Sound Program Transformation Based on Symbolic Execution and Deduction

Beweisbar korrekte Programmtransformation basierend auf symbolischer Ausführung und Deduktion
Zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation von Ran Ji M.Sc. aus Tianjin, China

1. Gutachten: Prof. Dr. Reiner Hähnle
2. Gutachten: Prof. Dr. Bernhard Beckert

Tag der Einreichung: 18.12.2013
Tag der Prüfung: 20.06.2014

Juni 2014 — Darmstadt — D 17

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Software Engineering

Abstract

In this thesis, we are concerned with the safety and security of programs. The problems addressed here are the correctness of SiJa (a subset of Java) source code and Java bytecode, and the information flow security of SiJa programs. A lot of research has been made on these topics, but almost all of them study each topic independently and no approach can handle all of these aspects. We propose a uniform framework that integrates the effort of proving correctness and security into one process. The core concept for this uniform approach is sound program transformation based on symbolic execution and deduction. The correctness of SiJa source code is verified with KeY, a symbolic execution based approach. Partial evaluation actions are interleaved during symbolic execution to reduce the proof size. By synthesizing the symbolic execution tree achieved in the source code verification phase, we can generate a program that is bisimilar to, but also more optimized than, the original one with respect to a set of observable locations. The soundness of program transformation is proven. Apply the sound program transformation approach, we can generate a program bisimilar to the original program with respect to the low security level variables. This results in a more precise analysis of information flow security than the approaches based on security type systems. We can also generate Java bytecode from SiJa source code program transformation approach, where the the correctness of the Java bytecode is guaranteed and compiler verification is not necessary.

Acknowledgment

First of all, I would like to express my deepest gratitude to Prof. Dr. Reiner Hähnle, for being a great supervisor and a best friend. His wise guidance always shows me a way out of the puzzle and lights up the new hope. My personal life is also enriched by his appreciation of art, wine, food and many more. It would be even better if and only if he is more interested in football.

I own my sincerely grateful to Dr. Richard Bubel. For all the times when I am in need, he is ready for an inspiring discussion. He shows me a good example of being not only a talented researcher but also a true gentleman.

It has been a great fun to work in our research group, thanks to the amazing colleagues and friends: Martin Hentschel, Antonio Flores Montoya, Nathan Wasser, Huy Quoc Do, Crystal Chang Din, and all the members of Prof. Dr. Mira Mezini's group. There exists a person I need to mention separately. Our secretary Gudrun Harris is always kind and helpful, and she has even taught me some German language that I will probably never learn from anywhere else.

I have pleasure to work with many excellent researchers in the KeY project and the HATS project. Every meeting and discussion with them is truly enjoyable. Among them, I would specially thank Prof. Dr. Bernhard Beckert for being the opponent for my PhD dissertation, whose valuable comments help a lot to improve this work.

It was in Chalmers University of Technology, when I started my PhD work. I am grateful to my co-supervisor Dr. Wolfgang Ahrendt, my colleague Gabriele Paganelli, and many nice people there, for your accompany of my two-and-a-half-years cheerful life in Göteborg, Sweden.

Last but not least, to my parents, I love you.

Contents

List of Figures

1 Introduction

When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem.

— Edsger W. Dijkstra [Dij86]

1.1 Overview: Software Correctness and Security

Along with the development of information technology, computers play an important role in the modern society. Not only are they used everywhere, especially for handling complicated tasks, but the computer itself is getting more and more complex in both hardware and software aspects. Though it is difficult to make the hardware part working flawlessly, it is even more challenging to ensure the correctness of the software part. Sometimes, the software failures can result in a disaster, such as the infamous explosion of Ariane 5 [Boa06]. There are more than 100 "software horror stories" listed in this website document [Der], to show the consequence of software bugs. Therefore, ensuring the *correctness* of software is extremely important, no matter of its difficulties.

What is the meaning of "correctness"? The correctness of software is asserted when it is correct with respect to the *specification* that describes the intended behavior of the software. Specification can be informal, in which case it can be considered as a blueprint or user manual from a developer's point of view. Formal specification, that models the software behavior in a mathematically rigorous way, will contribute to achieving a more reliable software system.

Assertions can be considered as a lightweight formal specification. When a program reaches a particular execution point, the assertions, normally written in Boolean expressions or formulas, should always be satisfied. An extension of the assertion mechanism is *design by contract* [Mey86, Mey92], which uses preconditions, postconditions and invariants to specify the classes of the object-oriented programs. The Eiffel programming language [Mey00] features built-in support for design-by-contract specifications.

It is worth to mention that writing a good formal specification is nontrivial due to the mathematical skills required from the programmers and the expressiveness of the specification language that is used. Research on specification generation includes QuickSpec [CSH10] that generates the specification automatically for sets of pure functions based on testing, and the ABS [CDH$^+$11] specification language that bridges the gap between a highly abstract modeling language and an implementation-oriented specification language.

A natural way to ensure the correctness of software is to take some inputs and execute the program to see if the outputs are as expected with respect to the specification. This method is known as program *testing*. In the testing process, a collection of test cases are used to cover as thoroughly as possible the program execution branches to show whether errors will occur. The chosen test cases are the key factor of testing; several quality criteria for a test suite have been proposed to describe the degree of code coverage [Mye04]. The test cases can also be generated automatically [EH07]. Testing is the most used method to establish high quality of software in industry nowadays, and it will retain its importance in the future. However, certain limitations of testing imply that it is not the only, not even the best, way to achieve bug-free software. For many software systems, the state space is too large for exhaustive testing; while in a concurrent setting, it is simply impossible to reproduce all feasible runs. As Dijkstra [Dij70] famously pointed out, “Program testing can be used to show the presence of bugs, but never to show their absence".

Another way to design reliable software is by using *formal methods*. It is a mathematically rigorous technique for specification, design and verification of software systems. One kind of formal methods is oriented on the abstract design process. The examples are the modeling languages such as *ASM* [BS03], *B* [Abr96], *Z* [Spi92] and *Alloy* [Jac02].

Formal software verification usually concentrates on the *source code* level, ensuring the correctness of the software implemented in a certain programming language. Both dynamic and static techniques can be used in software verification, depending on whether to execute the program or not. Assertions and design-by-contract specifications are often used for dynamic run-time checks to indicate whether a test run has been successful. Static verification techniques, which do not rely on program execution, include *abstract interpretation* [CC77], *software model checking* [MCDE02, HJMS03, CKL04, BBC$^+$06] and *deductive verification* [Hoa69].

Abstract interpretation relates abstract analysis to program execution by evaluating the behavior of a program on an abstract domain to obtain an approximate solution. Software model checking is often combined with abstraction techniques, e.g., *predicate abstraction* [FQ02], since the state space of software programs is typically too large to analyze directly. In both methods, abstraction techniques are used, but no formal logical proofs are created. Deductive verification constructs a logical proof of the program to show its correctness. The generated proof obligations or verification conditions for the program are proved automatically, or manually with some interactions, by *theorem provers* or *SMT* solvers. No matter which formal software verification technique is used, the state space exploration problem is always one of the major concerns and it is an important research topic in this area.

There are many formal software verification techniques to ensure the correctness of the source code, despite their limitations, however, the correctness of the software system is not guaranteed by the correct source code only, because errors can also happen e.g. due to a buggy compiler. Another important topic is *compiler verification* [Dav03] that aims to ensure the correctness of the compiled code (*bytecode*), with respect to the source code. It normally requires reasoning about actual compiler implementations and the behavior of the compiler for an infinite number of programs and their translations, so compiler verification is very expensive and hard to scale to realistic programming languages and sophisticated optimization.

A "grand challenge" for computer science proposed by Hoare [Hoa03] is to achieve a "verifying compiler" that checks the correctness of a program along with compilation, just like a compiler performing type checking nowadays.

Since flow of information plays a growing role in society, the preservation of *confidentiality* becomes an important concern. Confidentiality of programs is an issue of software *security*. *Information-flow control* [Den82] tracks the flow of information in programs to ensure that no *information leak* occurs. Language-based *information flow security* [SM03] applies language-based techniques to analyze the program, in an automated manner, to enforce that the program satisfies a *security policy* of interest.

In information-flow control, a security policy is accompanied by a permissive *enforcement mechanism*, proven sound with respect to a security policy. When run on a program, if the enforcement reports a positive result, then the soundness proof implies that the program satisfies the policy. There are several ways to achieve this. Static analysis approaches take the form of a security type system [VIS96, HS06], by tracking the confidentiality level of information contained in variables and program context, (over-)approximates information flows occurring in (an over-approximation of) the control flow paths the program can take. Dynamic analysis approaches are usually security *monitors* [Vol99, AS09], which monitor the propagation of the input data that is labeled with the confidentiality level at run time. Static analysis approaches have the advantage of no runtime overhead; while dynamic analysis approaches need to access to the current control flow path so that highly dynamic language constructs can be treated in a permissive manner.

1.2 Problems and Contributions

Traditionally, source code correctness, bytecode correctness and source code security are analyzed independently with different approaches and tools, as illustrated in Figure 1.1.

In this thesis, we try to integrate all these 3 aspects into a uniform framework, so that it is possible to ensure software correctness and security within one process. We studied SiJa as the programming language in this thesis. It is a subset of Java with certain restrictions.

The core concept is a *sound* approach of *program transformation*. A program transformation is an operation that takes a program and generates another program. The transformed program is required to be semantically equivalent to the original one. Our program transformation approach is realized based on symbolic execution and deduction, and it contains two phases. The first phase is *symbolic execution* [Kin76] of the source code performed by *KeY* [BHS06],

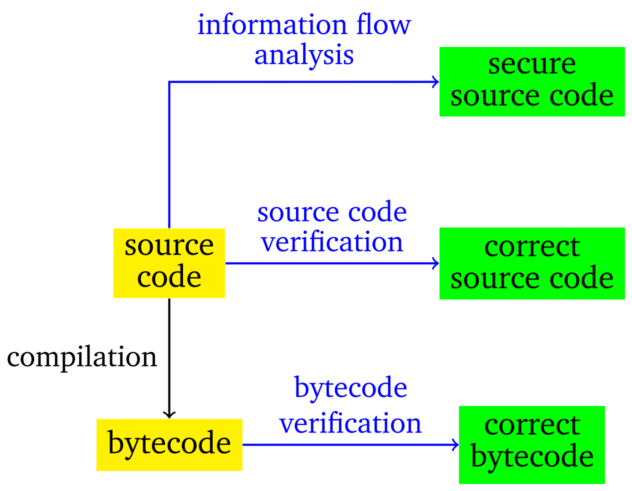

Figure 1.1: Software correctness and security: traditional approaches.

a state-of-the-art verification tool for Java programs. The symbolic execution is carried out by the application of *sequent calculus* rules, and the integrated first order deduction engine in KeY helps to achieve a precise analysis of variable dependencies, aliasing, and elimination of unfeasible paths. In the second phase, we extend the sequent calculus rules with suitable ingredients, e.g. *observable locations*, such that the rules can be applied reversely and the target program is generated in a step-wise manner. The soundness of our program transformation process is proved.

