$. Further, from 5.27 and the choice of τ', we obtain $\models \tau'(\text{Sim}(L'_1 +_{P'_1} 1, s'_1, L'_2 +_{P'_2} 1, s'_2, i', \text{left}(t)))$, which is SimIH1. We need still to prove SimIH3, i.e., that τ' is correct with respect to i'. We prove that SimIH3.2 holds. SimIH3.2.1–SimIH3.2.3 hold by the choice of τ'. SimIH3.2.4 holds because of 5.27 (τ_1 is τ). SimIH3.2.5 follows from SimIH3 of the induction hypothesis.

A6.3.2. The simulation invariant has been previously symbolically executed if $\langle\langle L'_1, L'_2\rangle, \langle s'_1, s'_2\rangle, t'\rangle \in i$ for some symbolic states $s'_{1/2}$ and some action tree t'. From SimIH1 of the induction hypothesis, there is a substitution τ such that $\models \tau(\text{subst-sim}(J, s_1, s_2))$. Since SimIH2 of the induction hypothesis holds, we can apply the Simulation Congruence Lemma 5.4 to obtain $m_1, a_1, m_2, a_2 \models J$, which is one of the requirements for $P_1, Q_1 \rhd_k P_2, Q_2$. We additionally show that SimIH holds after the invariant by showing that there exists τ' such that $\text{SimIH}(L_1, m_1, a_1, L_2, m_2, a_2, L'_1 +_{P'_1} 1, s'_1, L'_2 +_{P'_2} 1, s'_2, i, t', \tau')$ holds.

Since the invariant has been executed, $i = \{\langle\langle L'_1, L'_2\rangle, \langle s'_1, s'_2\rangle, t'\rangle\} \cup i_1$, where $s'_{1/2} = \{I^*_{1/2} \mapsto x_{1/2}\}$ for some $x^* = x^*_1 \cup x^*_2$ and t' is the action tree for the symbolic execution after the invariant. From SimIH3.2.2 of the induction hypothesis, we know that $\tau = \tau_1 \cup \{x^* \mapsto Z^*\}$ for some τ_1 such that, from SimIH3.2.3, none of x from x^* is in τ_1. Therefore, we can remap x^*, and by the Well-Defined Substitution Lemma, we can make $\tau' = \tau_1 \cup \{x^* \mapsto m_{1/2}(a_{1/2}(I^*_{1/2}))\}$. We prove that SimIH holds for that τ'.

SimIH2 follows from the choice of τ'. SimIH3 follows from SimIH3 of the induction hypothesis. We next show that SimIH1 also holds. From SimIH3.2.4 of the induction hypothesis, we have:

$$\begin{aligned} \models \tau(\forall x^*.\ \text{subst-sim}(J, s'_1, s'_2) \Rightarrow & \\ & \text{Sim}(L'_1 +_{P'_1} 1, s'_1, L'_2 +_{P'_2} 1, s'_2, i, t')), \end{aligned} \qquad (5.28)$$

As we have already shown that SimIH2 holds and $m_1, a_1, m_2, a_2 \models J$, we obtain $\models \tau'(\text{subst-sim}(J, s'_1, s'_2))$ by the Simulation Congruence Lemma 5.3. Therefore, from 5.28, we finally have that $\models \tau'(\text{Sim}(L'_1 +_{P'_1} 1, s'_1, L'_2 +_{P'_2} 1, s'_2, i, t'))$.

This concludes the proof for the simulation invariants and the whole simulation induction step. We next point out an important property of the symbolic execution of simulation invariants. Observe that the symbolic execution of an invariant moves past the halves of the invariant both in P'_1 and P'_2. In P'_1, the node after the half of an invariant has to be a non-invariant node. Therefore, between two consecutive executions of an invariant from P'_1, at least one non-invariant node is executed. (The symbolic execution of such a node corresponds directly to the concrete execution of the same node.)

Whenever a half of an invariant from P'_1 is executed, a half of an invariant from P'_2 is also executed. In P'_2, there can be many halves of invariants in front of some non-invariant node. Therefore, several consecutive executions of nodes from P'_2 can execute halves of invariants. However, there are finitely many halves of invariants in P'_2, and after finitely many executions of invariants, a non-invariant node has to be executed. We can formalize this reasoning to prove that for the consecutive concrete executions of P_1 that reach a simulation invariant, the length of the corresponding concrete executions of P_2 does not decrease, and, in fact, increases after a finite number of executions. We use this property in the proof of the termination simulation.

5.2.4 Termination Simulation

In this section we complete the proof that the validity of the simulation verification condition for two programs Q_1 and Q_2, i.e., $\models F^{vc}_{Q_1,Q_2}$, implies the termination simulation of those programs. More specifically, we prove that for any simulation context k for procedures $P_{1/2}$ of programs $Q_{1/2}$, P_1 may not terminate only if P_2 may not terminate.

An activation of a procedure does not terminate if the activation either contains an infinite loop or calls another activation that does not terminate. (Note that the activation can call, within the loop, other activations that terminate.) If the called activation does not terminate, it can be again because of an infinite loop or a call that does not terminate. Eventually, either some activation in the call tree has an infinite loop or the call tree has infinite height.

We have argued in the proof for call nodes (page 98) that if an execution of P_1 has a call tree of infinite height, then there is an execution of P_2 that also has a call tree of infinite height. Therefore, we analyze only the case of an infinite loop in this section. An activation of a procedure has an infinite loop if the partial execution of that activation has infinite length.[5] We prove the termination simulation theorem as follows.

Theorem 3 *(Termination Simulation) Assume that $\models F^{vc}_{Q_1,Q_2}$ for two programs Q_1 and Q_2. If a partial execution $\langle P^0_1, m^0_1, a_1\rangle \overset{+}{\dashrightarrow} \langle L_1, m_1, a_1\rangle$ of P_1 has infinite length, then there exists a partial execution $\langle P^0_2, m^0_2, a_2\rangle \overset{+}{\dashrightarrow} \langle L_2, m_2, a_2\rangle$ of P_2 such that if $m^0_1, a_1, m^0_2, a_2 \models J^{in}_k$, then the partial execution $\langle P^0_2, m^0_2, a_2\rangle \overset{+}{\dashrightarrow} \langle L_2, m_2, a_2\rangle$ has infinite length.*

Proof: By contradiction, assume that there exists no infinite partial execution $\langle P^0_2, m^0_2, a_2\rangle \overset{+}{\dashrightarrow} \langle L_2, m_2, a_2\rangle$ of P_2 such that $m^0_1, a_1, m^0_2, a_2 \models J^{in}_k$. Since $\models F^{vc}_{Q_1,Q_2}$, we have by the proof in the previous section[6], that for all $\langle P^0_1, m^0_1, a_1\rangle \overset{+}{\dashrightarrow} \langle L_1, m_1, a_1\rangle$ of P_1, there exists $\langle P^0_2, m^0_2, a_2\rangle \overset{+}{\dashrightarrow} \langle L_2, m_2, a_2\rangle$ of P_2 such that for all L_1`,`L'_2`:sim-inv` J' from S^*_k, there exists L_1`,`L''_2`:sim-inv` $J \in$ set-sim-inv(L_1, S^*_k) such that $L_2 \equiv L''_2$ and $m_1, a_1, m_2, a_2 \models J$. In other words, for all partial executions of P_1 that reach (a half of) a simulation invariant, there exists an execution of P_2 that reaches a corresponding (half of) simulation invariant. (Also, whenever the executions reach the invariant, its formula holds, but we do not need that for this proof.)

Since the partial execution of P_1 has infinite length, it executes at least one node in P_1 infinite number of times. Further, since $\models F^{vc}_{Q_1,Q_2}$, the SimVCG could generate $F^{vc}_{Q_1,Q_2}$. and therefore it follows that the execution reaches at least one half of a simulation invariant infinite number of times. Let the label for that invariant be L_1. We show that the partial executions of P_2 that reach a corresponding half of the invariant cannot be all finite, which thus contradicts the assumption that they are all finite.

We now consider different partial executions of P_1 (started from the same configuration $\langle P^0_1, m^0_1, a_1\rangle$) that reach the same L_1. Recall that the notation $\langle P^0_1, m^0_1, a_1\rangle \overset{+}{\dashrightarrow} \langle L_1, m_1, a_1\rangle$ is an abbreviation for the sequence $\langle P^0_1, m^0_1, a_1\rangle \overset{+}{\rightarrow} \ldots \overset{+}{\rightarrow} \langle L_1, m_1, a_1\rangle$. We can order the partial executions that reach L_1 by their length. Let the sequence

[5] An activation of a procedure has a call that does not terminate if the partial execution of that activation has a finite length, but the last configuration is a call node and not a return node.

[6] Note that the proof in the previous section and the current proof should be actually done simultaneously as described for the call nodes.

of such executions be:

$$\begin{array}{l}
pe_1^1 = \langle P_1^0, m_1^0, a_1\rangle \xrightarrow{+} \ldots \xrightarrow{+} \langle L_1, m_1^1, a_1\rangle \\
pe_1^2 = \langle P_1^0, m_1^0, a_1\rangle \xrightarrow{+} \ldots \xrightarrow{+} \langle L_1, m_1^1, a_1\rangle \xrightarrow{+} \ldots \xrightarrow{+} \langle L_1, m_1^2, a_1\rangle \\
pe_1^3 = \langle P_1^0, m_1^0, a_1\rangle \xrightarrow{+} \ldots \xrightarrow{+} \langle L_1, m_1^1, a_1\rangle \xrightarrow{+} \ldots \xrightarrow{+} \langle L_1, m_1^2, a_1\rangle \xrightarrow{+} \ldots \xrightarrow{+} \langle L_1, m_1^3, a_1\rangle \\
\vdots
\end{array}$$

Note that the memories m_1^i and m_1^j for two different partial executions pe_1^i and pe_1^j may or may not be different.

For each partial execution pe_1^i of P_1, there is a corresponding partial execution pe_2^i of P_2. Let the sequence of such corresponding executions be:

$$\begin{array}{l}
pe_2^1 = \langle P_2^0, m_2^0, a_2\rangle \xrightarrow{+} \ldots \xrightarrow{+} \langle L_2^1, m_2^1, a_2\rangle \\
pe_2^2 = \langle P_2^0, m_2^0, a_2\rangle \xrightarrow{+} \ldots \xrightarrow{+} \langle L_2^2, m_2^2, a_2\rangle \\
pe_2^3 = \langle P_2^0, m_2^0, a_2\rangle \xrightarrow{+} \ldots \xrightarrow{+} \langle L_2^3, m_2^3, a_2\rangle \\
\vdots
\end{array}$$

where L_1`,`L_2^i`:sim-inv` $J^i \in$ set-sim-inv(L_1, S_k^*). Note that the labels L_2^i and L_2^j, as well as the memories m_2^i and m_2^j, for two different indices i and j may or may not be different. Further, even the whole partial executions pe_2^i and pe_2^j, and thus their lengths, may or may not be different. We denote the length of pe_2^i by $|pe_2^i|$.

From $\models F_{Q_1,Q_2}^{vc}$, we can prove that the length of the corresponding partial executions does not decrease (for all i and for all $j > i$, $|pe_2^j| \geq |pe_2^i|$). Even more, we can prove that the length must increase after a finite number of executions in the sequence pe_2^i (for all i, exists $j > i$ such that $|pe_2^j| > |pe_2^i|$). The proof proceeds as outlined in the previous section (page 100). Therefore, the maximum length of pe_2^i cannot be bounded by any finite number, i.e., there is a partial execution of P_2 that has infinite length.

This concludes the proof that if the execution of P_1 does not reach a return node, then there exists an execution of P_2 that does not reach a return node either. We have also proven that if the execution of P_1 reaches a return node, then there exists an execution of P_2 that reaches a return node (and the simulation output context holds). This completes the proof of all simulation requirements—P_1 can generate only the results that P_2 can generate.

We next give examples of two interesting cases of simulation. First, P_2 may generate more results than P_1. For example, if P_1 never terminates, P_2 can terminate for some initial values of the local variables, but there definitely exist some initial values for which P_2 does not terminate:

`proc` I_1`()` $\{L_1$`:br(TRUE)`L_1`;ret;`$\}$ `proc` I_2`()` $\{L_2$`:br(i` $==$ `0)`L_2`;ret;`$\}$

Conversely, if P_1 always terminates, P_2 may not terminate for some initial values of the local variables, but there definitely exist some initial values for which P_2 terminates:

$$\texttt{proc}\ I_1\texttt{()}\ \{\texttt{ret;}\} \qquad \texttt{proc}\ I_2\texttt{()}\ \{L_2\texttt{:br(i} == 0\texttt{)}L_2\texttt{;ret;}\}$$

In the above examples, the result of P_2 depends on the uninitialized local variable `i`.

Second, if neither of procedure terminates, any simulation output context is provable. This is sound because the simulation definition requires the simulation output context to hold only if the executions reach return nodes. For example, for procedures:

$$\texttt{proc}\ I_1\texttt{()}\ \{\texttt{g=1;}L_1\texttt{:br(TRUE)}L_1\texttt{;ret;}\} \qquad \texttt{proc}\ I_2\texttt{()}\ \{\texttt{g=2;}L_2\texttt{:br(TRUE)}L_2\texttt{;ret;}\}$$

the compiler can prove that the output context is $\texttt{g}_1 = \texttt{g}_2$ for the input context `true`. This clearly would not hold if the executions could reach the return nodes.

Chapter 6

Extensions and Limitations

In Chapter 4 we presented a framework for credible compilation. The basic framework is for a simple language which we call basic language (BL). BL has only simple integer variables, the same as the initial language for credible compilation [41]. The advance is that BL has procedures and therefore supports modularity, whereas the programs in the initial language consist of only one procedure. However, the approach is essentially the same for both languages. First, the compiler outputs the *standard contexts* (and invariants) for the analysis results and the *simulation contexts* (and invariants) together with the transformed program. Next, the verifier uses the verification-condition generator (VCG), which consists of the standard VCG (StdVCG) and the simulation VCG (SimVCG), to generate the standard verification condition (StdVC) and the simulation verification condition (SimVC). These conditions are logic formulas, and the compiler has the obligation to prove that the formulas hold. What changes for BL, in comparison with the initial language, is the VCG.

In this chapter we discuss how the framework for credible compilation could handle more realistic programming languages. The main strategy still remains the same—the verifier requires the compiler to produce the standard and simulation contexts, and to prove the verification conditions for those contexts. We extend BL with some C constructs, such as pointers and arrays. We also discuss some extensions to the language semantics, in particular adding error states. The changes to the language syntax and semantics clearly necessitate changes to the verification-condition generators. For some changes, it is also necessary to change the logic, more precisely, to extend the formulas with new constructors and new types of variables and expressions. We present the extensions to the logic along with the extensions to the language.

We also show a group of extensions independent of the language used for intermediate representation of programs. These extensions change the language used for representing invariants and also require changes in the VCG. We introduce these extensions to make it easier to express the results of compiler analyses. The changes to the language for invariants are therefore primarily for standard invariants, but the changes also propagate to simulation invariants. It is important to point out that these changes are not fundamental to simulation invariants. As mentioned earlier, simulation invariants are the crucial concept for our approach to credible compilation. The fact that simulation invariants basically remain the same, in spite of changes to the programming language and standard invariants, supports our belief that simulation invariants are essential for credible compilation in general.

Finally, we present some limitations of the current framework for credible compilation. In particular, the framework does not support translations from one program representation to essentially different representations. Such translation can involve both changing the code and the data representation. Additionally, the framework currently does not support changes in the data layout even within the same representation. We believe that more general simulation invariants, together with the related changes in the SimVC, could enable the framework to support some of these changes.

6.1 Language Extensions

In this section we consider how the framework could handle several extensions to BL. We start by adding C-like pointers to BL. BL is obtained by adding procedures to the initial language for credible compilation. Even before adding procedures, we explored [42] adding C-like pointers to the initial language. It may look surprising that we extend the language first with such low-level constructs as pointers to arbitrary memory locations. We first present the motivation and then formalize in detail changes to the language, the logic, and the VCG.

We next briefly discuss how to add arrays to the language. Array-bounds checking emphasizes the necessity of a safe language semantics. We then discuss how to extend the language semantics and the VCG to include error states. Finally, we consider extending the language with some common constructs from imperative programming languages, such as expressions with side effects and computed gotos.

6.1.1 Pointers

Before we present the extensions to the language syntax and semantics, we discuss why we first add arbitrary C-like pointers to the language. Arbitrary pointers are clearly necessary if the credible compiler has to generate code for unsafe programming languages like C. They are, however, necessary even for more disciplined languages that provide a safe high-level memory model, e.g., the Java programming language [8]. Compilers translate the programs from the high-level representation into a lower-level representation before performing most of the optimizations. The translations introduce pointers in their full generality, so that the transformations can optimize the way in which the resulting code accesses memory. Therefore, a framework that allows the compiler to prove results of such transformations has to support pointers.

Pointers are also essential for an intermediate representation that describes assembly code. Pointers, or their equivalents, are used to represent memory reads and writes, and special global variables (that pointers cannot point to) are used to represent machine registers. Finally, we added pointers to the initial language for credible compilation to find out how the compiler could prove that the results of pointer analyses are correct. In particular, we developed a framework [42] which allows the compiler to prove that the results of flow-insensitive pointer analyses [2, 44] are correct. We discovered that flow-insensitive analyses require a language semantics with certain guarantees; we present details in Section 6.2.4.

We next extend the language syntax and semantics with pointers. Figure 6-1 shows the modified program syntax. There are three new forms for program expressions: $\texttt{\&}I$ denotes the address of the variable named I, $*E$ denotes the value stored at memory location E, and `NULL` denotes a special address that is different than possible addresses of the variables. We introduce a new syntactic domain for left-hand expressions which have two forms: a variable or a dereference of an expression. The assignment nodes are changed so that the left-hand side is W instead of I. We use $\texttt{\&}W$ to represent the syntactic elimination of one level of dereferencing: if $W \equiv I$, then $\texttt{\&}W \equiv \texttt{\&}I$; and if $W \equiv *E$, then we define $\texttt{\&}W \equiv E$.

SyntacticDomains :
$E \in$ Expression
$W \in$ L-expression
$\ldots \in$ the same as in BL, Figure 4-1

ProductionRules :

$E \equiv$	$\ldots$	[the same as in BL]
	$\mid \texttt{\&}I$	[Address-of Operation]
	$\mid *E_{deref}$	[Dereference]
	$\mid$ `NULL`	[Null Pointer Constant]
$W \equiv$	I	[Variable Location]
	$\mid *E_{deref}$	[Expression Location]
$N \equiv$	L:W=E	[Assignment Node]
	$\mid \ldots$	[the rest the same as in BL]

Figure 6-1: Extensions to the Abstract Syntax of BL

Note that the syntax allows expressing arbitrary pointer operations that involve arithmetic and use integers directly as pointers, e.g., `*(p+1)=*(q+i)-*8`. There are no types; pointers and integers are syntactically used in the same way. This is (modulo type casting) the way pointers are used in C. Additionally, pointers are semantically equivalent to integers in C. It is undecidable, in general, whether a C program is type and memory safe. We initially wanted to develop a framework in which the compiler can prove its transformation correct even if the input program is an arbitrary C program without any safety guarantees. However, trying to prove anything about the full C language leads to numerous technical problems. Therefore, we consider only a safer subset of C. In particular, we do not allow programs that apply the address-of operator to a local variable. We still allow completely arbitrary pointers to the global variables, more precisely, to the global memory.

We next describe how to modify the operational semantics of BL (Section 4.1.2) to add arbitrary pointers to non-local variables. The main change is to separate the memory into two parts: the local memory and the global memory. (We discuss in next sections how this corresponds to realistic machines.) The local variables and procedure parameters are stored in the local memory. The local memory represents (data) stack; we disallow pointers in the stack, and thus aliasing of the local variables. The global variables are stored in the global memory. The global memory represents static data and can also represent the heap, i.e., dynamically allocated memory.

Formally, we first change the domains of the operational semantics and the allocation functions in the following way:

$$\begin{aligned}
&\text{One-Memory} = \text{Address} \to \text{Value}\\
m \in\ &\text{Memory} = \text{One-Memory} \times \text{One-Memory}\\
&\text{One-Alloc-Pointer} = \text{Address}\\
p \in\ &\text{Alloc-Pointer} = \text{One-Alloc-Pointer} \times \text{One-Alloc-Pointer}
\end{aligned}$$

$$\begin{aligned}
&\text{alloc-init}^l(\langle m, a, p\rangle, I, Z) = \langle\langle m^g, m^l[p^l \mapsto Z]\rangle, \langle a^g, a^l[I \mapsto p]\rangle, \langle p^g, p^l + 1\rangle\rangle\\
&\text{alloc-init}^g(\langle m, a, p\rangle, I, Z) = \langle\langle m^g[p^g \mapsto Z], m^l\rangle, \langle a^g[I \mapsto p], a^l\rangle, \langle p^g + 1, p^l\rangle\rangle.
\end{aligned}$$

The Memory domain is now a pair that separately represents the global and local memories. We use m^g and m^l to denote the components of a pair m. Similarly, each memory has a separate allocation pointer p^g and p^l. For the local memory, p^l is the stack pointer; for the global memory, p^g is the heap pointer. The allocation functions for local and global variables change the respective memories, environments, and pointers.

The input and output functions of the operational semantics remain the same as for BL; the only requirement is that the initial value of the p^g pointer be greater than `0`. The expression evaluation is also similar as in BL; only the evaluation of a variable slightly changes as shown in Figure 6-2. We use helper function var-in-loc-env(W, a^l) that returns true if the expression $W \equiv I$, for some I, and the environment a^l maps the variable I; otherwise, var-in-loc-env(W, a^l) returns false. Figure 6-2 also shows the evaluation of the new forms of expressions.

$$\begin{aligned}
&m(a(I)) = \textbf{if } \text{var-in-loc-env}(I, a^l) \textbf{ then } m^l(a^l(I)) \textbf{ else } m^g(a^g(I)) \textbf{ fi}\\
&m(a(\texttt{\&}I)) = a(I) = \textbf{if } \text{var-in-loc-env}(I, a^l) \textbf{ then } a^l(I) \textbf{ else } a^g(I) \textbf{ fi}\\
&m(a(\texttt{*}E_{deref})) = m^g(m(a(E_{deref})))\\
&m(a(\texttt{NULL})) = \texttt{0}
\end{aligned}$$

Figure 6-2: Extensions to the BL Expression Evaluation

The rewrite rules remain the same as for BL, except that the rule for assignments is replaced with two new rules. Figure 6-3 shows the new rules for an assignment to a local variable and an assignment to the global memory. Note that W can be of the form $\texttt{*}E_{deref}$, where E_{deref} is an arbitrary expression that can evaluate to an address different than the addresses of the global variables.

$$
\begin{array}{ll}
\langle L, m, a, p, h, P\rangle \to \langle L +_P 1, \langle m^g, m^l[V \mapsto V']\rangle, a, p, h, P\rangle & [\mathit{assign\text{-}loc}] \\
\qquad \text{where } P(L) \equiv L\!:\!W\texttt{=}E & \\
\qquad \text{and var-in-loc-env}(W, a^l) & \\
\qquad \text{and } V \equiv m(a(\texttt{\&}W)) \text{ and } V' \equiv m(a(E)) & \\
& \\
\langle L, m, a, p, h, P\rangle \to \langle L +_P 1, \langle m^g[V \mapsto V'], m^l\rangle, a, p, h, P\rangle & [\mathit{assign\text{-}glob}] \\
\qquad \text{where } P(L) \equiv L\!:\!W\texttt{=}E & \\
\qquad \text{and not var-in-loc-env}(W, a^l) & \\
\qquad \text{and } V \equiv m(a(\texttt{\&}W)) \text{ and } V' \equiv m(a(E)) &
\end{array}
$$

Figure 6-3: Extensions to the BL Operational Semantics Rewrite Rules

Logic Extensions for Pointers

We next describe how to extend the logic after we add pointers to the language. There are several complications in obtaining an effective logic when dealing with pointers in the intermediate representation. These complications stem from the possibility that an assignment via a pointer may change a variable that is not syntactically visible in the assignment statement. The solution is to use logic expressions with *explicit memory*; these expressions are different from the program expressions where the memory is implicit.[1] We extend the logic with a new group of expressions that explicitly represent memory. In our presentation, we follow Necula [35, page 64], who attributes the rules related to these expressions to McCarthy [29].

We first illustrate the main difference between the expressions with explicit and implicit memory using an example. Consider the assignment `i=j` in a program with global variables `i` and `j`. No memory m^g is syntactically visible in the expressions `i` and `j`. However, the meaning of the assignment is to read (select) the value of variable `j` from the memory m^g, more precisely from the memory address to which the current environment maps `j`, and to write (update) that value to the location in memory m^g to which the current environment maps `i`.