The result of the program transformation is a program that has the same behavior as (or is *bisimilar* to) the original program with respect to the observable locations. The soundness of the program transformation process guarantees the soundness of the generated program. It enjoys the following properties:

- The generated program is optimized over the original program for the sake of the first order logic reasoning and possible simple partial evaluation steps performed during symbolic execution.
- We can generate a program that is bisimilar to the original one with respect to certain observable locations, e.g., low sensitive variables. This helps to achieve an information flow analysis.
- We can generate bytecode form source code, so that if the source code is verified correctly, the bytecode is also correct without further verification. This is a deductive compilation approach.

Figure 1.2 gives an overview of the contributions of this thesis. To summarize:

- We interleave symbolic execution and partial evaluation to reduce the proof state space and speed up the proving process.

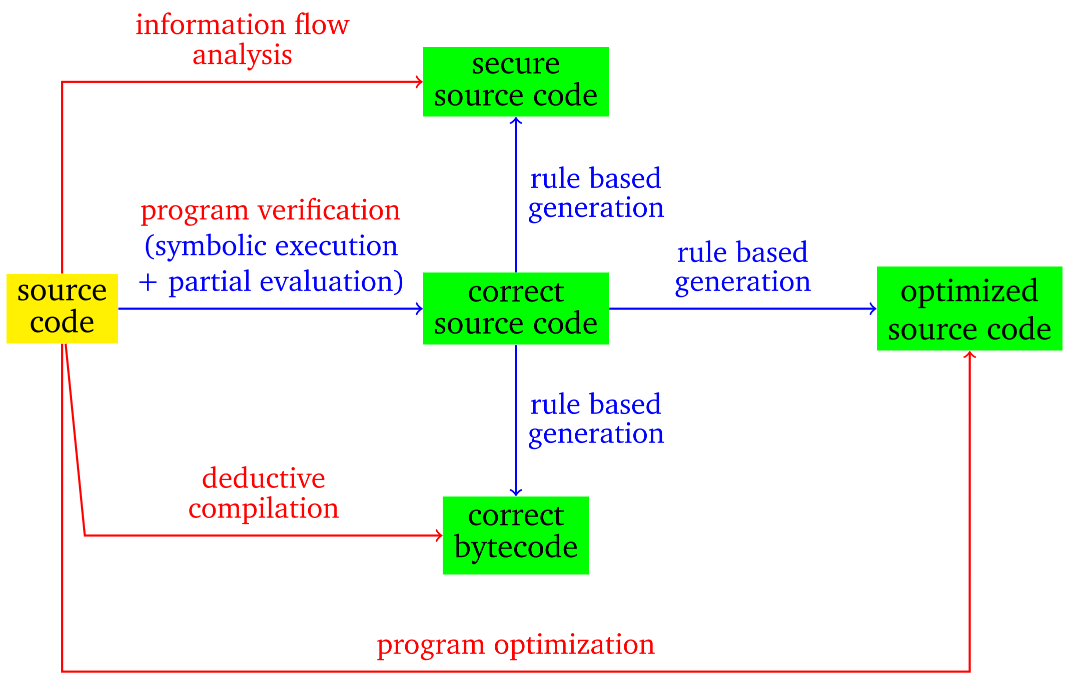

Figure 1.2: Software correctness and security: a uniform framework.

- We propose a sound approach of program transformation that ensures the correctness of the generated code with respect to the source code.
- We propose further optimization techniques for program transformation.
- We implement a partial evaluator for SiJa.
- We apply program transformation to information flow analysis.
- We apply program transformation to deductive compilation.

The following chapters are organized as follows:

- Chapter 2 gives the background of KeY and symbolic execution;
- Chapter 3 introduces partial evaluation and shows its interleaving with symbolic execution for speeding correctness proofs;
- Chapter 4 concentrates on the main approach of program transformation and its soundness proof;
- Chapter 5 shows the application of the program transformation to information flow;
- Chapter 6 gives an introduction of applying program transformation technique to bytecode compilation;
- Chapter 7 concludes the thesis and points out some further research directions.

1.3 Publications

Here is a list of publications related to this thesis work:

- *Interleaving Symbolic Execution and Partial Evaluation*. Richard Bubel and Reiner Hähnle and Ran Ji. 8th *International Symposium on Formal Methods for Components and Objects (FMCO)*. Eindhoven, the Netherlands. 2009. [BHJ09]

  It is concerned with the source code level verification for sequential Java programs. In this paper, we show that symbolic execution and partial evaluation not only are compatible with each other, but that there is considerable potential for synergies. Specifically, we integrate a simple partial evaluator for a Java-like language into the logic-based symbolic execution engine of the software verification tool KeY [BHS06]. This allows to interleave symbolic execution and partial evaluation steps within a uniform (logic-based) framework in a sound way. Intermittent partial evaluation during symbolic execution has the effect that the remaining program that is yet to be executed is continuously simplified relative to the current path conditions and the current symbolic state in each symbolic execution trace.

  I carried out the experiments and was involved in the paper writing.

- *Program Specialization Via a Software Verification Tool*. Richard Bubel and Reiner Hähnle and Ran Ji. 9th *International Symposium on Formal Methods for Components and Objects (FMCO)*. Graz, Austria. 2010. [BHJ10]

  We propose a new approach to specialize Java-like programs via the software verification tool KeY, in which a symbolic execution engine is used. It is a two-phase procedure that first symbolically executes the program interleaved with a simple partial evaluator, and then synthesizes the specialized program in the second phase. The soundness of the approach is guaranteed by a bisimulation relation on the source and specialized programs.

  I designed the main theory and was involved in the paper writing.

- *PE-KeY: A Partial Evaluator for Java Programs*. Ran Ji and Richard Bubel. 9th *International Conference on Integrated Formal Methods (iFM)*. Pisa, Italy. 2012. [JB12]

  In this paper we present a prototypical implementation of a partial evaluator for Java programs, named PE-KeY, based on the verification system KeY. We argue that using a program verifier as technological basis provides potential benefits leading to a higher degree of specialization. We discuss in particular how loop invariants and preconditions can be exploited to specialize programs. First experimental results are provided.

  I did the main implementation and was involved in the paper writing.

- *Program Transformation Based on Symbolic Execution and Deduction*. Ran Ji and Reiner Hähnle and Richard Bubel. 11th *International Conference on Software Engineering and Formal Methods (SEFM)*. Madrid, Spain. 2013. [JHB13]

  In this paper, we present a program transformation framework based on symbolic execution and deduction. Its virtues are: (i) behavior preservation of the transformed program is guaranteed by a sound program logic, and (ii) automated first-order solvers are used for simplification and optimization. Transformation consists of two phases: first the source program is symbolically executed by sequent calculus rules in a program logic. This involves a precise analysis of variable dependencies, aliasing, and elimination of unfeasible execution paths. In the second phase, the target program is synthesized by a leave-to-root traversal of the symbolic execution tree by backward application of (extended) sequent calculus rules. We prove soundness by a suitable notion of bisimulation and we discuss one possible approach to automated program optimization.

  I developed the main theory and was involved in the paper writing.

2 Background

2.1 KeY and Symbolic Execution

KeY [BHS06] is a deductive verification system for programs written in the *Java* language, or more precisely *Java Card* language [Mos05] that is roughly a subset of sequential Java with some smart card extensions. The *Java Modeling Language (JML)* [LBR03] is used as its specification language. On proving a program, KeY first translates the specifications into proof obligations, which are logic formulas whose logical validity corresponds to the correctness of the program with respect to the specification. The logic used is *dynamic logic* [Pra76, HKT00b], an extension of first-order predicate logic with modal operators that contain executable program fragments of some programming language. More specifically, KeY uses *Java Dynamic Logic (JavaDL)* in which the program fragments are written in Java. Based on the proof obligations, KeY acts as a theorem prover to perform deductive verification. A novel feature of JavaDL compared to other variants of dynamic logic is the use of state *updates* [Bec01, Rüm06], which capture the state changes during the program execution. The program is executed in a symbolic way such that the symbolic values of the program variables are used instead of the concrete ones. When the symbolic execution ends, the programs are removed completely from the JavaDL formulas and therefore the verification goal is reduced to prove the validity of first-order formulas with the help of some built-in theories. The proofs are usually performed by KeY itself, or handled by external *satisfiability modulo theories (SMT) solvers* such as *Simplify* [DNS05] or Z3 [dMB08]. In fact, KeY is not only a Java verifier, it also supports other useful features such as *test case generation* [EH07] and *symbolic visual debugging* [HBBR10] in its variants. The most recent version of KeY supports explicit memory heap reasoning [Wei11].

Tools for deductive verification of object-oriented programs that are similar to KeY include *KIV* [HHRS86, Ste04], *Jive* [MPH00] and *VeriFast* [JP08]. KIV uses a dynamic logic like KeY and performs verification within one prover. Jive uses a Hoare logic and employs a generic theorem prover, *Isabelle/HOL* [NPW02], or an SMT solver for proving program-independent properties. VeriFast works with *separation logic* [Rey02] and emphasizes on fast verification of C and Java.

While SMT solvers can be used in KeY as a trusted "black box" to gain possibly better automation and performance by sacrificing some traceability, they are treated as the primary foundation for several other tools. These include *ESC/Java* [FLL$^+$02], *ESC/Java2* [CK05], *Spec#* [BLS05, BFL$^+$11] and *Frama-C* [CKK$^+$12]. The common paradigm of them is *verification condition generation*, in which the program and its specification are translated into first-order formulas named *verification conditions* that are passed to an SMT solver. In practice, the source code is often compiled to an *intermediate language* before generating the verification condition. This allows the modularization of the verification system, however, as discussed earlier, the compilation itself is a non-deductive step, thus also needs to be verified. For instance, *Boogie* [BCD+05] is a tool used to generate verification conditions from the intermediate language for Spec#. Some other Java verifiers based on verification condition generation also use theorem provers for higher-order logic, in addition to SMT solvers, to prove the verification conditions, e.g., *JACK* [BRL03] and *Krakatoa/Why* [FM07]. The verification condition generation is usually not a deductive, rule-based process.