In the logic, we denote the addresses of the variables `i` and `j` as `addr(i)` and `addr(j)`. Suppose that the logic variable x^m represents the state of the global memory before the assignment `i=j`. The logic expression `sel(`x^m`,addr(j))` denotes the value of variable `j` in memory x^m. The logic expression `upd(`x^m`,addr(i),sel(`x^m`,addr(j)))` denotes the memory whose locations have the same values as in x^m, except that the location with address `addr(i)` has value `sel(`x^m`,addr(j))`. Therefore, the expression with `upd` represents the memory after the assignment `i=j`.

We proceed to formally describe the logic that includes the new expressions with memory. We first present the syntax of the new expressions. We next describe how to

[1] We initially used approximately the same syntax and semantics for the logic expressions as for the program expressions, and we devised special rules for substitution [42] to model assignments in the presence of pointers. However, those rules require the compiler to provide pointer-analysis results and to guide the use of the results in the VCG; otherwise, the VC can get exponentially large.

represent the local and global variables with those expressions. The new expressions necessitate changes to the functions for translating the program expressions to the logic expressions. We present the new translation functions, and then the semantics of the new logic expressions, as well as the proof rules for those expressions.

Figure 6-4 shows the modified logic syntax. We use G^m to range over memory expressions that have four forms: a logic variable x^m denotes some memory, the constant $\mathtt{mem}_0$ denotes the memory that maps all locations to 0, an expression $\mathtt{upd}(G^m, G^i_a, G^i_v)$ denotes the result of updating the memory G^m at location G^i_a with value G^i_v, and an expression M denotes a program memory. Observe that the new logic has two types of logic variables: integer logic variables and memory logic variables. We assume that the quantified formulas are extended so that they can quantify over both types of variables, and additionally that there are formula constructors for equality and inequality of memory expressions. There are also two new forms for integer logic expressions: $\mathtt{sel}(G^m, G^i_a)$ denotes the result of selecting the value at location G^i_a in the memory G^m and $H(I)$ denotes the value or the address of a program variable.

SyntacticDomains :

$H \in$ Program-variable $= \{\mathtt{val}, \mathtt{val}_1, \mathtt{val}_2\} \cup \{\mathtt{addr}, \mathtt{addr}_1, \mathtt{addr}_2\}$
$M \in$ Program-memory $= \{\mathtt{mem}, \mathtt{mem}_1, \mathtt{mem}_2\}$
$G^i \in$ Integer-expression
$G^m \in$ Memory-expression
$\ldots \in$ the rest the same as in Figure 4-4

ProductionRules :

$G^i \equiv \ldots$	[the rest the same as in Figure 4-4]
$\mid H(I)$	[Program Variable Value or Address]
$\mid x^i$	[Integer Logic Variable]
$\mid \mathtt{sel}(G^m, G^i_a)$	[Memory Read]
$G^m \equiv x^m$	[Memory Logic Variable]
$\mid \mathtt{mem}_0$	[Memory Constant]
$\mid \mathtt{upd}(G^m, G^i_a, G^i_v)$	[Memory Write]
$\mid M$	[Program Memory]

Figure 6-4: Extensions to the Abstract Syntax of the Logic Formulas

We next describe how to represent the program variables and the memory in the logic formulas. The introduction of pointers in the program expressions and the introduction of the memory expressions in the logic necessitate a change in the representation of the program variables in the logic. In the logic in the basic framework, the expression $H(I)$ always represents the value of the program variable I; depending on the particular constructor H, the variable is local or global, from one program or from one of the two programs. In the extended logic, we represent the local variables differently than the global variables: the expression $\mathtt{val}(I_l)$ represents the *value* of the *local* variable[2] I_l and the expression $\texttt{addr}(I_g)$ represents the *address* of the *global* variable I_g. There are still two groups of formulas: the formulas for one program (the analysis results) that can contain only the constructors `val` and `addr` and the formulas for two programs (the simulation relationships) that can contain only the indexed constructors $\texttt{val}_{1/2}$ and $\texttt{addr}_{1/2}$.

The logic formulas represent the (standard and simulation) invariants and the verification conditions. In the verification conditions, the memory is denoted with arbitrary memory logic expressions. In the invariants, we want to refer, at different program points, to a particular memory that is the global memory during the program execution. We use the constructor `mem` to denote the state of the global memory in the standard invariants. For instance, to represent that global pointer `p` points to global variable `x` at some node `3`, we write `3:inv sel(mem,addr(p)) = addr(x)`. We use indexed versions $\texttt{mem}_1$ and $\texttt{mem}_2$ to represent the states of the global memories of two programs in the simulation invariants. For instance, to represent that the global variable $\texttt{g}_1$ in program 1 at node $\texttt{3}_1$ has the same value as the variable $\texttt{g}_2$ in program 2 at node $\texttt{3}_2$, we write $\texttt{3}_1\texttt{,3}_2\texttt{:sim-inv sel(mem}_1\texttt{,addr}_1\texttt{(g))} = \texttt{sel(mem}_2\texttt{,addr}_2\texttt{(g))}$.

We also present several other examples of the formulas used in the simulation invariants. To represent that the two programs have exactly the same memories, we write $\texttt{mem}_1 = \texttt{mem}_2$. If the memories have the same values at all locations except for, say, the location with the address of $\texttt{g}_1$, we can write[3]:

$$\forall x.\ x \neq \texttt{addr}_1\texttt{(g)} \Rightarrow \texttt{sel(mem}_1\texttt{,}x\texttt{)} = \texttt{sel(mem}_2\texttt{,}x\texttt{)}.$$

We can also represent this without the universal quantification:

$$\texttt{mem}_1 = \texttt{upd(mem}_2\texttt{,addr}_1\texttt{(g),sel(mem}_1\texttt{,addr}_1\texttt{(g)))}.$$

In general, the memories can have different values for a set of locations. (We briefly discuss extending the logic formulas with sets in Section 6.2.3.) The compiler can generate arbitrary invariants as long as it can prove the verification conditions generated for those invariants.

We next describe the functions for translating the program expressions with implicit memory to the logic expressions with explicit memory. In the logic in the basic framework, the translation functions take a symbolic environment that maps variable names to the appropriate logic expressions for the values of those variables. In the extended logic, the translation functions take a symbolic environment e that maps each local variable name to the logic expression for the value of that variable and each global variable name to the logic expression for the address of that variable. (We use

[2] The constructor `val` is the same as `loc` that we use in the basic framework for local variables. We avoid the name `loc` because it might be misinterpreted as "location." That is, indeed, the name that McCarthy and Painter use for locations in the first published paper on compiler correctness [29]. They use a binary constructor that takes the name of the variable and the environment. We do not need to represent the environment explicitly in the verification conditions.

[3] Note that x is an integer logic variable in this example. We omit the explicit typing of the variables and expressions when it is possible to infer the types.

the function var-in-loc-env(I, e) to test if the variable I is local in the environment e.) Additionally, the new translation functions take an expression G^m that represents the logic expression to use for the memory.

Figure 6-5 shows the new translation functions. The translation of the address-of operation is the expression representing the address of the variable in the symbolic environment. (This translation is used only for global variables.) The translation of the dereference operation first translates the program expression E_{deref} to the logic expression G, and then generates the expression representing a read from location G of the symbolic memory G^m. Similarly, a global variable reference is translated into the expression representing a read from the symbolic memory at the address of the variable in the symbolic environment. The other translations are done as before, by structural induction on the program expressions, passing the symbolic memory and environment to the translations of the subexpressions.

```
$\text{translate-type}(\texttt{\&}I, G^m, e) = \langle e(I), \texttt{int}\rangle$
$\text{translate-type}(\texttt{*}E_{deref}, G^m, e) =$
  $\textbf{let } G \textbf{ be } \text{to-type}(\text{translate-type}(E_{deref}, G^m, e), \texttt{int}) \textbf{ in}$
    $\langle \texttt{sel(}G^m\texttt{,}G\texttt{)}, \texttt{int}\rangle$
$\text{translate-type}(\texttt{NULL}, G^m, e) = \langle 0, \texttt{int}\rangle$
$\text{translate-type}(I, G^m, e) =$
  $\textbf{if } \text{var-in-loc-env}(I, e) \textbf{ then } \langle e(I), \texttt{int}\rangle \textbf{ else } \langle \texttt{sel(}G^m\texttt{,}e(I)\texttt{)}, \texttt{int}\rangle \textbf{ fi}$
$\text{translate-type}(\ldots, G^m, e) = \text{the same as in Figure 4-5}$
```

Figure 6-5: Extensions to the Functions for Translating Program Expressions to Logic Expressions

We next define the semantics of the new logic expressions. Figure 6-6 shows the modifications to the basic logic. We add the domain *Store* to the domains for the basic logic (Figure 4-6). The new domain represents memories, i.e., functions from addresses to values. We also use the new valuation function $\mathcal{G}^m$ for the meaning of the memory expressions. The meaning of the $\texttt{mem}_0$ is the constant function 0. The meaning of $\texttt{upd(}G^m\texttt{,}G^i_a\texttt{,}G^i_v\texttt{)}$ is the meaning of G^m, which is a function, σ, with a change that the meaning of G^i_a, z_a, is mapped to the meaning of G^i_v, z_v:

$$\sigma[z_a \mapsto z_v] = \lambda z.\ \textbf{if } z = z_a \textbf{ then } z_v \textbf{ else } \sigma(z) \textbf{ fi}.$$

The meaning of $\texttt{sel(}G^m\texttt{,}G^i_a\texttt{)}$ is the application of the meaning of G^m, σ, to the meaning of G^i_a, z_a: $\sigma(z_a)$. We define the meaning of the formulas that quantify the memory variables in the same (informal) way as the meaning of the formulas that quantify the integer variables. Finally, program expressions (values, addresses, and memories) get the meaning from the context c that consists of one or two concrete memory-environment pairs m, a.

We finally present a set of proof rules for the new logic expressions. For the

$\sigma \in \mathit{Store} = \mathit{Int} \rightarrow \mathit{Int}$

$\mathcal{G}^m$: Memory-expression $\rightarrow \mathit{Context} \rightarrow \mathit{Store}$

$\mathcal{G}^m[\![\mathtt{mem}_0]\!]\ c = \lambda z.\ 0$
$\mathcal{G}^m[\![\mathtt{upd(}G^m\mathtt{,}G^i_a\mathtt{,}G^i_v\mathtt{)}]\!]\ c = (\mathcal{G}^m[\![G^m]\!]\ c)[(\mathcal{G}^i[\![G^i_a]\!]\ c) \mapsto (\mathcal{G}^i[\![G^i_v]\!]\ c)]$

$\mathcal{G}^i[\![\mathtt{sel(}G^m\mathtt{,}G^i_a\mathtt{)}]\!]\ c = (\mathcal{G}^m[\![G^m]\!]\ c)(\mathcal{G}^i[\![G^i_a]\!]\ c)$
$\mathcal{G}^i[\![\ldots]\!]\ c =$ the same as in Figure 4-8

$c \models \forall x^m.\ F'$ iff $c \models F'[\sigma/x^m]$ for all $\sigma \in \mathit{Store}$
$c \models \exists x^m.\ F'$ iff $c \models F'[\sigma/x^m]$ for some $\sigma \in \mathit{Store}$
$c \models \ldots$ iff the rest the same as in Figure 4-8

$\mathcal{G}^i[\![\mathtt{val(}I\mathtt{)}]\!]\ c = m^l(a^l(I))$
$\mathcal{G}^i[\![\mathtt{addr(}I\mathtt{)}]\!]\ c = a^g(I)$
$\mathcal{G}^i[\![\mathtt{val}_{1/2}\mathtt{(}I\mathtt{)}]\!]\ c_1, c_2 = m^l_{1/2}(a^l_{1/2}(I))$
$\mathcal{G}^i[\![\mathtt{addr}_{1/2}\mathtt{(}I\mathtt{)}]\!]\ c_1, c_2 = a^g_{1/2}(I)$
$\mathcal{G}^i[\![\mathtt{mem}]\!]\ c = m^g$
$\mathcal{G}^i[\![\mathtt{mem}_{1/2}]\!]\ c_1, c_2 = m^g_{1/2}$

Figure 6-6: Valuation Functions for Expressions and Validity of Formulas

memory expressions, we use the following two rules, called the McCarthy rules:

$$\frac{}{\vdash \mathtt{sel(upd(}G^m\mathtt{,}G^i_a\mathtt{,}G^i_v\mathtt{),}G^i_a\mathtt{)} = G^i_v} \qquad [\mathit{Alias}]$$

$$\frac{\vdash G^i_a \neq G^i_b}{\vdash \mathtt{sel(upd(}G^m\mathtt{,}G^i_a\mathtt{,}G^i_v\mathtt{),}G^i_b\mathtt{)} = \mathtt{sel(}G^m\mathtt{,}G^i_b\mathtt{)}} \qquad [\mathit{Non\text{-}Alias}]$$

The [*Alias*] rule states that a read from the memory location with address G^i_a returns value G^i_v that has been written to that location. The [*Non-Alias*] rule states that a read from the memory location with address G^i_b returns the value that does not depend on the writes to other memory locations.

We next present several rules that involve the program variable names, i.e., identifiers. In BL, the identifiers cannot appear in the verification conditions, because the VCG for BL substitutes the program variables with the logic expressions that represent their values. Therefore, in the logic for BL, there is no need to have proof rules that involve identifiers. However, we later show that the identifiers can appear in the verification conditions for the extended language. In particular, the identifiers of the global program variables appear in the address-of expressions constructed with `addr`.

We introduce the following rules for expressions representing addresses of variables:

$$\frac{\vdash I_1 \equiv I_2}{\vdash \texttt{addr}(I_1) = \texttt{addr}(I_2)} \qquad [\textit{Same-Id}]$$

$$\frac{\vdash I_1 \not\equiv I_2}{\vdash \texttt{addr}(I_1) \neq \texttt{addr}(I_2)} \qquad [\textit{Diff-Id}]$$

$$\frac{}{\vdash \texttt{addr}(I) \neq 0} \qquad [\textit{Non-Null}]$$

$$\frac{}{\vdash \texttt{addr}_1(I) = \texttt{addr}_2(I)} \qquad [\textit{Same-Env}]$$

The [*Same-Id*] rule is an instantiation of the general congruence rule for equality. We present the rule explicitly because it involves the syntactic equality of the program identifiers. This requires that a machine-verifiable representation of the proofs have support for the program identifiers.[4] The [*Diff-Id*] rule states that the environments are injective—they map different identifiers to different memory locations. The [*Non-Null*] rule states that the environments do not map any identifier to the value of `NULL`. For each of the rules [*Same-Id*], [*Diff-Id*], and [*Non-Null*], there are two analogous rules for the indexed versions of the constructor `addr`. Finally, the [*Same-Env*] rule states that the environments from the two programs map identical identifiers to the same memory locations. Therefore, we abbreviate all $\texttt{addr}_i(I)$ expressions to $\texttt{\&}I$ in the rest of the text. We discuss the relationship between the addresses of the variables from two programs in the next section.

Verification-Condition Generator Extensions for Pointers

Since we extended the language and the logic, we also need to extend the VCG. We first describe the small changes to the (standard and simulation) contexts and invariants, and the analysis and transformation correctness requirements. We then present the extensions to the StdVCG and SimVCG algorithms for BL. We also show an example of verification condition generated by the new algorithms.

The (standard and simulation) contexts remain the same; they consist of an input context, an output context, a set of invariants, and the other additional information. The input and output contexts and the invariants are similar as the contexts and invariants in BL. The only change is that these formulas are now from the extended logic with explicit memory. The analysis and the transformation requirements are

[4] It is not strictly necessary to use the identifiers. Instead, we can use (distinct) integer constants to represent (distinct) identifiers from some lexical scope. That, in turn, requires encoding integer literals and proofs involving them in a machine-verifiable form. The Athena framework, which we use for proof representation and verification, has a built-in support for both integer literals and object-level identifiers. We use identifiers as they allow a better readability of verification conditions.

also similar as in BL. The VCG generates StdVC and SimVC, and the compiler has to prove that those formulas hold. The only difference is in the simulation context for the starting procedures of the two programs.

The compiler has to prove that the two programs generate the same output given the same input. For BL, the input consists of the values of the global variables and the parameters of the starting procedures, and the output consists of the values of the global variables. For the extended language, instead of the values of the global variables, we use the entire global memories. To prove that $Q_1 \rhd Q_2$, the compiler has to prove the simulation context whose input context is:

$$J^{in} \equiv \texttt{mem}_1 = \texttt{mem}_2 \wedge \bigwedge \texttt{val}_1\texttt{(}I_1^p\texttt{)} = \texttt{val}_2\texttt{(}I_2^p\texttt{)},$$

for all parameters I_1^p of the starting procedure of Q_1 and the corresponding parameters I_2^p of the starting procedure of Q_2, and the output context is:

$$J^{out} \equiv \texttt{mem}_1 = \texttt{mem}_2.$$

The compiler can additionally assume that the two programs have the same allocation of global variables; the corresponding global addresses have the same addresses and thus $a_1^g = a_2^g$. (This does not allow the compiler to prove that changes in the data layout are correct.) We need to use the same addresses in both programs and the whole memories in the simulation requirement because we allow arbitrary pointers. For instance, consider a program that only increments a global variable `g` with the value of some location with address `8`: `g=g+*8`. Even if `g` has the same value at the beginning of the two programs, it would not have the same value at the end unless the value at location `8` is the same. Additionally, if `g` were mapped to different addresses in the two programs, and one of the addresses happened to be `8`, the value of `g` would not be the same at the end of the two programs. Therefore, the compiler can use the rule $\vdash \texttt{addr}_1\texttt{(}I\texttt{)} = \texttt{addr}_2\texttt{(}I\texttt{)}$ in the proof, and $\texttt{mem}_1 = \texttt{mem}_2$ in J^{in}.

We next present changes to the StdVCG (Figure 4-9) and the SimVCG (Figure 4-11) for BL. The StdVCG and SimVCG for the extended language also symbolically execute procedures and generate the appropriate StdVC and SimVC. However, a symbolic execution operates on a different symbolic state than the symbolic execution of BL procedures. A symbolic state s for the StdVCG now maps $\texttt{val(}I_l\texttt{)}$, for each local variable I_l, to an integer logic expression representing the value of the variable, as for BL, but s also maps `mem` to a memory logic expression representing the global memory. Analogously, states $s_{1/2}$ for the SimVCG map $\texttt{val}_{1/2}\texttt{(}I_{1/2}^*\texttt{)}$ and $\texttt{mem}_{1/2}$.

We next describe changes to the helper functions for the StdVCG and SimVCG. The function fresh-sym-state returns fresh integer logic variables for all $\texttt{val(}I_l\texttt{)}$ and a fresh memory logic variable for `mem`. The function fresh-globals remaps only `mem` to a fresh memory logic variable. The substitution of the symbolic state in formulas is the substitution of the expressions G^{i*} for $\texttt{val(}I_l^*\texttt{)}$ and the expression G^m for `mem`: $\text{subst}(F, s) = F[G^{i*}/\texttt{val(}I_l^*\texttt{)}][G^m/\texttt{mem}]$. The functions for the SimVCG change in a similar way, taking into account the special form of logic formulas J that express the related variables in two programs. (We also allow the pair $\texttt{mem}_1, \texttt{mem}_2$ to appear in the sequence of pairs of related variables.)

Both StdVCG and SimVCG use the same helper functions for the assignment, branch, and call nodes. Figure 6-7 shows the new functions for the extended language. Compared to BL, the new functions have an extra argument M that represents the memory to use for the translations. The modified StdVCG calls these functions with `mem` for M, and the modified SimVCG calls with `mem`$_1$ or `mem`$_2$ depending on the program. In practice, the translations of the expressions from the program form to the logic form are done while preparing procedures for symbolic executions (both for BL and the extended language), i.e., while merging the invariants. The executions then perform only the substitutions in the current symbolic states.

```
translate-assign($W, E, M, s, e$) =
  let $G_E$ be translate($E, M, e$) in
    if var-in-loc-env($W, e$) then
      $s$[translate($I, e$) $\mapsto$ subst($G_E, s$)]
    else
      let $G_W$ be translate(&$W, M, e$) in
        $s$[$M$ $\mapsto$ subst(upd($M$,$G_W$,$G_E$), $s$)]
    fi
translate-branch($E, M, s, e$) = subst(translate-bool($E, M, e$), $s$)
translate-call($E^*, M, s, e$) = subst-seq(translate-seq($E^*, M, e$), $s$)
```

Figure 6-7: Changes to the Helper Functions for Verification-Condition Generators

We next show the verification conditions that the modified VCG generates for an example slightly changed from the example presented in Chapter 3. We change the original procedure from Figure 3-2 in the following way: there is a new global variable `h`, there is a new node `0:h=&g` before node `1:i=0`, and the node with label `3` is `3:*h=g+2*i`. We consider that the compiler can perform constant propagation on the input procedure and can transform the nodes `3:*h=g+2*i` and `5:i=i+c` so that the output procedure has nodes `3:g=g+2*i` and `5:i=i+3`. We first describe the analysis results that the compiler generates and the StdVC for those results. We then describe the simulation relationships and the SimVC for this example.

The compiler first performs a pointer analysis on the original program. Consider that the procedure `q` is such that no execution of `q` changes the pointer `h` when `h` points to `g`. The compiler can then generate and prove the standard context for `q` with both input and output contexts being `sel(mem,&h)` $=$ `&g`. For the procedure `p`, we consider the standard context with both input and output contexts being `true`. Suppose that the compiler generates only one standard invariant for the analysis results of this context: `3:inv sel(mem,&h)` $=$ `&g` $\land$ `val(c)` $= 3$. For this example context of p, the StdVCG generates the following StdVC:

$$\begin{array}{l}
\forall i^1, c^1, m^1.\ \texttt{true}^{p_{in}} \Rightarrow \\
\quad \texttt{sel(upd(}m^1\texttt{,\&h,\&g),\&h)} = \texttt{\&g} \wedge 3 = 3 \wedge \\
\quad \forall i^2, c^2, m^2.\ \texttt{sel(}m^2\texttt{,\&h)} = \texttt{\&g} \wedge c^2 = 3 \Rightarrow \\
\qquad \texttt{sel(upd(}m^2\texttt{,sel(}m^2\texttt{,\&h),sel(}m^2\texttt{,\&g)} + 2 * i^2\texttt{),\&h)} = \texttt{\&g} \wedge \\
\qquad \forall m^3.\ \texttt{sel(}m^3\texttt{,\&h)} = \texttt{\&g} \Rightarrow \\
\qquad\quad (i^2 + c^2 < 24 \Rightarrow \texttt{sel(}m^3\texttt{,\&h)} = \texttt{\&g} \wedge c^2 = 3) \wedge \\
\qquad\quad (\neg(i^2 + c^2 < 24) \Rightarrow \texttt{true}^{p_{out}}).
\end{array}$$

We next consider, for the original and transformed procedures p, a simulation context with both simulation input and output contexts being just $\texttt{mem}_1 = \texttt{mem}_2$. We use the same simulation input and output contexts for the calls to procedure q. Suppose that the compiler generates only one simulation invariant for the context for procedures p: $\texttt{3}_1\texttt{,3}_2\texttt{:sim-inv mem}_1 = \texttt{mem}_2 \wedge \texttt{val}_1\texttt{(i)} = \texttt{val}_2\texttt{(i)}$. Also, the compiler generates that the initial values for local variables $\texttt{i}_2$ and $\texttt{c}_2$ are the same as for $\texttt{i}_1$ and $\texttt{c}_1$. In this example, the SimVCG generates the following SimVC:

$$\begin{array}{l}
\forall m_1^1, i_1^1, c_1^1, m_2^1.\ m_1^1 = m_2^1 \Rightarrow \\
\quad m_1^1 = m_2^1 \wedge 0 = 0 \wedge \forall m_1^2, i_1^2, c_1^2, m_2^2, i_2^1, c_2^1.\ m_1^2 = m_2^2 \wedge i_1^2 = i_2^2 \Rightarrow c_2^2 = 3 \Rightarrow \\
\qquad \texttt{upd(}m_1^2\texttt{,sel(}m_1^2\texttt{,\&}_1\texttt{h),sel(}m_1^2\texttt{,\&}_1\texttt{g)} + 2 * i_1^2\texttt{)} = \\
\qquad \texttt{upd(}m_2^2\texttt{,sel(}m_2^2\texttt{,\&}_2\texttt{h),sel(}m_2^2\texttt{,\&}_2\texttt{g)} + 2 * i_2^1\texttt{)} \wedge \forall m_1^3, m_2^3.\ m_1^3 = m_2^3 \Rightarrow \\
\qquad\ (i_1^2 + 3 < 24 \Rightarrow i_2^1 + c_2^1 < 24 \wedge m_1^3 = m_2^3 \wedge i_1^2 = i_2^1) \wedge \\
\qquad\ (\neg(i_1^2 + 3 < 24) \Rightarrow \neg(i_2^1 + c_2^1 < 24) \wedge m_1^3 = m_2^3).
\end{array}$$

Observe that the above example does not show the use of the formulas J with a sequence of related variables. Using these formulas, the input and output contexts are just $\texttt{true,(mem}_1\texttt{,mem}_2\texttt{)}$ (i.e., $\texttt{true} \wedge \texttt{mem}_1 = \texttt{mem}_2$), and the simulation invariant is $\texttt{3}_1\texttt{,3}_2\texttt{:sim-inv true,(mem}_1\texttt{,mem}_2\texttt{),(val}_1\texttt{(i),val}_2\texttt{(i))}$. In this case, the SimVC is much shorter[5]:

$$\begin{array}{l}
\forall m^1, i^1, c_1^1.\ \texttt{true} \Rightarrow \\
\quad \texttt{true} \wedge \forall m^2, i^2, c_1^2, c_2^1.\ \texttt{true} \Rightarrow c_2^1 = 3 \Rightarrow \\
\qquad \texttt{true} \wedge \forall m^3.\ \texttt{true} \Rightarrow \\
\qquad\quad (i^2 + 3 < 24 \Rightarrow i^2 + c_2^1 < 24 \wedge \texttt{true}) \wedge \\
\qquad\quad (\neg(i^2 + 3 < 24) \Rightarrow \neg(i^2 + c_2^1 < 24) \wedge \texttt{true}).
\end{array}$$

6.1.2 Arrays

We next briefly show how to add (static) arrays to the BL with pointers. In the simplest case, we add a new declaration form for (one-dimensional) arrays, $I[Z]$. The allocation of array $I[Z]$ takes Z consecutive locations in the memory and the environment maps I to the address of the first location. We also add a new expression form for array accesses, $W[E_{index}]$, both to the left-hand expressions W and to the

[5] In practice, the SimVCG does not even generate true in $\texttt{true} \Rightarrow F$ or $\texttt{true} \wedge F$ or $F \wedge \texttt{true}$.

right-hand expressions E. We allow only global variables to be arrays that are indexed with arbitrary expressions. Since the language has pointer arithmetic, we can consider array expressions simply as syntactic sugar: $W[E] \equiv$ `*(&`$W + E$`)`. This concludes the extensions for arrays with arbitrary indices.