The mechanism used in KeY that corresponds to verification condition generation is *symbolic execution*. Dating back to its introduction in the 1970s [Kin76, Bur74, BEL75], symbolic execution has only recently been realized efficiently for industrially relevant programming languages. It is a central, very versatile program analysis technique that is used for formal program verification [BHS06, HRS87, PV04], extended static checking and verification [BLS05], debugging [Bau07], and automatic test case generation [dHT08, EH07]. In the last decade a number of efficient symbolic execution engines for real heap-based programming and intermediate languages were created including, besides KeY (for Java, C, ABS, see [BHS06]), KIV (for Java, see [Ste04]), Bogor/Kiasan (for BIR, see [DLR06]), Pex (for MSIL, see [dHT08]), VeriFast (for C, Java, see [JP08]) and COSTA/PET (for Java bytecode, see [AAG+07, AGZP10]).

The main idea of symbolic execution is to use *symbolic values*, instead of actual data, as input, and to represent the values of program variables as symbolic expressions. The output values computed by a program are expressed as a function of the input symbolic values.

The *state* of a symbolically executed program includes the (symbolic) values of program variables, a *path condition* and a program counter. The path condition is a (quantifier-free) boolean formula over the symbolic inputs which accumulates constraints that the inputs must satisfy in order for an execution to follow the particular path. The program counter defines the next statement to be executed. A *symbolic execution tree,* in which the nodes represent the program states, characterizes the execution paths followed during the symbolic execution of a program.

Symbolic execution in KeY is performed based on the *sequent calculus* rules for JavaDL, and therefore it is a deductive process compared to verification condition generation. In general, a sequent is an expression of the form $\Gamma \Longrightarrow \Delta$ with the *antecedent* Γ, and the *succedent* Δ being sets of formulas. A sequent has the same meaning as the formula

$$\bigwedge_{\phi \in \Gamma} \phi \rightarrow \bigvee_{\psi \in \Delta} \psi \ .$$

Sequent rules have the general form

$$\mathsf{name}\ \frac{s_1 \quad \cdots \quad s_n}{s}$$

where $s, s_1, \ldots, s_n$ are sequents. The sequents above the line are the rule's *premises* while sequent s is called the rule's *conclusion*. A *sequent proof* is a tree whose nodes are labeled with sequents and with a sequent whose validity is to be proven at its root. This *proof tree* is constructed by applying sequent rules r to leaf nodes n whose sequent matches the conclusion r. The premises of r are then added as children of n. A branch of a proof tree is *closed* if and only if it contains an application of an axiom. A proof tree is closed if and only if all its branches are closed.

Accordingly, sequent rules for JavaDL work on a *first active statement* `s` and a current *update* $\mathcal{U}$ in the following general form of a conclusion:

$$\Gamma \Longrightarrow \mathcal{U}[\pi\ \mathtt{s};\ \omega]\phi, \Delta$$

Path conditions are represented by suitable formulas and accumulate in the antecedent Γ, and ω is the remaining program. In addition, π stands for an inactive prefix containing labels, opening braces or method-frames. In this thesis, we do not write down π explicitly, but keep in mind this possible inactive prefix.

An example of symbolic execution is shown in the following program fragment that orders the values of `x` and `y`. After its execution, `x` is the maximum of x_0, y_0 and `y` their minimum.

```
int x = x0;
int y = y0;
if (x > y) {
  int t = x;
  x = y;
  y = t;
}
```

State *update* captures the state changes during program execution. We use location-value pairs to represent states in symbolic execution. The expression $\{\mathtt{l}_1 := t_1 \,||\, \cdots \,||\, \mathtt{l}_n := t_n\}$ denotes a symbolic state in which each program location of the form $\mathtt{l}_i$ has the expression t_i as its symbolic value. After symbolic execution of the first three statements of the program above we obtain the symbolic state $\mathcal{U} = \{\mathtt{x} := x_0 \,||\, \mathtt{y} := y_0\}$. Symbolic execution of the conditional splits the execution into two branches, because the value $x_0 > y_0$ of the guard expression is symbolic and cannot be reduced immediately. The value of the guard becomes a path condition relative to which symbolic execution continues. Under the path condition $P_1 \equiv x_0 > y_0$ the body of the conditional is executed which results in the final symbolic state $\mathcal{U}' = \{\mathtt{x} := y_0 \,||\, \mathtt{y} := x_0 \,||\, \mathtt{t} := x_0\}$. The other branch terminates immediately in state $\mathcal{U}$ under path condition $P_2 \equiv x_0 \leq y_0$.

Symbolic execution as realized in KeY makes sure that every possible branch of the execution is considered. With the help of the built-in reasoning system, e.g. preconditions, path conditions, invariants, some unfeasible paths can be eliminated and the branches to be proved are reduced. The work flow of KeY can be summarized in Figure. 2.1.

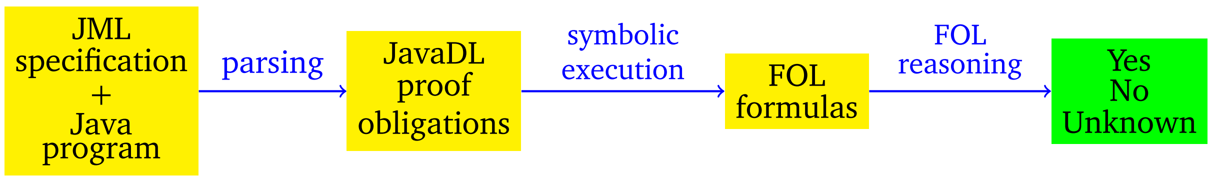

Figure 2.1: Work flow of KeY.

2.2 Programming Language

In this thesis, we use SiJa as the programming language. The object-oriented programming language SiJa is a simplified Java variant and closely related to the language defined in [BP06]. The differences to Java can be summarized as follows:

- *Unsupported Features.* Multi-threading, graphics, dynamic class loading, generic types or floating point datatypes are *not* supported by SiJa. Formal specification and verification of these features is a topic of ongoing research, therefore, left out completely.

- *Restricted Features.* For ease of presentation SiJa imposes some additional restrictions compared to Java. The KeY tool and the prototype implementation of our ideas evaluated in Chapter 3 do not impose these restrictions, but model and respect the Java semantics faithfully. The following restrictions apply to SiJa:

  Inheritance and Polymorphism. For the sake of a simple semantics for dynamic dispatch of method invocations SiJa abstains from Java-like interfaces and method overloading. Likewise, with exception of the `Null` type, the type hierarchy induced by user-defined class types has a tree structure with class `Object` as root.

  Prohibiting method overloading allows to identify a method within a class unambiguously by its name and number of parameters. We allow polymorphism (i.e. methods can be overwritten in subclasses) but require that their signature must be exactly the same, otherwise it is a compile-time error.

  Visibility. All classes, methods and fields are publicly visible. This restriction contributes also to a simpler dynamic dispatch semantics.

  No Exceptions. SiJa has no support for exceptions. Instead of runtime exceptions like `NullPointerExceptions` the program will simply not terminate in these cases.

  No class/object Initialization. In Java the first active usage of a type or creation of a new instance triggers complex initialization. SiJa supports only instance creation, but does not initialize fields upon creation. In particular, SiJa does not support static or instance initializers. User defined constructors are also missing in SiJa, a new instance is simply created by the expression `new` $T()$.

Primitive Types. Only `boolean` and `int` are available. To keep the semantics of standard arithmetic operators simple, `int` is an unlimited datatype representing the whole numbers $\mathbb{Z}$ rather than a finite datatype with overflow.

A **SiJa** program `p` is a non-empty set of class declarations with at least one class of name `Object`. The class hierarchy is a tree with class `Object` as root. A class $Cl := (cname, sname_{opt}, fld, mtd)$ consists of (i) a classname *cname* unique in `p`, (ii) the name of its superclass *sname* (only omitted for *cname* = `Object`), and (iii) a list of field *fld* and method *mtd* declarations.

The syntax for class declaration is the same as in Java. The only lacking features are constructors and static/instance initialization blocks. **SiJa** knows also the special reference type `Null` which is a singleton with `null` as the only element. It may be used in place of any reference type and is the only type that is a subtype of all class types.

To keep examples short we agree on the following convention: if not explicitly stated otherwise, any given sequence of statements is seen as if it would be the body of a static, void method declared in a class `Default` with no fields declared.

Dynamic dispatch works in **SiJa** as follows: we need to determine the implementation of a method on encountering a method invocation such as `o.m(a)`. To do so, first look up the dynamic type T of the object referenced by `o`. Then scan all classes between T and the static type of o for an implementation of a method named m and the correct number of parameters. The first match is taken.

The syntax of the executable fragment of **SiJa** is given in Figure 2.2.

Statements
```
stmnt ::= stmnt stmnt | lvarDecl | locExp'='exp';' | cond | loop
loop ::= while '('exp')' '{'stmnt'}'
lvarDecl ::= Type IDENT ('=' exp)_opt ';'
cond ::= if '('exp')' '{'stmnt'}' else '{'stmnt'}'
```
Expressions
```
exp ::= (exp.)_opt mthdCall | opExp | locExp
mthdCall ::= mthdName'('exp_opt(','exp)*')'
opExp ::= opr(exp_opt(,exp)*) | ℤ | TRUE | FALSE | null
opr ::= ! | - | < | <= | >= | > | == | && | || | + | - | * | / | % | ++
```
Locations
```
locExp ::= IDENT | exp.IDENT
```

Figure 2.2: Syntax of SiJa.

Figure 2.3 shows an example of **SiJa** program. It can be used in an online shopping session. If the customer buys at least 3 items and has a coupon, a 10% discount for all but the first two items will be granted.

```
public class OnLineShopping {
  boolean cpn;
  public int read() { /* read price of item */ }
  public int sum(int n) {
    int i = 1;
    int count = n;
    int tot = 0;
    while(i <= count) {
      int m = read();
      if(i >=3 && cpn) {
        tot = tot + m * 9 / 10;
        i++; }
      else {
        tot = tot + m;
        i++; }
    }
    return tot;
  }
}
```

Figure 2.3: A SiJa program fragment.

Any complex statement can be easily decomposed into a sequence of simpler statements without changing the meaning of a program, e.g., `y = z ++;` can be decomposed into `int t = z; z = z + 1; y = t;`, where `t` is a *fresh* variable, not used anywhere else. As we shall see later, a suitable notion of simplicity is essential, for example, to compute variable dependencies and simplify symbolic states. This is built into our semantics and calculus, so we need a precise definition of *simple statements*. In Figure 2.4, statements in the syntactic category *spStmnt* have at most one source of side effect each. This can be a non-terminating expression (such as a null pointer access), a method call, or an assignment to a location.

$\mathit{spStmnt} ::= \mathit{spLvarDecl} \mid \mathit{locVar}\texttt{'='}\mathit{spExp}\texttt{';'} \mid \mathit{locVar}\texttt{'='}\mathit{spAtr}\texttt{';'} \mid \mathit{spAtr}\texttt{'='}\mathit{spExp}\texttt{';'}$
$\mathit{spLvarDecl} ::= \mathit{Type}\ \texttt{IDENT';'}$
$\mathit{spExp} ::= (\mathit{locVar}\texttt{.})_{opt}\mathit{spMthdCall} \mid \mathit{spOpExp} \mid \mathit{litVar}$
$\mathit{spMthdCall} ::= \texttt{mthdName'('}\mathit{litVar}_{opt}\texttt{(','}\mathit{litVar})^{*}\texttt{')'}$
$\mathit{spOpExp} ::= \texttt{!}\mathit{litVar} \mid \texttt{-}\mathit{litVar} \mid \mathit{litVar}\ \mathit{binOpr}\ \mathit{litVar}$
$\mathit{litVar} ::= \mathit{litval} \mid \mathit{locVar}$ $\qquad \mathit{litval} ::= \mathbb{Z} \mid \texttt{TRUE} \mid \texttt{FALSE} \mid \textbf{null}$
$\mathit{binOpr} ::= \texttt{<} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{==} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}$
$\mathit{locVar} ::= \texttt{IDENT}$
$\mathit{spAtr} ::= \mathit{locVar}\texttt{.IDENT}$

Figure 2.4: Syntax of SiJa simple statements.

By decomposing every complex statement, a SiJa program p can be transformed into an equivalent (on the variables of p) program containing only simple statements. The program shown in Figure 2.5 has the same meaning as the program in Figure 2.3, but contains only simple statements.

```
public class OnLineShopping {
  boolean cpn;
  public int read() { /* read price of item */ }
  public int sum(int n) {
    int i = 1;
    int count = n;
    int tot = 0;
    while(i <= count) {
      int m;
      m = read();
      boolean b;
      b = i >= 3;
      boolean b1;
      b1 = b && cpn;
      if(b1) {
        int t;
        t = m * 9;
        int t1;
        t1 = t / 10;
        tot = tot + t1;
        i = i + 1; }
      else {
        tot = tot + m;
        i = i + 1; }
    }
    return tot;
  }
}
```

Figure 2.5: A SiJa program fragment contain only simple statements.

Because SiJa is a simple version of Java, the theories developed in this thesis are naturally applicable for this subset of Java. The implementation is integrated and evaluated in the KeY system.

2.3 Program Logic

Our program logic is *dynamic logic (DL)* [HKT00a]. The target program occurs in unencoded form as a first-class citizen inside the logic's connectives. Sorted first-order dynamic logic is sorted first-order logic that is syntactically closed with respect to the program correctness modalities $[\cdot]\cdot$ (box) and $\langle\cdot\rangle\cdot$ (diamond). The first argument is a program and the second a dynamic logic formula. Let p denote a program and ϕ a dynamic logic formula then $[\mathrm{p}]\phi$ and $\langle \mathrm{p}\rangle\phi$ are DL-formulas. Informally, the former expresses that if p is executed and terminates *then* in all reached final states ϕ holds; the latter means that if p is executed then it terminates *and* in at least one of the reached final states ϕ holds. The box modality expresses *partial correctness* of a program, while the diamond modality coincides with *total correctness*. Hoare logic [Hoa69] can be subsumed by dynamic logic since the Hoare triple $\{pre\}$ p $\{post\}$ can be expressed as the DL formula $pre \rightarrow [\mathtt{p}]post$.

We consider only deterministic programs, hence, a program p executed in a given state s *either* terminates and reaches exactly *one* final state *or* it does not terminate and there are no reachable final states.

A dynamic logic based on SiJa-programs is called *SiJa-DL*. The *signature* of the program logic depends on a *context SiJa-program* $\mathscr{C}$.

Definition 1 (SiJa-Signature $\Sigma_{\mathscr{C}}$). *A* signature $\Sigma_{\mathscr{C}} = (\mathsf{Sort}, \preceq, \mathsf{Pred}, \mathsf{Func}, \mathsf{LVar})$ *consists of:*

(i) *a set of names* Sort *called* sorts *containing at least one sort for each primitive type and one for each class Cl declared in* $\mathscr{C}$*:* $\mathsf{Sort} \supseteq \{\mathtt{int}, \mathtt{boolean}\} \cup \{Cl \mid \text{for all classes } Cl \text{ declared in } \mathscr{C}\}$*;*

(ii) *a* partial subtyping order $\preceq$: Sort × Sort *that models the subtype hierarchy of* $\mathscr{C}$ *faithfully;*

(iii) *a set of* predicate symbols $\mathsf{Pred} := \{p : T_1 \times \ldots \times T_n \mid T_i \in \mathsf{Sort}, n \in \mathbb{N}\}$. *We call* $\alpha(p) = T_1 \times \ldots \times T_n$ *the* signature *of the predicate symbol.*

(iv) *a set of* function symbols $\mathsf{Func} := \{f : T_1 \times \ldots \times T_n \rightarrow T \mid T_i, T \in \mathsf{Sort}, n \in \mathbb{N}\}$. *We call* $\alpha(f) = T_1 \times \ldots \times T_n \rightarrow T$ *the* signature *of the function symbol.* $\mathsf{Func} := \mathsf{Func}_r \cup \mathsf{PV} \cup \mathsf{Attr}$ *is further divided into disjoint subsets:*

   - *the* rigid *function symbols* Func_r*;*
   - *the* program variables $\mathsf{PV} = \{\mathtt{i}, \mathtt{j}, \ldots\}$, *which are* non-rigid *constants;*
   - *the* non-rigid *function symbols* attribute Attr*, such that for each attribute* a *of type T declared in class Cl an attribute function* $\mathtt{a}@Cl : Cl \rightarrow T \in \mathsf{Attr}$ *exists. We omit the @Cl from attribute function names if no ambiguity arises.*

(v) *a set of* logical variables $\mathsf{LVar} := \{x : T \mid T \in \mathsf{Sort}\}$.

We distinguish between *rigid* and *non-rigid* function and predicate symbols. Intuitively, the semantics of rigid symbols does not depend on the current state of program execution, while non-rigid symbols are state-dependent. Local program variables, static, and instance fields are modeled as non-rigid function symbols and together form a separate class of non-rigid symbols called *location* symbols. Specifically, local program variables and static fields are modeled as non-rigid constants, instance fields as unary non-rigid functions.

Example 1. *In the program shown in Figure 2.3,*

- `int, boolean` *are sorts;*

- `<=`, `&&` *are predicate symbols;*
- `+`, `=`, `*`, `/` *are rigid function symbols;*
- `i`, `count`, `tot` *are program variables;*

$\Pi_{\Sigma_{\mathscr{C}}}$ denotes the set of all executable SiJa programs (i.e., sequences of statements) with locations over signature $\Sigma_{\mathscr{C}}$. In this thesis, we use the notion of a program to refer to a sequence of executable SiJa-statements. If we want to include class, interface or method declarations, we either include them explicitly or make a reference to the context program $\mathscr{C}$.

The inductive definition of terms and formulas is standard, but we introduce a new syntactic category called *update* to represent state updates with symbolic expressions.

Definition 2 (Terms, Updates and Formulas)**.** *Terms t, updates u and formulas ϕ are well-sorted first-order expressions of the following kind:*

$$\begin{array}{lcl}
t & ::= & x \mid \mathtt{i} \mid t.\mathtt{a} \mid f(t,\ldots,t) \mid (\phi \;?\; t \;:\; t) \mid \\
 & & \mathbb{Z} \mid \mathtt{TRUE} \mid \mathtt{FALSE} \mid \mathtt{null} \mid \{u\}t \\
u & ::= & \mathtt{i} := t \mid t.\mathtt{a} := t \mid u \parallel u \mid \{u\}u \\
\phi & ::= & \mathit{true} \mid \mathit{false} \mid p(t,\ldots,t) \mid \neg\phi \mid \phi \circ \phi \; (\circ \in \{\wedge,\vee,\rightarrow,\leftrightarrow\}) \mid (\phi \;?\; \phi \;:\; \phi) \mid \\
 & & \forall x : T.\phi \mid \exists x : T.\phi \mid [\mathtt{p}]\phi \mid \langle\mathtt{p}\rangle\phi \mid \{u\}\phi
\end{array}$$

where $\mathtt{a} \in \mathsf{Attr}, f \in \mathsf{Func}, p \in \mathsf{Pred}, \mathtt{i} \in \mathsf{PV}, x : T \in \mathsf{LVar}$, *and* $\mathtt{p}$ *is a sequence of executable* SiJa *statements.*

An *elementary update* $\mathtt{i} := t$ or $t.\mathtt{a} := t$ is a pair of location and term. They are of *static single assignment (SSA)* form [AWZ88, RWZ88], with the same meaning as simple assignments. Elementary updates are composed to *parallel updates* $u_1 \| u_2$ and work like simultaneous assignments. Updates applied to terms or formulas are again terms or formulas.

Terms, formulas and updates are evaluated with respect to a SiJa-DL Kripke structure.

Definition 3 (Kripke structure)**.** *A SiJa-DL Kripke structure $\mathscr{K}_{\Sigma_{\mathsf{SiJa}}} = (\mathscr{D}, I, S)$ consists of*

(i) a set of elements $\mathscr{D}$ called domain,

(ii) an interpretation I with

- *$I(T) = \mathscr{D}_T$, $T \in \mathsf{Sort}$ assigning each sort its non-empty domain $\mathscr{D}_T$. It adheres to the restrictions imposed by the subtype order $\preceq$;* `Null` *is always interpreted as a singleton set and subtype of all class types;*
- *$I(f) : \mathscr{D}_{T_1} \times \ldots \times \mathscr{D}_{T_n} \rightarrow \mathscr{D}_T$ for each rigid function symbol $f : T_1 \times \ldots \times T_n \rightarrow T \in \mathsf{Func}_r$;*
- *$I(p) \subseteq \mathscr{D}_{T_1} \times \ldots \times \mathscr{D}_{T_n}$ for each predicate symbol $p : T_1 \times \ldots \times \; T_n \in \mathsf{Pred}$;*

(iii) a set of states S assigning meaning to non-rigid function symbols: let $s \in S$ then $s(\mathtt{a@}Cl) : \mathscr{D}_{Cl} \to \mathscr{D}_T$, $\mathtt{a@}Cl : Cl \to T \in \mathsf{Attr}$ and $s(\mathtt{i}) : \mathscr{D}_T$, $\mathtt{i} \in \mathsf{PV}$.