We also point out a useful restricted form of arrays. Namely, if the size of arrays is known at compile time and arrays can be indexed only with integer constants, it is trivial to check at compile time that array indices are within bounds. These arrays can be even local, and we use them to model an activation frame containing local variables addressed using the frame pointer. Each array element is, essentially, treated as a separate variable.

We finally argue why it is necessary to have run-time array-bounds checking for general array indices, as well as more restricted pointers, in the language semantics. The main reason is to formally model the execution of programs on realistic machines while still retaining certain guarantees in the language. For example, in the BL with pointers, a pointer expression involving global variables cannot access local variables. However, to guarantee that, the semantics presented in Section 6.1.1 requires the machine to have two separate memories, which is not the case in practice. Running a program on a machine with one memory could generate a different result than running the program on a machine with separate memories. Therefore, to model the execution of programs on realistic machines, we need to change the semantics. We discuss in next section how to introduce error states in the semantics to restrict memory accesses that pointer/array expressions can make.

6.1.3 Error States

We next briefly discuss how we could change the framework to support error states in the language semantics. Error states are added to the semantics as follows. For each group of nodes in the language, the execution checks whether certain error conditions are satisfied. For instance, for the nodes that evaluate expressions, there are checks for whether array indices are out of bounds and for division by zero. If the error conditions are satisfied, the execution goes to an error state; otherwise, the execution continues as normal. Error states are final states of a program execution, and an error is one of the observable results that a program can generate.

We do not formally present error conditions for nodes in the BL with pointers (and arrays). We assume that these conditions provide certain guarantees for the program execution. For example, a read/write of `a[i]` can read/write only an element of the array `a`. More precisely, if the execution (evaluation) of `a[i]` does not end up in an error state (i.e., the index `i` is within the bounds of `a`), then `a[i]` accesses an element of `a`. Note that defining error conditions for arbitrary expressions with pointer arithmetic is much more involved. For the full C language, the standard [26] does not formally present semantics (and error conditions), and the paper [21], which presents formal semantics for C, ignores the issue of errors.

We next consider two approaches that we could use to handle errors in BL extended with pointers: disallowing programs with errors and extending the framework. We would disallow errors by requiring that the compiler input program have no errors, i.e., that the program be such that its execution can never get into an error state. The compiler cannot determine whether this holds for an arbitrary BL program. However, BL is only an intermediate representation in the compiler, and we can restrict the *source language* so that the compiler can effectively check whether a source program has an error. This approach is used in the *certifying compilers* Touchstone [38] and Popcorn [31]. The source language for these compilers is a type-safe subset of C, and each compiler translation (or transformation) preserves type safety and generates evidence that the output program has no errors.[6] To use this approach for credible compilation, we would extend BL with static types. Each compiler transformation would first show that the output program has no errors and then that the output program simulates, as before, the input program.

The other approach to handling errors is to allow the input program to potentially have errors, but extend the framework so that the compiler can prove that it correctly transformed the input program even if it has an error. We extend the definition of the simulation requirement to include errors—program Q_1 simulates program Q_2 iff the following holds: an execution of Q_1 can generate an error only if an execution of Q_2 can generate an error and, as before, an execution of Q_1 can terminate only if an execution of Q_2 can terminate with the same output and an execution of Q_1 does not terminate only if an execution of Q_2 does not terminate. The compiler need prove only that the output program simulates the input program, and it follows that the output program has no errors if the input program has no errors.

We next describe how to extend the SimVCG to generate the SimVC for two programs that can have errors. Conceptually, we introduce, in each procedure from the two programs, a special node that represents an error state. We can then replace each original node that can generate an error, when some error condition C holds, with two new nodes: a branch node, whose condition is C and whose target is the special error node, and a copy of the original node, where the copy now cannot generate an error. After this, we can use, for the new procedures, a similar SimVCG as we use for the language without errors. The only change in the symbolic executions is that both procedures need to simultaneously execute the error nodes (as they simultaneously execute the return and call nodes). This approach to errors allows us to describe the semantics of the language using dynamic types in error conditions. We have started exploring whether it is practical to treat pointers as pairs of a base address and offset. We believe that this would enable easier modeling of some of the "undefined" constructs in the C language.

6.1.4 Side Effects

We next briefly consider extending BL by adding expressions that have side effects. In particular, we discuss how to change the logic and the VCG to support functions.

[6] In Touchstone, the evidence is a proof; in Popcorn, it is type information for the output program. The type system is sound— every well-typed program is guaranteed to have no errors. The evidence is statically checked, and the machine that executes the program does not need to perform run-time checks.

Functions are subroutines that return a result, as opposed to procedures, which only modify the state. To add functions to the language, we add a new expression form for function calls, $I\texttt{(}E^*\texttt{)}$, and we change return nodes to be L`:ret` E. In the language semantics, we need to operationally describe the expression evaluation that now takes many steps, instead of one step as in BL.

We change the VCG similarly; the VCG does not translate in one step the program expressions that contain function calls to logic expressions. Instead, the VCG translates the expressions in several steps and uses a symbolic state that contains a special element that represents the value of the expression. The compiler provides (standard and simulation) contexts for functions, and the VCG uses those contexts for modeling the function calls. As usual, the VCG generates a VC that requires the input context to hold before the call(s) and assumes the output context to hold after the call(s). The symbolic state after the call has a fresh logic variable for the global memory and for the special element representing the value of the expression. In the logic formulas, we only add a new constructor for representing the return value of the function, and the compiler can use the new constructor in output contexts. This concludes the list of sufficient changes to the framework to make it support functions in the language. (We present in next section another approach, extending the logic formulas with expressions that represent calls, that can be used to add support for functions.)

6.1.5 Computed Jumps

We next consider extending BL with jumps that have computed targets. We consider two groups of "computed jumps": "computed jumps to subroutine" (subroutine calls with the subroutine being an arbitrary expression instead of a subroutine identifier) and "indirect jumps" (branch nodes with the label being an arbitrary expression instead of a label identifier).

We first consider changing the calls from direct $I\texttt{(}E^*\texttt{)}$ to indirect $W\texttt{(}E^*\texttt{)}$, where the expression `&`W evaluates to the address of the called procedure. (We present the changes only for procedures as the changes for functions are similar.) We also add the expression form `&`I for taking the address of a procedure named I. For example, the sequence `l=&p;(*l)()` makes a call to a parameterless procedure `p`. The change of the calls in the language requires a change of the VCG. We can change the VCG to support indirect calls using two approaches: extending the description of contexts at call sites or extending the logic formulas.

We extend the description of contexts at call sites by allowing the compiler to generate which procedures might be called. For programs with direct calls, only one procedure can be called at any call site, and the VCG requires the compiler to generate only the index of a callee context to use for the call site. (The StdVCG requires an index of a standard context for one callee, and the SimVCG requires an index of a simulation context for two callees.) The VCG for direct calls uses the input and output contexts, for the specified index, to generate a part of VC that requires the input context to hold before the call and assumes the output context to hold after the call. The VCG for indirect calls generates a VC that additionally requires that before the call, the call expression $\&W$ evaluates to one of the procedures specified by the compiler. The VCG also checks that all those procedure have the context specified by the compiler.

If the compiler cannot determine which procedures might be called at a call site, the compiler uses the "default context." For standard contexts, the default context has both input and output contexts `true`, and this context can be used for any procedure. For simulation contexts, the default context represents that two procedures generate the same output given the same input, i.e., the default simulation context has input context $\mathtt{mem}_1 = \mathtt{mem}_2 \wedge \bigwedge \mathtt{val}_1\mathtt{(}I_1^p\mathtt{)} = \mathtt{val}_2\mathtt{(}I_2^p\mathtt{)}$, where $\bigwedge$ ranges over the parameters of the procedures, and output context $\mathtt{mem}_1 = \mathtt{mem}_2$. This context can be used only for the pairs of procedures for which it holds. Since it holds if the two procedures are identical, the SimVCG would generate the SimVC that requires the (translations of) call expressions to be equal at the two call sites.

The other approach to supporting indirect calls is to extend the logic formulas with expressions that represent procedure calls. If the memory before the call is G^m, then the memory after the call would be $\mathtt{app}_i\mathtt{(}G^m\mathtt{,}G^p\mathtt{,}G^*\mathtt{)}$, where $\mathtt{app}_i$ is a family of memory expression constructors indexed by the number of procedure parameters, G^p is a logic expression representing the procedure that is called, and G^* represents the parameters. The new constructors would be *uninterpreted function symbols* in the logic. (The same result is achieved in the previous approach if the compiler uses "default contexts" at all call sites because the compiler does not perform an interprocedural analysis or transformation.) Extending the logic formulas for indirect calls does not support interprocedural analyses and transformations in a clean manner.[7] Therefore, we prefer extending the compiler description of contexts.

We next consider changing the branches from direct $\mathtt{br(}E\mathtt{)}\,L$ to indirect $\mathtt{br(}E_c\mathtt{)}\,E_t$, where the target E_t evaluates to the label to branch to if the condition E_c evaluates to true. We also change the labels to be integers. For example, the sequence `1:j=1;br(TRUE)j` is an indefinite loop. The change of the branches in the language also requires a change of the VCG. However, we cannot change the symbolic execution of the VCG to support arbitrary indirect jumps. The reason is that the VCG could not decide, in general, where to continue the symbolic execution for the branch-taken path. If we restricted the indirect jumps in some way so that the VCG could determine what *all* possible targets are, then we could simply change the VCG to follow all those paths.

6.2 Invariant Extensions

In this section we discuss several extensions to the language used for representing invariants. An invariant consists of a logic formula and one program label (for a standard invariant) or two program labels (for a simulation invariant). We first extend the

[7] The support can be added by changing the VCG to generate, as assumptions in the VC, formulas that involve $\mathtt{app}_i$ and describe contexts. We do not present more details here, but suffice it to say that this would make the VC much more difficult to prove.

invariants so that the compiler can represent a set of variables that are not modified during a loop. We then extend the logic formulas with some new expression constructors, more specifically, constructors that allow the compiler to refer to the program state at the beginning of an execution and constructors that allow the compiler to more succinctly express the results of analyses and the simulation relationships. We finally extend the representation of the program labels in standard invariants to allow the compiler to more easily express the results of flow-insensitive analyses.

6.2.1 Loop Constants

We next describe how to change standard invariants and the StdVCG to allow the compiler to represent that some variables have constant values during some loop. We model this change after the invariants that Necula and Lee use in their certifying compiler [38].

With the invariants presented so far, the compiler can represent that some variables have the same values for *all* executions that reach a program point. The compiler has to determine these values and to represent them as constants. For example, the invariant `2:inv sel(mem,&g)` $= 1 \land$ `val(i)` $= 0$ represents that the global variable `g` and the local variable `i` have the value 1 and 0, respectively, for all executions that reach the node with label `2`. However, the compiler cannot represent that the value of a variable (in general, the value of an expression) does not change between the *consecutive* executions that reach the invariant. More precisely, we refer to the consecutive executions within the same loop, i.e., the consecutive executions that do not reach any node before the invariant. The value can change for different loops.

We extend the invariants so that each invariant has, beside a label and a formula, also a set of expressions that do not change within the innermost loop containing the invariant. We write such invariant as L`:inv` F`;`G^*. For example, the extended invariant `5:inv val(j)` $>$ `3;sel(mem,&g),val(i)` represents that the value of `j` is greater than 3 for all executions reaching the node `5` and that the values of `g` and `i` are constant within the innermost loop containing the node `5`. Note that the compiler can represent that an expression has a constant value within the loop although the compiler does not determine that value. The compiler can use arbitrary expressions in G^*, e.g., `mem` in G^* represents that the whole memory does not change, and `sel(mem,sel(mem,&p))` in G^* represents that the value of the location pointed to by the pointer `p` does not change.

The extended invariants require changes in the StdVCG for BL (Figure 4-9). Figure 6-8 shows the new symbolic execution of the invariants with expressions G^*. The most common expressions in G^* are the values of variables (`val(`I_l`)` for local I_l and `sel(mem,&`I_g`)` for global I_g) and the value of the whole memory (`mem`). Instead of the helper function fresh-sym-state, the new StdVCG uses the function fresh-sym-state-related(s, G^*) that generates the state s' from s with respect to G^*. For each expression G (be it `val(`I_l`)` or `mem`) that s maps, s' maps G to a fresh logic variable if G is not in G^*, and s' maps G to $s(G)$ if G is in G^*. The extended StdVCG also uses the function seq-eq(G^*, s, s') that generates $\bigwedge \text{subst}(G, s) = \text{subst}(G, s')$, where $\bigwedge$ ranges over all G in G^* for which $\text{subst}(G, s) \not\equiv \text{subst}(G, s')$. The soundness proof for this StdVCG is similar to the soundness proof for the StdVCG for BL; only StdIH3 in the induction hypothesis changes. Necula [35] shows a similar proof in detail.

```
matching $P'(L)$
  ⋮
  $\triangleright$ $L$:inv $F$;$G^*$ $\|$
   if $\text{member}(\langle L, s'\rangle, i)$ then
     $\text{subst}(F, s) \wedge \text{seq-eq}(G^*, s, s')$
   else
     let $\langle s', x^*\rangle$ be $\text{fresh-sym-state-related}(s, G^*)$ in
       $\text{subst}(F, s) \wedge$
       $\forall x^*.\ \text{subst}(F, s') \wedge \text{seq-eq}(G^*, s, s') \Rightarrow \text{Std}(L +_{P'} 1, s', \text{union}(\langle L, s'\rangle, i))$
   fi
endmatching
```

Figure 6-8: Extensions to the Verification-Condition Generator for BL

We have described so far how to extend the standard invariants with sets of expressions that do not change during a loop. An analogous extension can be used for standard contexts for procedures. We can allow the compiler to represent in the input (or the output) context a set of expressions that do not change during the execution of one activation. (Alternatively, the compiler can represent a set of all expressions that the procedure may change, i.e., the write-set of the procedure.) The StdVCG can then generate, at call sites, a symbolic state for the output context with respect to the symbolic state for the input context. In the next section we present another extension that allows the compiler to represent relationships between the states for the input and output contexts.

Finally, the simulation invariants and contexts can be extended in a similar way as the standard invariants and contexts. Each simulation invariant would relate not only the states from two programs, but also different states from one program. The SimVCG would then generate the fresh symbolic states with respect to the old symbolic states.

6.2.2 Starting States in Formulas

We next describe how to extend the logic formulas so that the compiler can represent the starting state in them. (By starting state we mean the state of the program memory at the beginning of the execution of a procedure activation.) Using the logic formulas presented so far, the compiler can represent in standard invariants only the current state during the execution. (In simulation invariants, the compiler can represent two states, but again only the current states of two memories.) The compiler uses the constructors `loc` and `glob`, in the logic for BL (Section 4.2), or the constructors `val` and `mem`, in the logic for BL with pointers (Section 6.1.1), to represent the current values of program variables or memory.[8] (In both logics, there are also indexed versions of constructors for simulation invariants.) We will use H to refer to all these constructors. Similar to the formulas for invariants, the compiler can represent only the current state in the formulas for input and output contexts, respectively, the state at the beginning and the state at the end of the execution of an activation.

We extend the logic formulas to allow the compiler to represent in any formula the starting state m^0, beside the current state m. We add a set of constructors H^0 (one for each appropriate H). The expressions with H^0 denote the corresponding values in m^0. For example, `2:inv glob(g)` $=$ $\texttt{glob}^0\texttt{(g)}$ means that the value of some global variable `g` is the same at node `2` as it is in the beginning of the procedure. Formally, the meaning of the new formulas is defined with respect to the contexts that consist of two memories and an environment: $c = \langle m^0, m, a\rangle$. The translation functions from program expressions to logic expressions remain the same. We next discuss the effect of the new expressions on the (standard and simulation) contexts, and then we describe the changes to the VCG.

Using the new formulas, the compiler can represent in an output context the state at the beginning of the context. This makes the contexts much more expressive. For example, consider a simple procedure that swaps the values of two variables `x` and `y`. The compiler can generate only one context for this procedure, namely $F^{in} \equiv \texttt{true}$ and $F^{out} \equiv \texttt{glob(x)} = \texttt{glob}^0\texttt{(y)} \wedge \texttt{glob(y)} = \texttt{glob}^0\texttt{(x)}$. Without $\texttt{glob}^0$, the compiler would need to generate a context with $F^{in} \equiv \texttt{glob(x)} = C_x \wedge \texttt{glob(y)} = C_y$ and $F^{out} \equiv \texttt{glob(x)} = C_y \wedge \texttt{glob(y)} = C_x$ for every two constants C_x and C_y for which the compiler uses the fact that the procedure swaps values. In general, the compiler can now generate only one context for the results of any analysis on any procedure. We argue that this is not always the best approach.

In the simple example with swap, different input contexts have only different values of the parameters. However, in more involved examples of context-sensitive interprocedural analyses, different input contexts may express different relationships between variables. Consider that such an analysis generates n different input contexts $F_1^{in}, \ldots, F_n^{in}$ and n corresponding output contexts $F_1^{out}, \ldots, F_n^{out}$ for some procedure. The compiler can combine all these contexts into one: $F^{in} \equiv F_1^{in} \vee \ldots \vee F_n^{in}$ and $F^{out} \equiv F_1^{in}[H^0/H] \Rightarrow F_1^{out} \wedge \ldots \wedge F_n^{in}[H^0/H] \Rightarrow F_n^{out}$. (The compiler has also to combine the invariants similarly as the output contexts.)

The VCG for the new formulas uses the contexts at call sites in the same way as the VCG for BL. When a procedure has only one combined context, the new VCG would use that context for all calls to that procedure. The first problem with the combined context is that the VCG uses F^{in} and F^{out}, instead of $F_{k'}^{in}$ and $F_{k'}^{out}$ for some k', and thus a part of VC is (roughly n times) longer at each call site. A much bigger problem is that the proof generator that proves the VC has to "rediscover" which of the contexts $F_1^{in}, \ldots, F_n^{in}$ to actually prove for each call site. Therefore, in the new VCG, we still allow the compiler to generate many standard contexts for the

[8] The addresses of global variables do not change during the execution of a program, and thus `addr(`I`)` represents the address of the variable I throughout the execution.

same procedure. (The compiler can also generate many simulation contexts for the same pair of procedures.)

The new VCG differs from the VCG for BL only in the substitutions of symbolic states, both for the StdVCG (Figure 4-9) and for the SimVCG (Figure 4-11). Each subst(F, s) for a standard invariant is replaced with subst(subst$(F, s), s^0)$, and analogously each subst-sim(J, s_1, s_2) is replaced with subst-sim(subst-sim$(J, s_1, s_2), s_1^0, s_2^0)$. We allow only H^0 constructors in the input context formulas, and the starting symbolic states (s^0, s_1^0, and s_2^0) map the expressions with H^0 to some logic expressions. The substitution at call sites also changes. The new StdVCG first creates $s^{in} =$ set-params$(I, G^*, s)[H^0/H]$ and then performs the substitutions subst(F^{in}, s^{in}) and subst(subst$(F^{out}, s'), s^{in})$; the new SimVCG operates analogously. The soundness proofs for the new StdVCG and SimVCG proceed in a similar way as for the StdVCG and SimVCG for BL.

6.2.3 Formula Extensions

We next discuss general extensions to the logic formulas and, in particular, adding set expressions to the formulas. Generally, adding new predicates and/or types of logic variables and expressions allows the compiler to generate shorter invariants. As a simple example, consider an analysis that determines which program variables have truth values (0 or 1). Using the formulas presented so far, the compiler can represent the results of the analysis only with expressions `val(`I`)` $=$ `0` $\vee$ `val(`I`)` $=$ `1`. Introducing a new constructor `bool` allows the compiler to use `bool(val(`I`))` instead.

Adding new constructors requires adding proof rules for the formulas with new constructors. In the example with `bool`, it is enough to add only the rule for the definition of `bool` (for all x, `bool(`x`)` iff $x = 0 \vee x = 1$), and the compiler could add such definitions automatically. However, in more complex cases, it is usually necessary to add several proof rules for the new formulas and to generate a *meta-proof* that the new proof rules are sound. Since the compiler cannot generate a meta-proof, we need to specify, before a compilation, a logic that allows the compiler to efficiently represent and prove the results of "standard" analyses.

Based on our experience with the implementation of a (flow-sensitive intraprocedural) pointer analysis, we find sets (sequences) of constants to be particularly useful for expressing the results of compiler analyses. In BL extended with pointers, we regard as constants the addresses of global variables as well as the integer constants (C ::= `addr(`I`)`$|Z$). We have started extending the logic with the expressions that denote sets of constants: G^s ::= `empty`|`union(`C`,`G^s`)`. The predicate `in(`G^i`,`G^s`)` denotes that the value of expression G^i is in the set G^s. For example, the expression `in(val(i),union(0,union(1,empty)))` denotes that the value of `i` is either 0 or 1, i.e., `val(i)` $=$ `0` $\vee$ `val(i)` $=$ `1`. Similarly,

```
in(sel(mem,&p),union(&x,union(&y,empty)))
```

denotes that the pointer `p` points either to `x` or to `y`. To fully utilize the sets, we plan to further extend the logic with ways for naming sets and for proving subset relationships between sets.[9]

As mentioned briefly in Section 6.1.1, we can also use a set of addresses of variables to represent, in simulation invariants, that two memories have the same value in all locations except for the locations whose addresses are from the set G^s:

$$\forall x.\ \texttt{in(}x\texttt{,}G^s\texttt{)} \Rightarrow \texttt{sel(mem}_1\texttt{,}x\texttt{)} = \texttt{sel(mem}_2\texttt{,}x\texttt{)}.$$

Observe that once the formulas involve sets of addresses, it is not possible to prove that two memories are equal even if the set is empty. We need therefore to add a rule that $\texttt{mem} = \texttt{mem}'$ if $\forall x.\ \texttt{sel(mem}_1\texttt{,}x\texttt{)} = \texttt{sel(mem}_2\texttt{,}x\texttt{)}$.

6.2.4 Flow-Insensitive Analyses

We next describe how the compiler represents the results of flow-insensitive analyses and how a specialized StdVCG generates StdVC for those results. We also discuss the relationship between those results and the initialization of variables.

Flow-insensitive analyses generate, for one context (a pair F^{in} and F^{out}), the same result (formula F) for all nodes in a procedure. We can represent this result with only one standard invariant of a special form, e.g., $\texttt{*:inv}\ F$. This invariant represents that F holds everywhere.

Further, flow-insensitive analyses do not use the information from the branch conditions. We can therefore use a specialized StdVCG that does not execute the branch nodes of the procedure and generates a shorter StdVC.[10] The StdVCG for $\texttt{*:inv}\ F$ executes only the assignment nodes and the call nodes. The StdVCG first creates fresh symbolic states s^0 and s, with logic variables x^{*0} and x^*, and then generates the following part of StdVC for the input and output contexts and the assignment nodes:

$$\begin{aligned}\forall x^*, x^{*0}.\ &\text{subst}(F^{in}, s^0) \Rightarrow \text{subst}(\text{subst}(F, s), s^0) \Rightarrow \text{subst}(\text{subst}(F^{out}, s), s^0) \wedge \\ &\textstyle\bigwedge_{W\texttt{=}E\in P} \text{subst}(\text{subst}(F, \text{translate-assign}(W, E, \texttt{mem}, s, e)), s^0),\end{aligned}$$

where the conjunction ranges over all assignment nodes in the procedure. The StdVCG also generates a similar part of StdVC for the call nodes as for the assignment nodes.