The pair $D = (\mathscr{D}, I)$ is called a first-order structure.

As usual in first-order logic, to define evaluation of terms and formulas in addition to a structure we need the notion of a *variable assignment*. A *variable assignment* $\beta : \mathsf{LVar} \to \mathscr{D}_T$ maps a logical variable $x : T$ to its domain $\mathscr{D}_T$.

Definition 4 (Evaluation function)**.** *A term, formula or update is evaluated relative to a given first-order structure $D = (\mathscr{D}, I)$, a state $s \in S$ and a variable assignment β, while programs and expressions are evaluated relative to a D and $s \in S$. The* evaluation function *val is defined recursively. It evaluates*

(i) every term $t : T$ to a value $val_{D,s,\beta}(t) \in \mathscr{D}_T$;

(ii) every formula ϕ to a truth value $val_{D,s,\beta}(\phi) \in \{tt, ff\}$;

(iii) every update u to a state transformer $val_{D,s,\beta}(u) \in S \to S$;

(iv) every expression $e : T$ to a set of pairs of state and value $val_{D,s}(e) \subseteq 2^{S \times T}$;

(v) every statement st to a set of states $val_{D,s}(st) \subseteq 2^S$.

Since SiJa is deterministic, all sets of states or state-value pairs have at most one element.

Figure 2.6 shows a collection of the semantic definition. The expression $s[\mathtt{x} \leftarrow \mathtt{v}]$ denotes a state coincides with s except at $\mathtt{x}$ which is mapped to the evaluation of $\mathtt{v}$.

Example 2 (Update semantics)**.** *We illustrate the semantics of updates of Figure 2.6. Evaluating $\{\mathtt{i} := \mathtt{j+1}\}\mathtt{i} \geq \mathtt{j}$ in a state s is identical to evaluating the formula $\mathtt{i} \geq \mathtt{j}$ in a state s' which coincides with s except for the value of $\mathtt{i}$ which is evaluated to the value of $val_{D,s,\beta}(\mathtt{j}+1)$. Evaluation of the parallel update $\mathtt{i} := \mathtt{j} \| \mathtt{j} := \mathtt{i}$ in a state s leads to the successor state s' identical to s except that the values of $\mathtt{i}$ and $\mathtt{j}$ are swapped. The parallel update $\mathtt{i} := 3 \| \mathtt{i} := 4$ has a* conflict *as $\mathtt{i}$ is assigned different values. In such a case the last occurring assignment $\mathtt{i} := 4$ overrides all previous ones of the same location. Evaluation of $\{\mathtt{i} := \mathtt{j}\}\{\mathtt{j} := \mathtt{i}\}\phi$ in a state s results in evaluating ϕ in a state, where $\mathtt{i}$ has the value of $\mathtt{j}$, and $\mathtt{j}$ remains unchanged.*

Remark. *$\{\mathtt{i} := \mathtt{j}\}\{\mathtt{j} := \mathtt{i}\}\phi$ is the sequential application of updates $\mathtt{i} := \mathtt{j}$ and $\mathtt{j} := \mathtt{i}$ on the formula ϕ. To ease the presentation, we overload the concept of update and also call $\{\mathtt{i} := \mathtt{j}\}\{\mathtt{j} := \mathtt{i}\}$ an update. In the following context, if not stated otherwise, we use the upper-case letter $\mathscr{U}$ to denote this kind of update, compared to the real update that is denoted by a lower-case letter u. An update $\mathscr{U}$ could be the of form $\{u\}$ and $\{u_1\} \ldots \{u_n\}$. Furthermore, $\{u_1\} \ldots \{u_n\}$ can be simplified into the form of $\{u\}$, namely the* normal form *(NF) of update.*

For terms:

$$val_{D,s,\beta}(\mathtt{TRUE}) = \mathit{True}$$
$$val_{D,s,\beta}(\mathtt{FALSE}) = \mathit{False},\ \text{where } \{\mathit{True}, \mathit{False}\} = D(\mathtt{boolean})$$
$$val_{D,s,\beta}(x) = \beta(x),\ x \in \mathsf{LVar}$$
$$val_{D,s,\beta}(\mathtt{x}) = s(\mathtt{x}),\ \mathtt{x} \in \mathsf{PV}$$
$$val_{D,s,\beta}(o.\mathtt{a}) = s(\mathtt{a})(val_{D,s,\beta}(o)),\ \mathtt{a} \in \mathsf{Attr}$$
$$val_{D,s,\beta}(f(t_1,\dots,t_n)) = D(f)(val_{D,s,\beta}(t_1),\dots,val_{D,s,\beta}(t_n))$$
$$val_{D,s,\beta}(\psi\ ?\ t_1\ :\ t_2) = \begin{cases} val_{D,s,\beta}(t_1) & \text{if } val_{D,s,\beta}(\psi) = tt \\ val_{D,s,\beta}(t_2) & \text{otherwise} \end{cases}$$
$$val_{D,s,\beta}(\{u\}t) = val_{D,s',\beta}(t),\ s' = val_{D,s,\beta}(u)(s)$$

For formulas:

$$val_{D,s,\beta}(\mathit{true}) = tt$$
$$val_{D,s,\beta}(\mathit{false}) = ff$$
$$val_{D,s,\beta}(p(t_1,\dots,t_n)) = tt \text{ iff } (val_{D,s,\beta}(t_1),\dots,val_{D,s,\beta}(t_n)) \in D(p)$$
$$val_{D,s,\beta}(\neg\phi) = tt \text{ iff } val_{D,s,\beta}(\phi) = ff$$
$$val_{D,s,\beta}(\psi \wedge \phi) = tt \text{ iff } val_{D,s,\beta}(\psi) = tt \text{ and } val_{D,s,\beta}(\psi) = tt$$
$$val_{D,s,\beta}(\psi \vee \phi) = tt \text{ iff } val_{D,s,\beta}(\psi) = tt \text{ or } val_{D,s,\beta}(\psi) = tt$$
$$val_{D,s,\beta}(\psi \rightarrow \phi) = val_{D,s,\beta}(\neg\psi \vee \phi)$$
$$val_{D,s,\beta}(\psi \leftrightarrow \phi) = val_{D,s,\beta}(\psi \rightarrow \phi \wedge \phi \rightarrow \psi)$$
$$val_{D,s,\beta}([\mathtt{p}]\phi) = tt \text{ iff } ff \notin \{val_{D,s',\beta}(\phi) | s' \in val_{D,s}(\mathtt{p})\}$$
$$val_{D,s,\beta}(\{u\}\phi) = val_{D,s',\beta}(\phi),\ \text{where } s' = val_{D,s,\beta}(u)(s)$$

For updates:

$$val_{D,s,\beta}(\mathtt{x} := t)(s) = s[\mathtt{x} \leftarrow t]$$
$$val_{D,s,\beta}(o.\mathtt{a} := t)(s) = s[(\mathtt{a})(val_{D,s,\beta}(o)) \leftarrow t]$$
$$val_{D,s,\beta}(u_1 \| u_2)(s) = val_{D,s,\beta}(u_2)(val_{D,s,\beta}(u_1)(s))$$
$$val_{D,s,\beta}(\{u_1\}u_2)(s) = val_{D,s',\beta}(u_2)(s'),\ \text{where } s' = val_{D,s,\beta}(u_1)(s)$$

For expressions:

$$val_{D,s}(\mathtt{x}) = \{(s, s(\mathtt{x}))\},\ \mathtt{x} \in \mathsf{PV}$$
$$val_{D,s}(o.\mathtt{a}) = \{(s', s(a)(d)) \mid (s', d) \in val_{D,s}(o) \wedge d \neq \mathit{null}\}$$
$$val_{D,s}(e_1 \circ e_2) = \{(s'', D(\circ)(d_1, d_2)) \mid (s', d_1) \in val_{D,s}(e_1) \wedge (s'', d_2) \in val_{D,s'}(e_2)\}$$
$$\circ \in \{+, -, *, \dots\}$$

For statements:

$$val_{D,s}(\mathtt{x} = e) = \{s'[\mathtt{x} \leftarrow d] \mid (s', d) \in val_{D,s}(e)\},\ \mathtt{x} \in \mathsf{PV}$$
$$val_{D,s}(o.\mathtt{a} = e) = \{s''[\mathtt{a}(d_o) \leftarrow d_e] \mid (s', d_o) \in val_{D,s}(o) \wedge (s'', d_e) \in val_{D,s'}(e)\}$$
$$val_{D,s}(\mathtt{p}_1;\mathtt{p}_2) = \bigcup_{s' \in val_{D,s}(\mathtt{p}_1)} val_{D,s'}(\mathtt{p}_2)$$
$$val_{D,s}(\mathtt{if}(e)\ \{\mathtt{p}\}\ \mathtt{else}\ \{\mathtt{q}\}) = \begin{cases} val_{D,s',\beta}(\mathtt{p}), & (s', \mathit{True}) \in val_{D,s}(e) \\ val_{D,s',\beta}(\mathtt{q}), & (s', \mathit{False}) \in val_{D,s}(e) \\ \emptyset, & \text{otherwise} \end{cases}$$
$$val_{D,s}(\mathtt{while}(e)\{\mathtt{p}\}) = \begin{cases} \bigcup_{s_1 \in S_1} val_{D,s_1}(\mathtt{while}(e)\{\mathtt{p}\}) \text{ where } S_1 = val_{D,s'}(\mathtt{p}), \\ \qquad \text{if } (s', \mathit{True}) \in val_{D,s}(e) \\ \{s'\},\ \text{if } (s', \mathit{False}) \in val_{D,s}(e) \\ \emptyset,\ \text{otherwise} \end{cases}$$

Figure 2.6: Definition of SiJa-DL semantic evaluation function.