[9] Checking the subset/membership relationships using the proof checker requires the relationships to be encoded in proofs. It is more efficient to check the relationships using a function additional to the proof checker. The Athena framework, which we use for proof representation and verification, offers a direct way to add such a function, i.e., to add computation to deduction.

[10] Similarly, we can use specialized StdVCGs for other kinds of analyses. For example, standard (non-predicated) dataflow analyses do not use the information from the branch conditions. Therefore, for the results of such analyses, the StdVCG does not need to add the branch conditions to the StdVC. (Omitting those conditions from StdVC is sound since the conditions are used only as assumptions in the StdVC. Also, the proof generator can prove the StdVC without the conditions since the analysis results do not depend on those conditions.) The StdVCG still needs to symbolically execute all nodes of the procedure to generate the StdVC. In general, the compiler has to execute all nodes and to also generate the branch conditions; otherwise, the proof generator might not be able to prove the StdVC.

We next discuss the relationship between flow-insensitive analyses and the initialization of program variables. We show that obtaining effective flow-insensitive analyses requires that the language provides some guarantees. As an example, we consider a flow-insensitive pointer analysis. A pointer analysis determines where each pointer can point to. Assume that some procedure (in BL with pointers) has only one pointer `p` and two assignments to `p`: `p=&x` and `p=&y`. Usually, an analysis would generate that $\texttt{val(p)} = \texttt{\&x} \vee \texttt{val(p)} = \texttt{\&y}$ everywhere in the procedure. However, this actually does not hold (and thus cannot be proven) at the beginning of the procedure—`p` can have any value in the starting state. Therefore, a flow-insensitive result has to include the starting value, namely $\texttt{val(p)} = \texttt{\&x} \vee \texttt{val(p)} = \texttt{\&y} \vee \texttt{val(p)} = \texttt{val}^0\texttt{(p)}$.

The results become less precise (and thus less useful) after adding the option that a variable can have the starting value anywhere. For example, even if `p` could otherwise point only to `x`, the compiler could not replace `*p` with `x` when `p` can also have the starting value $\texttt{p}^0$. Furthermore, if the value $\texttt{p}^0$ is unknown, as it is for the uninitialized local variables in BL, then `*p` could access any location. This, in turn, prevents effectively performing a flow-insensitive pointer analysis because variables that are assigned the value read from `*p` get an arbitrary value.[11]

We showed in [42] how to minimally change the language semantics to enable a flow-insensitive pointer analysis to generate provably correct results. We require all variables to be initialized to some value, v, and accessing the location with address v has special behavior: a read from location v always returns v and a write to location v does not change the memory. (We use 0 for v, and thus give a special semantics to null pointer dereferencing.) This still requires that the results of the analyses include v as a possible value for all variables.

As mentioned, flow-insensitive analyses in general, and in particular pointer analyses such as Steengaards's [44] and Andersen's [2], generate results that do not include the starting values. The assumption under which these analyses operate is "no use before definition," i.e., no memory location is read from before it is written to. The generated results then hold for all uses of a variable, but they need not hold before the first definition. (Additionally, when the semantics has error states, the results need not hold if the execution gets to an error, but the results hold if there is no error.)

For the full C language, as well as for BL, the "no use before definition" assumption clearly does not hold in general. It is, also, undecidable to determine whether the assumption holds for an arbitrary program. One method to ensure that the assumption holds is to initialize, at run-time, all local variables at the beginning of a procedure. For efficiency reasons, realistic languages do not require the initialization of local variables. (They may require the initialization of heap data.) The other method to achieve that the assumption holds is to accept (for compilation or execution) only the "correct" programs, i.e., programs for which some analysis determines that the assumption holds. For example, the Java Bytecode Verifier [28] performs a

[11] Even the results of flow-insensitive analysis in programs without pointers become less precise when variables can have starting values anywhere. The reason is that assignments propagate the starting values; for instance, the assignment `x=y` necessitates that the result for `x` also includes $\texttt{y}^0$.

dataflow analysis to determine that "no local variable is accessed unless it is known to contain a value of an appropriate type."

Finally, observe that flow-insensitive analyses cannot, by themselves, determine whether a procedure is "correct." Therefore, another method has to first establish that the procedure is "correct." The result then cannot depend on the starting values of the local variables, and the StdVCG can generate StdVC that existentially quantifies logic variables representing the starting values.

6.3 Limitations

In this section we present some limitations of the current framework for credible compilation. We discuss how the framework could or could not support translations of programs from one representation to another representation with different syntax and semantics. As mentioned in Chapter 1, we make a distinction between transformations and translations.

We use the term transformation for a compiler pass whose input and output programs are in a similar representation. For example, before register allocation, the representation of a program includes temporary variables (virtual registers), but no physical registers. After register allocation, the representation includes physical registers and spill slots. Although the representations before and after register allocation are not exactly the same, they are quite similar. In fact, each representation is a subset of a general representation that includes variables and registers. Therefore, a VCG can use the same symbolic execution for both input and output programs. That is exactly what the basic VCG and the extensions presented so far do.

We use the term translation for a compiler pass whose input and output programs are in essentially different representations. For example, lexical analysis translates the program from the source code to a sequence of tokens and parsing translates the list of tokens into an abstract syntax tree. (After these two passes, a compiler front-end usually performs semantic analysis that checks whether the program satisfies semantic conditions.) A C compiler that uses a BL-like intermediate representation also needs to translate the syntax tree into a flow graph. Compilers for more advanced languages usually do not generate a low-level BL-like representation directly from the syntax tree. Instead, these compilers use several levels of intermediate representations and translate the program from higher-level to lower-level representations. These translations involve both transforming code and changing data representation. Finally, compilers also perform a translation in the back-end where code generation pass translates the program from an intermediate representation to the machine language.

The framework presented so far can support only transformations. It is not clear how we could extend the framework to support front-end translation passes. The representations before and after the front-end are completely different, and the source code is not suited for a symbolic execution. Therefore, the front-end of a credible compiler has to be trusted. We can say, alternatively, that a verifier can check the results of the compiler front-end only if the verifier itself has an implementation of the same front-end as the compiler has. With tools available for automatic generation of lexical analyzers and parsers, paranoid programmers could develop their own front-ends. In general, the front-end is not regarded as an origin of many compiler errors.

At a glance, it seems easier to extend the framework to support translations from one intermediate representation to another. All we need is a SimVCG with two different symbolic executions: one for the input program representation and one for the output program representation. However, we also need simulation invariants that can express relationships between different program representations. It is not clear how we could make such simulation invariants in general. The main problem is how to efficiently express relationships between different data representations.

Consider, for example, a translation from a Java-like program representation to a C-like representation. Such translation needs to translate data represented with Java-like classes and objects into a representation with C-like structures. The simulation invariants would then need to describe which data in one representation corresponds to which data in the other. As even simpler example, consider a program in BL with pointers and a transformation that does not change the code but only changes the data layout, i.e., the addresses of the global variables. Although this change is within the same representation, the compiler would need to specify a mapping from the new addresses to the old addresses and to represent, in simulation invariants, that the two memories are related under the given (re)mapping of the addresses. We believe that the framework can be extended to support some of these translations using simulation invariants that would involve mappings from one data representation to another.

We do not explore credible code generation in this thesis, but in principle, it is not a limitation for the presented framework. Rinard [41] briefly discusses how a credible compilation framework can support code generation. The idea is that the compiler first transforms the control flow graph of the program so that each node closely corresponds to a machine instruction. After that, the compiler uses a simple translation to generate the actual binary code. This approach requires an intermediate representation that models all details of the target instruction set architecture. Designing such an intermediate representation for a complex architecture, such as Intel IA-64, is a non-trivial task, but we believe that it can be done by extending the types of nodes in the control flow graph, as described in [41].

The presented framework can also support "compiling to logic" as done, for instance, in the DeepC compiler developed by Babb et al. [9]. This compiler targets FPGA-based systems and has a much cleaner code generation than a compiler that targets some specific instruction set. We believe that this makes it even easier to develop a credible code generation for DeepC.

Chapter 7

Related Work

The most widely used compilers today do not provide any formal evidence of correct compilation. However, there is a large body of research on correct compilation. The first proof of a compiler correctness can be tracked down to a 1967 paper by McCarthy and Painter [29]. They present a paper-and-pencil proof that a compiler algorithm is correct. Even before, McCarthy argued that proofs should be, instead, machine-verifiable and that computers should be used to check proofs. There is a difference, though, between checking a specification of a compiler algorithm, or for that matter any algorithm, and the actual implementation of the algorithm.

Most research on compiler correctness focused on proving translation algorithms correct [18,22,24,33,47]. There are several aspects in which these projects differ. First, some projects present proofs for all translation steps from a high-level source language to a machine language, whereas other projects present proofs only for some parts of compilers, or do not translate to a realistic machine language. Second, in several projects mechanical proof verifiers are used to complement the manual proofs or to substitute them. Finally, implementations of some algorithms are carefully verified through stepwise refinements. These implementations, however, do not generate run-time proofs that show the compilation to be correct.

There are several pragmatic drawbacks in implementing a fully verified compiler. They stem from the fact that the implementation and verification methodology is not completely automatic. It is therefore possible to have human-introduced errors in the development process. Also, the effort of changing a compiler is much greater for a fully verified compiler than for a compiler that generates proofs at run-time. Furthermore, some changes are almost impossible in practice—it is extremely costly to extend a fully verified compiler with a transformation from an untrusted source. This would require checking the whole implementation of the new transformation before it can be safely added. That is why all fully verified compilers were developed by small, closed groups of people. Compilers that generate run-time proofs, on the other hand, offer much more possibility for having an open source compiler to which anyone can contribute. It is not the compiler program that is checked, but its result.

The concept of checking the equivalence of the input and output programs after each compiler run appeared in several works at approximately the same time. Cimatti et al. [10,12] present a system for verifying translations of non-executable "embedded" programs to an executable form. These programs consist of only a single loop whose body is translated. Their verifier checks that the input and output programs satisfy a particular syntactic equivalence condition, which then implies semantic equivalence. Cimatti et al. call the syntactic equivalence proof "on-line," as opposed to the "off-line" proof which shows the soundness, i.e., that the syntactic equivalence implies the semantic equivalence for all possible pairs of programs. Pnueli et al. [39, 40] present a system for *translation validation*—verifying translations from programs in synchronous languages to programs in the C programming language. These programs also consist of a single loop that cyclically computes the values of the output variables from the values of the input variables. Our approach is designed for imperative languages with programs that can have arbitrary flow of control.

The general technique of *program checking*—checking the program run-time results instead of verifying the whole program code—was first considered by Blum and Kannan [11]. Clearly, checking the program results is much easier than verifying the code in many applications. In some applications, it is possible to verify that the output of the program is correct by checking only that the output itself satisfies some conditions with respect to the input. In other applications, though, the program needs to generate the regular output and also additional information which eases, or enables, the checking. As explained in Section 2.2, this in particular holds for compilers. It is not possible, *in general*, to check that the output program is correct simply by considering the output and input programs.

Goos and Zimmermann [20] present another methodology for developing correct compilers. This work is a part of the bigger Verifix project which proposed several approaches for constructing provably correct compilers for realistic programming languages. The earlier approaches used only the compiler implementation verification, whereas the new approach [19, 20] also uses program checking techniques for the verification of compiler output. However, the program checking idea is used in the direct way, without requiring the compiler to generate any additional output besides the transformed program. The drawback of this technique is that adding a new transformation requires a new checker which has to be verified itself using standard methods. Therefore, the compiler cannot be easily extended from untrusted sources. Gaul et al. [17] report on the use of this methodology for developing compiler back-ends.

Proof carrying code (PCC), introduced by Necula and Lee [34, 37], is a general framework for attaching proofs to the compiled code. Credible compilation can be regarded as an instance of this framework, with the main goal to deliver proofs that the transformed code is semantically equivalent to the original code. Necula and Lee [38] use the name *certifying compiler* to refer to a pair consisting of a compiler, which produces code annotated with some additional information, and a certifier, which uses the annotations to generate the proof and check it. We prefer to use a different name, because so far certifying compilers have been developed to generate proofs for properties of *one* (compiled) program, whereas credible compilers generate proofs about *two* programs. We are not aware of any other work with the goal of generating equivalence/simulation proofs in PCC framework.

Necula describes in his PhD thesis [35] the Touchstone compiler, a certifying compiler that translates a type-safe subset of C[1] into machine code annotated with *invariants*. The invariants allow the certifier to prove safety properties, in particular type safety, of the compiled code. For the proof formalism, Necula uses an extension of first-order predicate logic. The certifier has a verification-condition generator which takes the annotated code and generates a verification condition—a formula whose validity implies the safety of the code. An untrusted theorem prover is then used to generate the proof of the verification condition, and a trusted proof checker verifies this proof. Our structure of the credible compiler is similar to the Touchstone, but the credible compilers should prove more; we try "to be more ambitious and attempt to verify not just the type safety of the target code but also its equivalence to the source program" [35, page 152].

Morrisett et al. [32] present another approach in building a certifying compiler for a PCC framework. They do not use first-order predicate logic for expressing safety policies; instead, they use type systems, and proof checking reduces to type checking. They based their work on typed intermediate languages [43,45] and designed an idealized typed assembly language. In a later work [31], they develop a type system for the Intel IA32 assembly language and implemented the Popcorn compiler that translates a type-safe subset of C into machine code and generates the required typing information. This system has been extended to support advanced language constructs, e.g., run-time code generation [25], and more expressive security properties, e.g., resource bound verification [15].

Appel and Felty [3] present a PCC framework in which a code producer has much more flexibility. The typing rules are not fixed in the safety policy, but the code producer can choose a set of typing rules, prove them sound, and then use them to prove the safety of a program. Similarly, the machine code semantics is not fixed in the verification-condition generator, but is a part of the safety policy. This eliminates the need for the verification-condition generator, but requires more complex proofs (as we also explained in Section 2.3). The increase in complexity is not as huge for a machine language as it would be for a syntactically (and semantically) richer higher-level language such as BL in our basic framework.

Necula's recent work [36] is more related to our approach on credible compilation.[2] He describes a *translation validation infrastructure* that checks equivalence of the compiler input and output programs (both in an intermediate representation), and not only properties of the output program. His framework is similar to ours in that it uses simulation invariants (called simulation relations) and symbolic execution. However, the difference is that his symbolic execution does not use simulation contexts and therefore his current framework can support only intraprocedural transformations.

Another difference from our approach is that there are no proofs in Necula's translation validation. The compiler (or a theorem prover) does not generate any proof. Instead, the checker has built in rules for equivalence and uses them to verify

[1] Recently, Colby et al. [13,14] report on the development of the Special J compiler, a certifying compiler for Java.

[2] Necula's recent work was done in parallel with the work discussed in this thesis. We do not use any results from [36] in the approach presented in this thesis.

the simulation invariants. This complexity makes the checker bigger and more difficult to verify than a "standard" proof checker used in logical frameworks. Additionally, to support the full C language, Necula uses the rules that informally model the C notion of "undefined." This approach can lead to errors that introduce unsound rules in the checker, especially for aliasing.

Beside the checking algorithm, Necula also presents the *inference algorithm*. This algorithm discovers the simulation invariants for the input and output programs, without requiring any additional information from the compiler. Necula implemented his inference algorithm for verifying transformations in the widely used GNU C optimizing compiler (`gcc`). The inference algorithm can discover the simulation invariants in all intraprocedural transformations that `gcc` performs, except for some cases of loop unrolling. This is an important result that shows that the simulation invariants can be practically inferred for a realistic compiler output. However, `gcc` is not an aggressive compiler. For example, it does not have transformations that involve pointers, and it would not even try to optimize the third loop in the example shown in Section 1.3. The conclusion is that, in general, a credible compiler should generate some additional information to enable the verifier to check the correctness of a transformation.

Chapter 8

Conclusions and Future Work

Today, widely-used industry compilers offer no formal guarantees that they work correctly. Most previous research on compiler correctness focused on developing compilers that are guaranteed to correctly translate every input program. It is extremely difficult, however, to verify that a complex code, which implements a compiler, is correct. Therefore, a novel approach has been recently proposed: instead of verifying a compiler, verify the result of each single compilation. We require the compiler to generate a transformed program and some additional information that enables a simple verifier to check the compilation. We call this approach *credible compilation.*

This thesis presents a theoretical framework for credible compilation. We develop a framework in which a compiler proves correct the results of transformations. The transformations operate on programs in an intermediate representation based on flow graphs. Each transformation generates an output program and two sets of invariants and contexts: standard invariants and contexts, which allow the compiler to prove that the analysis results are correct, and simulation invariants and contexts, which allow the compiler to prove that the output program simulates the input program. Additionally, the compiler has the proof generator that generates a proof that all the invariants and contexts are correct.

We describe in detail the structure of a verifier that checks the invariants and contexts. The verifier first uses the standard and simulation verification-condition generators to generate the verification condition for the given programs and the additional information. The verifier then uses a proof checker to verify that the supplied proof indeed proves the particular verification condition. If the proof fails, the output program potentially does not simulate the input program, and the compiler should not use this transformation for this input program. If the proof is accepted, the particular transformation is correct.

This thesis shows how to formalize the basic techniques for building credible compiler transformations for a simple imperative language. There are several directions for the future work on credible compilation, both in extending the theoretical framework and implementing a credible compiler.

Two questions about the framework are what language it supports and what transformations it supports. The main goal is to develop a formal framework that supports a realistic language. We believe that it can be done using a compiler intermediate representation that has cleaner and safer semantics than the full-blown unsafe C language. Otherwise, to handle C, it is necessary to compromise the presented formal approach, or use complex theoretical models that would give poor performance in an implementation. Regarding transformations, the presented framework can be extended to support more transformations. However, the ultimate goal is to have credible compilation for all compiler phases, not only intermediate transformations. This requires support for code generation and front-end translations. Additionally, compilers for advanced languages usually have several intermediate representations, and the framework should support translations between those representations, in particular translations from abstract to concrete data representations.

Two fundamental questions that an implementation can answer are is it possible for a credible compiler to generate the required additional information and is it possible to automatically prove the verification conditions. Additional pragmatic issues in the context of credible compilation are the difficulty of generating the proofs, the size of the generated proofs, and the difficulty of checking the proofs. To explore these issues, we have started developing a prototype of a credible compiler. We have implemented a small system for the language without procedures, but with pointers. We have used Java [8] for implementing a flow-sensitive pointer analysis and constant propagation analysis/transformation.

For proof representation and verification we use Athena [5,6], a *denotational proof language* [7] developed by Kostas Arkoudas at MIT. Athena is a flexible logical framework that allows a compact, procedural representation of proofs. This makes it possible to balance the division of labor between the proof generator and the proof checker, while retaining the full soundness guarantee. It also simplifies the construction of the compiler by simplifying the proof generator and allowing the compiler developer to easily generate proofs. Based on our initial positive experience with Athena, we believe that a key enabling feature to obtaining reasonable proof sizes and compiler complexity is the use of such a flexible logical framework. We intend to continue to use Athena for credible compilation. Our plan is to describe, in a follow-up paper, the implementation strategy for a credible compiler based on Athena.

Bibliography

[1] A. V. Aho, R. I. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., Reading, MA, second edition, 1986.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994.

[3] A. Appel and A. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th Annual ACM Symposium on the Principles of Programming Languages*, pages 243–253, Boston, Massachusetts, Jan. 2000.

[4] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, second edition, 1997.

[5] K. Arkoudas. *An Athena Tutorial.* Available from: `http://www.ai.mit.edu/projects/express/athena.html`.

[6] K. Arkoudas. Deduction vis-a-vis computation: The need for a formal language for proof engineering. Available from: `http://www.ai.mit.edu/projects/express/athena.ps`.

[7] K. Arkoudas. *Denotational Proof Languages.* PhD thesis, Massachusetts Institute of Technology, May 2000.

[8] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, Reading, Mass., 1996.

[9] J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe. Parallelizing applications into silicon. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines (FCCM)*, Napa Valley, CA, Apr. 1999.

[10] P. Bertoli, A. Cimatti, F. Giunchiglia, and P. Traverso. Certification of translators via off-line and on-line proof logging and checking. Technical Report 9710–14, IRST, Trento, Italy, Oct. 1997.

[11] M. Blum and S. Kannan. Designing programs that check their work. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 86–97, Seattle, Washington, May 1989.

[12] A. Cimatti, F. Giunchiglia, P. Pecchiari, B. Pietra, J. Profeta, D. Romano, P. Traverso, and B. Yu. A provably correct embedded verifier for the certification of safety critical software. In *Proceedings of the 9th International Conference on Computer Aided Verification*, pages 202–213, Haifa, Israel, June 1997.

[13] C. Colby, P. Lee, and G. C. Necula. A proof-carrying code architecture for Java. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV00)*, Chicago, July 2000.

[14] C. Colby, P. Lee, G. C. Necula, F. Blau, K. Cline, and M. Plesko. A certifying compiler for Java. In *Proceedings of the ACM SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, British Columbia, Canada, June 2000.

[15] K. Crary and S. Weirich. Resource bound certification. In *Proceedings of the 27th Annual ACM Symposium on the Principles of Programming Languages*, pages 184–198, Boston, Massachusetts, Jan. 2000.

[16] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proceedings of Symposium in Applied Mathematics*, volume 19, pages 19–32, 1967.

[17] T. Gaul, A. Heberle, W. Zimmermann, and W. Goerigk. Construction of verified software systems with program-checking: An application to compiler back-ends. In *Proceedings of the Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[18] W. Goerigk. Towards rigorous compiler implementation verification. In R. Berghammer and F. Simon, editors, *Proc. of the 1997 Workshop on Programming Languages and Fundamentals of Programming*, pages 118–126, Avendorf, Germany, Nov. 1997.

[19] W. Goerigk, T. Gaul, and W. Zimmermann. Correct programs without proof? On checker-based program verification. In *Proceedings of ATOOLS'98 Workshop on "Tool Support for System Specification, Development, and Verification"*, Malente, 1998.

[20] G. Goos and W. Zimmermann. Verification of compilers. *Correct System Design, Lecture Notes in Computer Science*, 1710:201–230, 1999.

[21] Y. Gurevich and J. K. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

[22] J. Guttman, J. Ramsdell, and M. Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computing*, 8(1–2):33–110, Mar. 1995.

[23] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct 1969.

[24] C. A. R. Hoare, H. Jifeng, and A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30:701–739, 1993.

[25] L. Hornof and T. Jim. Certifying compilation and run-time code generation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 60–74, San Antonio, Texas, Jan 1999.

[26] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

[27] J. C. King. Proving programs to be correct. In *IEEE Transactions on Computers*, volume 20, pages 1331–1336, Nov 1971.

[28] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, Mass., 1996. The Bytecode Verifier: `http://java.sun.com/docs/books/vmspec/html/ClassFile.doc.html#9801`.

[29] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Proceedings of Symposia in Applied Mathematics.* American Mathematical Society, 1967.

[30] R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 19, pages 1201–1242. Elsevier and MIT Press, 1994.

[31] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *Second Workshop on Compiler Support for System Software*, Atlanta, Georgia, May 1999.

[32] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proceedings of the 25th Annual ACM Symposium on the Principles of Programming Languages*, pages 85–97, San Diego, Jan. 1998.

[33] M. Müller-Olm. Modular compiler verification. *Lecture Notes in Computer Science*, 1283, 1996.

[34] G. C. Necula. Proof-carrying code. In N. D. Jones, editor, *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, Jan. 1997. ACM Press.

[35] G. C. Necula. *Compiling with Proofs.* PhD thesis, Carnegie Mellon University, Oct. 1998. Available as Technical Report CMU-CS-98-154.

[36] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, British Columbia, Canada, June 2000.

[37] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI'96)*, pages 229–243, Seattle, Washington, Oct. 1996.

[38] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In K. D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 333–344, Montreal, Canada, June 1998. ACM Press.

[39] A. Pnueli, O. Shtrichman, and M. Siegel. Translation validation for synchronous languages. *Lecture Notes in Computer Science*, 1443:235–246, 1998.

[40] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166, Lisbon, Portugal, Mar. 1998.

[41] M. Rinard. Credible compilation. Technical Report MIT-LCS-TR-776, Laboratory for Computer Science, Massachusetts Institute of Technology, Mar. 1999. Available from: `http://www.cag.lcs.mit.edu/~rinard/techreport/credibleCompilation.ps`.

[42] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[43] Z. Shao and A. W. Appel. A type-based compiler for Standard ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Program Language Design and Implementation*, pages 116–129, 1995.

[44] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, Jan. 1996.

[45] D. Tarditi, J. G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Program Language Design and Implementation*, pages 181–192, May 1996.

[46] F. Turbak and D. Gifford. Applied semantics of programming languages. Unpublished book draft used in MIT course 6.821, Sep 1999.

[47] W. Zimmermann and T. Gaul. On the construction of correct compiler backends: An ASM-approach. *Journal of Universal Computer Science*, 3(5):504–567, 1997.