Definition 5 (Normal form of update). *An update is in* normal form*, denoted by $\mathcal{U}^{nf}$, if it has the shape $\{u_1 \| \dots \| u_n\}$, $n \geq 0$, where each u_i is an elementary update and there is no conflict between u_i and u_j for any $i \neq j$.*

Example 3 (Normal form of update). *For the following updates,*

- $\{\mathtt{i} := \mathtt{j} + 1\}$ *and* $\{\mathtt{i} := \mathtt{j} + 1 \| \mathtt{j} := \mathtt{i}\}$ *are in normal form.*
- $\{\mathtt{i} := \mathtt{j} + 1\}\{\mathtt{j} := \mathtt{i}\}$ *is not in normal form.*
- $\{\mathtt{i} := \mathtt{j} + 1 \| \mathtt{j} := \mathtt{i} \| \mathtt{i} := \mathtt{i} + 1\}$ *is not in normal form, because there is a conflict between* $\mathtt{i} := \mathtt{j} + 1$ *and* $\mathtt{i} := \mathtt{i} + 1$.

The normal form of an update $\mathcal{U} = \{u_1\} \dots \{u_n\}$ can be achieved by applying a sequence of *update simplification* steps shown in Figure 2.7. Soundness of these rules and that they achieve normal form are proven in [Rüm06].

$$
\begin{array}{l}
\{\dots \| \mathtt{x} := v_1 \| \dots \| \mathtt{x} := v_2 \| \dots\} v \rightsquigarrow \{\dots \| \dots \| \dots \| \mathtt{x} := v_2 \| \dots\} v \\
\quad \text{where } v \in t \cup f \cup \phi \\
\{\dots \| \mathtt{x} := v' \| \dots\} v \rightsquigarrow \{\dots \| \dots\} v, \text{ where } v \in t \cup f \cup \phi,\ \mathtt{x} \notin fpv(v) \\
\{u\}\{u'\} v \rightsquigarrow \{u \| \{u\} u'\} v, \text{ where } v \in t \cup f \cup \phi \\
\{u\} x \rightsquigarrow x, \text{ where } x \in \mathsf{LVar} \\
\{u\} f(t_1, \dots, t_n) \rightsquigarrow f(\{u\}(t_1), \dots, \{u\}(t_n)) \\
\{u\} \neg \phi \rightsquigarrow \neg \{u\} \phi \\
\{u\}(\phi_1 \circ \phi_2) \rightsquigarrow \{u\}(\phi_1) \circ \{u\}(\phi_2), \text{ where } \circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\} \\
\{u\}(\mathtt{x} := v) \rightsquigarrow \mathtt{x} := \{u\} v \\
\{u\}(o.\mathtt{a} := v) \rightsquigarrow o.\mathtt{a} := \{u\} v \\
\{u\}(u_1 \| u_2) \rightsquigarrow \{u\} u_1 \| \{u\} u_2 \\
\{\mathtt{x} := v\} \mathtt{x} \rightsquigarrow v \\
\{o.\mathtt{a} := v\} o.\mathtt{a} \rightsquigarrow v
\end{array}
$$

Figure 2.7: Update simplification rules.

Finally, we give the definitions of satisfiability, model and validity of formulas.

Definition 6 (Satisfiability, model and validity). *A formula ϕ*

- *is* satisfiable*, denoted by $D, s, \beta \models \phi$, if there exists a first-order structure D, a state $s \in S$ and a variable assignment β with $val_{D,s,\beta}(\phi) = tt$.*
- *has a* model*, denoted by $D, s \models \phi$, if there exists a first-order structure D, a state $s \in S$, such that for all variable assignments β: $val_{D,s,\beta}(\phi) = tt$ holds.*

- *is* valid, *denoted by* $\models \phi$, *if for all first-order structures* D, *states* $s \in S$ *and for all variable assignments* β: $val_{D,s,\beta}(\phi) = tt$ *holds.*

We also introduce two other notions which will be used later.

Definition 7 (Signature Extension). *Let* Σ, Σ' *denote two signatures.* Σ' *is called a* signature extension *of* Σ *if there is an embedding* $\sigma(\Sigma) \subset \Sigma'$ *that is unique up to isomorphism and enjoys the following properties:*

- $\sigma(\mathsf{Sort}_\Sigma) = \mathsf{Sort}_{\Sigma'}$
- $\sigma(\mathsf{Func}_\Sigma) \subseteq \mathsf{Func}_{\Sigma'}$ *where for any arity countably infinite additional function symbols exist (analogously for predicates and logic variables)*
- $\sigma(\Pi_\Sigma) \subseteq \Pi_{\Sigma'}$

An important property of signature extensions is the following:

Lemma 1. *Let* $\Sigma' \supseteq \Sigma$ *denote a signature extension as described in Definition 7. If a* SiJa-DL*-formula* ϕ *over* Σ *has a counter example, i.e., a* SiJa-DL*-Kripke structure* $\mathcal{K}_\Sigma$, $s \in S_\Sigma$ *with* $\mathcal{K}, s \not\models \phi$, *then* $\sigma(\mathcal{K}, s) \not\models \phi$ *. In words, signature extensions are counter example preserving.*

Finally, we define the notion of an *anonymizing update*. The motivation behind anonymizing updates is to erase knowledge about the values of the fields included in the *modifier set mod* of locations that can be modified by a program. This is achieved by assigning fresh constant or function symbols to those locations. For example, an anonymizing update for $mod_\Sigma = \{\mathtt{i}, \mathtt{j}\}$ is $\{\mathtt{i} := c_i \,||\, \mathtt{j} := c_j\}$ where c_i, c_j are constants freshly introduced in the extended signature Σ'.

Definition 8 (Anonymizing Update). *Let mod denote a set of terms built from location symbols in* Σ. *An* anonymizing update for *mod is an update* $\mathcal{V}_{mod}$ *over an extended signature* Σ' *assigning each location* $l(t_1, \ldots, t_n) \in mod$ *a term* $f_l'(t_1, \ldots, t_n)$ *where* $f_l' \in \Sigma' \setminus \Sigma$.

2.4 Sequent Calculus

To analyze a SiJa-DL formula for validity, we use a Gentzen style sequent calculus. A sequent

$$\underbrace{\phi_1, \ldots, \phi_n}_{\Gamma} \Longrightarrow \underbrace{\psi_1, \ldots, \psi_m}_{\Delta}$$

is a pair of sets of formulas Γ (antecedent) and Δ (succedent). Its meaning is identical to the meaning of the formula

$$\bigwedge_{\phi \in \Gamma} \phi \rightarrow \bigvee_{\psi \in \Delta} \psi$$

A sequent calculus rule

$$\text{rule}\ \frac{\overbrace{\Gamma_1 \Longrightarrow \Delta_1 \quad \ldots \quad \Gamma_n \Longrightarrow \Delta_n}^{\textit{premises}}}{\underbrace{\Gamma \Longrightarrow \Delta}_{\textit{conclusion}}}$$

consists of one conclusion and possibly many premises. One example of a schematic sequent calculus rule is the rule andRight:

$$\text{andRight}\ \frac{\Gamma \Longrightarrow \phi,\ \Delta \qquad \Gamma \Longrightarrow \psi,\ \Delta}{\Gamma \Longrightarrow \phi \land \psi,\ \Delta}$$

We call ϕ and ψ *schema variables* which match here any arbitrary formula. A rule is applied on a sequent s by matching its conclusion against s. The instantiated premises are then added as children of s. For example, when applying andRight to the sequent $\Longrightarrow \texttt{i} \geq 0 \land \neg \texttt{o.a} \doteq \texttt{null}$ we instantiate ϕ with $\texttt{i} \geq 0$ and ψ with $\neg \texttt{o.a} \doteq \texttt{null}$. Here, $\doteq$ is an equality predicate symbol. The instantiated sequents are then added as children to the sequent and the resulting partial proof tree becomes:

$$\frac{\Longrightarrow \texttt{i} \geq 0 \qquad \Longrightarrow \neg \texttt{o.a} \doteq \texttt{null}}{\Longrightarrow \texttt{i} \geq 0 \land \neg \texttt{o.a} \doteq \texttt{null}}$$

Figure 2.8 shows a selection of first-order sequent calculus rules. A proof of the validity of a formula ϕ in a sequent calculus is a tree where

- each node is annotated with a sequent,
- the root is labeled with $\Longrightarrow \phi$,
- for each inner node n: there is a sequent rule whose conclusion matches the sequent of n and there is a bijection between the rule's premises and the children of n, and,
- the last rule application on each branch is the application of a close rule (axiom).

So far the considered rules were pure first-order reasoning rules. The calculus design regarding rules for formulas with programs is discussed next. Since in most cases the partial correctness of programs (without termination) is our main concern, we consider only the box modality variant of these rules.

Our sequent calculus variant is designed to symbolically execute a program in a step-wise manner. It behaves for most parts as a symbolic program interpreter. A *sequent* for SiJa-DL is of the form

$$\Gamma \Longrightarrow \mathcal{U}[\texttt{p}]\phi, \Delta$$

<u>Axioms</u>

$$\text{close}\ \frac{*}{\phi \Longrightarrow \phi} \qquad \text{closeTrue}\ \frac{*}{\Longrightarrow \mathit{true}} \qquad \text{closeFalse}\ \frac{*}{\mathit{false} \Longrightarrow}$$

<u>Propositional Rules</u>

$$\text{andLeft}\ \frac{\Gamma, \psi, \phi \Longrightarrow \Delta}{\Gamma, \phi \wedge \psi \Longrightarrow \Delta} \qquad \text{orRight}\ \frac{\Gamma \Longrightarrow \phi, \psi, \Delta}{\Gamma \Longrightarrow \phi \vee \psi,\ \Delta} \qquad \text{impRight}\ \frac{\Gamma, \phi \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \rightarrow \psi,\ \Delta}$$

$$\text{andRight}\ \frac{\Gamma \Longrightarrow \phi,\ \Delta \qquad \Gamma \Longrightarrow \psi,\ \Delta}{\Gamma \Longrightarrow \phi \wedge \psi,\ \Delta} \qquad \text{orLeft}\ \frac{\Gamma, \phi \Longrightarrow \Delta \qquad \Gamma, \psi \Longrightarrow \Delta}{\Gamma, \phi \vee \psi,\ \Delta \Longrightarrow}$$

<u>First-Order Rules</u>

$$\text{allLeft}\ \frac{\Gamma, \phi[x/t] \Longrightarrow \Delta}{\Gamma, \forall x : T.\phi \Longrightarrow \Delta} \qquad \text{exRight}\ \frac{\Gamma \Longrightarrow \phi[x/t], \Delta}{\Gamma \Longrightarrow \exists x : T.\phi\ \Delta}$$

$$\text{allRight}\ \frac{\Gamma \Longrightarrow \phi[x/c], \Delta}{\Gamma \Longrightarrow \forall x : T.\phi,\ \Delta} \qquad \text{exLeft}\ \frac{\Gamma, \phi[x/c] \Longrightarrow \Delta}{\Gamma, \exists x : T.\phi \Longrightarrow \Delta}$$

$$c \text{ new},\ \text{freeVars}(\phi) = \emptyset$$

Figure 2.8: First-order calculus rules (excerpt).

The general form of sequent calculus rules for SiJa-DL is:

$$\text{ruleName}\ \frac{\Gamma_1 \Longrightarrow \mathcal{U}_1[\mathrm{p}_1]\phi_1, \Delta_1 \quad \dots \quad \Gamma_n \Longrightarrow \mathcal{U}_n[\mathrm{p}_\mathrm{n}]\phi_n, \Delta_n}{\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}]\phi, \Delta}$$

During symbolic execution performed by the sequent rules, the antecedents Γ accumulate path conditions and contain possible preconditions. The updates $\mathcal{U}$ record the current symbolic value at each point during program execution and the ϕ's represent postconditions.

We explain the core concepts along a few selected rules. Starting with the `assignment` rule:

$$\text{assignment}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathrm{x} := \mathit{litVar}\}[\omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathrm{x} = \mathit{litVar}; \omega]\phi, \Delta}$$

where $\mathrm{x} \in \mathsf{PV}$, and *litVar* is either a boolean/integer literal or a program variable, and ω the rest of the program. The assignment rule works as most program rules on the first active statement ignoring the rest of the program (collapsed into ω). Its effect is the movement of the elementary program assignment into an update.

The assignment rule for an elementary addition is similar and looks like

$$\text{assignAddition}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{x} := \mathit{litVar}_1 + \mathit{litVar}_2\}[\omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{x} = \mathit{litVar}_1 + \mathit{litVar}_2; \omega]\phi, \Delta}$$

There is a number of other assignment rules for the different program expressions. All of the assignment rules have in common that they operate on elementary (pure) expressions. This is necessary to reduce the number of rules and also as expressions may have side-effects that need to be "computed" first. Our calculus works in two phases: first complex statements and expressions are decomposed into a sequence of simpler statements, then they are moved to an assignment or are handled by other kinds of rules (e.g., a loopInvariant rule). The decomposition phase consist mostly of so called unfolding rules such as:

$$\text{assignAdditionUnfold}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\mathtt{T}_{\mathit{exp}_1}\ \mathtt{v}_1 = \mathit{exp}_1; \mathtt{T}_{\mathit{exp}_2}\ \mathtt{v}_2 = \mathit{exp}_2;\ \mathtt{x} = \mathtt{v}_1 + \mathtt{v}_2; \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{x} = \mathit{exp}_1 + \mathit{exp}_2; \omega]\phi, \Delta}$$

where exp_1, exp_2 are arbitrary (nested) expressions of type T_{exp_1}, T_{exp_2}; and $\mathtt{v}_1, \mathtt{v}_2$ *fresh* program variables not yet used in the proof or in ω.

The conditional rule is a typical representative of a program rule to show how splits in control flows are treated:

$$\text{ifElse}\ \frac{\Gamma, \mathcal{U}\mathtt{b} \Longrightarrow \{\mathcal{U}\}[\mathtt{p}; \omega]\phi, \Delta \qquad \Gamma, \mathcal{U}\neg\mathtt{b} \Longrightarrow \{\mathcal{U}\}[\mathtt{q}; \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{if\ (b)\ \{p\}\ else\ \{q\}}\ \omega]\phi, \Delta}$$

where `b` is a program variable.

The calculus provides two different kinds of rules to treat loops. The first one realizes—as one would expect from a program interpreter—a simple unwinding of the loop:

$$\text{loopUnwind}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\mathtt{if\ (b)\ \{p; while\ (b)\ \{p\}\}}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}}\ \omega]\phi, \Delta}$$

The major drawback of this rule is that except for cases where the loop has a fixed and known number of iterations, the rule can be applied arbitrarily often. Instead of unwinding the loop, one often used alternative is the loop invariant rule $\mathsf{loopInvariant}$:

$$\mathsf{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U} inv, \Delta & \text{(init)}\\ \Gamma, \mathcal{U}\mathcal{V}_{mod}(\mathtt{b} \wedge inv) \Longrightarrow \mathcal{U}\mathcal{V}_{mod}[\mathtt{p}]inv, \Delta & \text{(preserves)}\\ \Gamma, \mathcal{U}\mathcal{V}_{mod}(\neg\mathtt{b} \wedge inv) \Longrightarrow \mathcal{U}\mathcal{V}_{mod}[\omega]\phi, \Delta & \text{(use case)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while\ (b)\ \{p\}}\ \omega]\phi, \Delta}$$

The loop invariant rule requires the user to provide a sufficiently strong formula *inv* capturing the functionality of the loop. The formula needs to be valid before the loop is executed (`init` branch) and must not be invalidated by any loop iteration started from a state satisfying the loop condition (`preserves` branch). Finally, in the third branch the symbolic execution continues with the remaining program after the loop.

The *anonymizing update* $\mathcal{V}_{mod}$ requires further explanation. We have to show that *inv* is preserved by an arbitrary iteration of the loop body as long as the loop condition is satisfied. But in an arbitrary iteration, values of program variables may have changed and outdated the information provided by Γ, Δ and $\mathcal{U}$. In traditional loop invariant rules, this context information is removed completely and the still valid portions have to be added to the invariant formula *inv*. We use the approach described in [BHS07] and avoid to invalidate all previous knowledge. For this we require the user to provide a superset of all locations *mod* that are potentially changed by the loop. The anonymizing update $\mathcal{V}_{mod}$ erases all knowledge about these locations by setting them to a fixed, but unknown value. An overapproximation of *mod* can be computed automatically.

The last rule we want to introduce is about method contracts and it is a necessity to achieve modularity in program verification. More important for this thesis is that it allows to achieve a modular program transformation scheme. Given a method $\mathtt{T}\ \mathtt{m}(\mathtt{T}\ \mathtt{param}_1, \ldots, \mathtt{T}_n\ \mathtt{param}_n)$ and a method contract

$$C(m) = (pre(\mathtt{param}_1, \ldots, \mathtt{param}_n), post(\mathtt{param}_1, \ldots, \mathtt{param}_n, \mathtt{res}), mod)$$

The formulas *pre* and *post* are the precondition and postcondition of the method with access to the parameters and to the result variable `res` (the latter only in *post*). The location set *mod* describes the locations (fields) that may be changed by the method. When we encounter a method invocation, the calculus first unfolds all method arguments. After that the method contract rule is applicable:

$$\text{methodContract}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\{\mathtt{param}_1 := \mathtt{v}_1 \| \ldots \| \mathtt{param}_n := \mathtt{v}_n\} pre, \Delta \\ \Gamma \Longrightarrow \mathcal{U}\{\mathtt{param}_1 := \mathtt{v}_1 \| \ldots \| \mathtt{param}_n := \mathtt{v}_n\} \mathcal{V}_{mod}(post \rightarrow [\mathtt{r} = \mathtt{res}; \omega]\phi), \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{r} = m(\mathtt{v}_1, \ldots, \mathtt{v}_n);\ \omega]\phi, \Delta}$$

In the first branch we have to show that the precondition of the method is satisfied. The second branch then allows us to assume that the postcondition is valid and we can continue to symbolically execute the remaining program. The anonymizing update $\mathcal{V}_{mod}$ erases again all information about the locations that may have been changed by the method. About the values of these locations, the information encoded in the postcondition is the only knowledge that is available and on which we can rely in the remaining proof.

Figure 2.9 gives a selection of sequent calculus rules, more detail can be found in [BHS07]. Some decomposition rules are given in Figure 2.10.

Symbolic execution of a program works as follows:

1) Select an open proof goal with a $[\cdot]$ modality. If no $[\cdot]$ exists on any branch, then symbolic execution is completed. Focus on the first active statement (possibly empty) of the program in the modality.

2) If it is a complex statement, apply rules to decompose it into simple statements and goto 1), otherwise continue.

3) Apply the sequent calculus rule corresponding to the first active statement.

4) Simplify the resulting updates using update simplification rules given in Figure 2.7, and apply first-order simplification to the premises. This might result in some closed branches. It is possible to detect and eliminate infeasible paths in this way. This step is optional.

5) Goto 1).

Example 4. *We look at typical proof goals that arise during symbolic execution:*

1. $\Gamma, \mathtt{i} > \mathtt{j} \Rightarrow \mathcal{U}[\mathtt{if\ (i>j)\ \{p\}\ else\ \{q\}}\ \omega]\phi$.

   *Applying rule **ifElse** and simplification eliminates the* `else` *branch and symbolic execution continues with* `p` ω*.*

2. $\Gamma \Rightarrow \{\mathtt{i} := \mathtt{c} \| \ldots\}[\mathtt{j\ =\ i;}\ \ \omega]\phi$ *where* `c` *is a constant.*

   It is sound to replace the statement $\mathtt{j} = \mathtt{i}$ *with* $\mathtt{j} = \mathtt{c}$ *and continue with symbolic execution. This is known as* constant propagation. *Chapter 3 shows more actions for* partial evaluation *that can be integrated into symbolic execution.*

$$\text{emptyBox}\ \frac{\Gamma \Longrightarrow \mathcal{U}\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\,]\phi, \Delta}$$

$$\text{assignment}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{x} := \mathit{litVar}\}[\omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{x} = \mathit{litVar};\, \omega]\phi, \Delta}$$

$$\text{assignAddition}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{x} := \mathit{litVar}_1 + \mathit{litVar}_2\}[\omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{x} = \mathit{litVar}_1 + \mathit{litVar}_2;\, \omega]\phi, \Delta}$$

$$\text{writeAttribute}\ \frac{\Gamma, \mathcal{U}\neg(\mathtt{o} \doteq \mathtt{null}) \Longrightarrow \mathcal{U}\{\mathtt{o.a} := \mathit{se}\}[\omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{o.a} = \mathit{se};\ \omega]\phi, \Delta}$$

$$\text{ifElse}\ \frac{\Gamma, \mathcal{U}\mathtt{b} \Longrightarrow \mathcal{U}[\mathtt{p};\omega]\phi, \Delta \qquad \Gamma, \mathcal{U}\neg\mathtt{b} \Longrightarrow \mathcal{U}[\mathtt{q};\omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{if}\ (\mathtt{b})\ \{\mathtt{p}\}\ \mathtt{else}\ \{\mathtt{q}\}\ \omega]\phi, \Delta}$$

$$\text{loopUnwind}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\mathtt{if}\ (\mathit{exp})\ \{\mathtt{p};\mathtt{while}\ (\mathit{exp})\ \{\mathtt{p}\}\}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while}\ (\mathit{exp})\ \{\mathtt{p}\}\ \omega]\phi, \Delta}$$

$$\text{loopInvariant}\ \frac{\begin{array}{ll} \Gamma \Longrightarrow \mathcal{U}\mathit{inv}, \Delta & \text{(init)} \\ \Gamma, \mathcal{U}\mathcal{V}_{mod}(\mathtt{b} \wedge \mathit{inv}) \Longrightarrow \mathcal{U}\mathcal{V}_{mod}[\mathtt{p}]\mathit{inv}, \Delta & \text{(preserves)} \\ \Gamma, \mathcal{U}\mathcal{V}_{mod}(\neg\mathtt{b} \wedge \mathit{inv}) \Longrightarrow \mathcal{U}\mathcal{V}_{mod}[\omega]\phi, \Delta & \text{(use case)} \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while}\ (\mathtt{b})\ \{\mathtt{p}\}\ \omega]\phi, \Delta}$$

$$\text{methodInvocation}\ \frac{\begin{array}{l} \Gamma, \mathcal{U}\neg(\mathtt{o} \doteq \mathtt{null}) \Longrightarrow \{\mathcal{U}\}[\\ \qquad \mathtt{if}\ (\mathtt{o}\ \mathtt{instanceof}\ \mathtt{T}_n)\ \mathtt{res} = \mathrm{o.m(se)@T}_n; \\ \qquad \mathtt{else}\ \mathtt{if}(\mathtt{o}\ \mathtt{instanceof}\ \mathtt{T}_{n-1})\ \mathtt{res} = \mathrm{o.m(se)@T}_{n-1}; \\ \qquad \ldots \\ \qquad \mathtt{else}\ \mathtt{res} = \mathrm{o.m(se)@T}_1; \\ \quad \omega]\phi, \Delta \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{res} = \mathrm{o.m(se)};\ \omega]\phi, \Delta}$$

$$\text{methodContract}\ \frac{\begin{array}{l} \Gamma \Longrightarrow \mathcal{U}\{\mathtt{param}_1 := \mathtt{v}_1 \| \ldots \| \mathtt{param}_n := \mathtt{v}_n\}\mathit{pre}, \Delta \\ \Gamma \Longrightarrow \mathcal{U}\{\mathtt{param}_1 := \mathtt{v}_1 \| \ldots \| \mathtt{param}_n := \mathtt{v}_n\}\mathcal{V}_{mod}(\mathit{post} \rightarrow [\mathtt{r} = \mathtt{res};\omega]\phi), \Delta \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{r} = m(\mathtt{v}_1, \ldots, \mathtt{v}_n);\ \omega]\phi, \Delta}$$

Figure 2.9: Selected sequent calculus rules.

For decomposition of complex expressions:

$$\text{postInc}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\mathtt{T_y\ v_1 = y; y = y + 1; x = v_1;} \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{x = y++;} \omega]\phi, \Delta}$$

$$\text{assignAdditionUnfold}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\mathtt{T}_{exp_1}\ \mathtt{v_1} = exp_1; \mathtt{T}_{exp_2}\ \mathtt{v_2} = exp_2;\ \mathtt{x = v_1 + v_2;} \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{x} = exp_1 + exp_2; \omega]\phi, \Delta}$$

$$\text{writeAttributeUnfold}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\mathtt{T}_{nse}\ \mathtt{v_1} = nse; \mathtt{v_1.a} = se;\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[nse.\mathtt{a} = se;\ \omega]\phi, \Delta}$$

$$\text{ifElseUnfold}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\mathtt{boolean\ b\ =}\ nse;\ \mathtt{if\ (b)\ \{p\}\ else\ \{q\}}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{if}\ (nse)\ \mathtt{\{p\}\ else\ \{q\}}\ \omega]\phi, \Delta}$$

Figure 2.10: Selected sequent calculus rules for decomposition of complex expressions.

3. $\Gamma \Rightarrow \{\mathtt{o_1.a} := \mathtt{v_1} \| \ldots\}[\mathtt{o_2.a\ =\ v_2;}\ \ \omega]\phi$.

   After executing $\mathtt{o_2.a} = \mathtt{v_2}$*, the alias is analyzed as follows: (i) if* $\mathtt{o_2} = \mathtt{null}$ *is true the program does not terminate; (ii) else, if* $\mathtt{o_2} = \mathtt{o_1}$ *holds, the value of* $\mathtt{o_1.a}$ *in the update is overriden and the new update is* $\{\mathtt{o_1.a} := \mathtt{v_2} \| \ldots \| \mathtt{o_2.a} := \mathtt{v_2}\}$*; (iii) else the new update is* $\{\mathtt{o_1.a} := \mathtt{v_1} \| \ldots \| \mathtt{o_2.a} := \mathtt{v_2}\}$*. Neither of (i)–(iii) might be provable and symbolic execution split into these three cases when encountering a possibly aliased object access.*

The result of symbolic execution for a SiJa program p following the sequent calculus rules is a *symbolic execution tree (SET)*, as illustrated in Figure 2.11. Note that, here we do not show the part that does not contain any SiJa program, e,g, the `init` branch obtained after applying the loopInvariant rule.

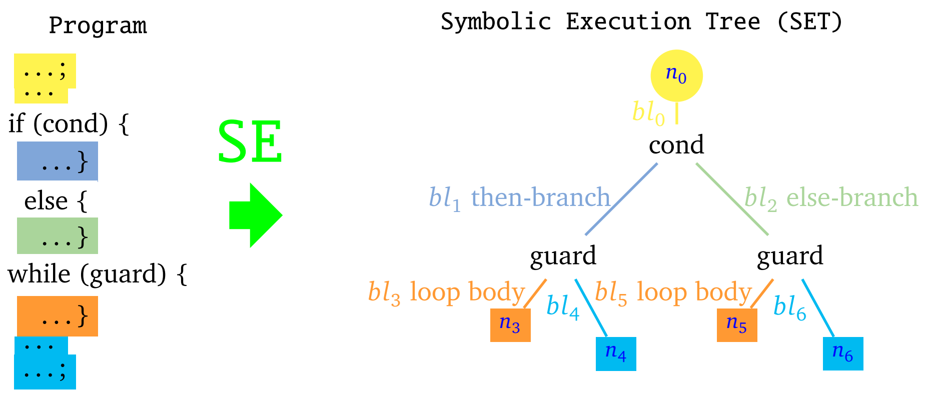

Figure 2.11: Symbolic execution tree with loop invariant applied.

Complete symbolic execution trees are finite acyclic trees whose root is labeled with $\Gamma \Longrightarrow [\mathtt{p}]\phi, \Delta$ and no leaf has a $[\cdot]$ modality. Without loss of generality, we can assume that each inner node i is annotated by a sequent $\Gamma_i \Longrightarrow \mathcal{U}_i[\mathtt{p_i}]\phi_i, \Delta_i$, where $\mathtt{p_i}$ is the program to be executed. Every child node is generated by rule application from its parent. A *branching node* represents a statement whose execution causes branching, e.g., conditional, object access, loops etc.

Definition 9 (Sequential block). *A* sequential block *(*SB*) is a maximal program fragment in an SET that is symbolically executed without branching.*

For instance, there are 7 sequential blocks bl_0,...,bl_6 in the SET in Figure 2.11.

Definition 10 (Child, descendent and sibling sequential block). *For sequential blocks bl_0 and bl_1:*

- *bl_1 is the* child *of bl_0, if bl_0 ends in a branching node n and bl_1 starts with n.*
- *bl_1 is the* descendant *of bl_0, if there exists sequential blocks bl^0,...,bl^m,$0 < m$ such that bl_0=bl^0, bl_1=bl^m and each bl^{i+1} is the child of bl^i for $0 \leq i < m$. Intuitively when $m = 1$, a child is also a descendant.*
- *bl_1 is the* sibling *of bl_0, if both bl_0 and bl_1 starts with the same branching node n .*

In the SET in Figure 2.11, bl_3 is the child of bl_1, the sibling of bl_4 and the descendant of bl_0.

Definition 11 (Generalized sequential block). *A* generalized sequential block *(*GSB*) is a sequential block together with all its* descendant *sequential blocks.*

It is a recursive definition, so a GSB always ends with leaf nodes. In the SET in Figure 2.11, we have GSB $\{bl_1, bl_3, bl_4\}$ and $\{bl_2, bl_5, bl_6\}$. However, $\{bl_0, bl_1, bl_2, bl_5, bl_6\}$ is not a GSB because bl_1 does not end with leaf nodes. Another remark is that a program is a GSB itself, which is $\{bl_0, bl_1, bl_2, bl_3, bl_4, bl_5, bl_6\}$ in this SET. For convenience, we refer to a GSB with the father sequential block. For instance, GSB $\{bl_1, bl_3, bl_4\}$ is denoted as GSB(bl_1).

3 Partial Evaluation

3.1 Partial Evaluation

The ideas behind partial evaluation go back in time even further than those behind symbolic execution: Kleene's well-known s_{mn} theorem from 1943 states that for each $m+n$-ary computable function $f(\vec{x},\vec{y})$ where $\vec{x}=x_1,\ldots,x_m$, $\vec{y}=y_1,\ldots,y_n$ there is an $m+1$-ary primitive recursive function s^m_n such that $\phi_{s^m_n(f,\vec{x})}=\lambda\vec{y}.f(\vec{x},\vec{y})$. Partial evaluation can be characterized as the research problem to prove Kleene's theorem under the following conditions:

1. $\phi_{s^m_n(f,\vec{x})}$ is supposed to run more efficiently than f for any given $\vec{x}$.
2. f is a program from a non-trivial programming language, not merely a recursive function.
3. The construction of $\phi_{s^m_n(f,\vec{x})}$ is efficient, i.e., its runtime should be comparable to compilation of f-programs.

In contrast to symbolic execution the result of a partial evaluator is not the value of output variables, but another program. The known input (named $\vec{x}$ above) is also called *static input* while the general part $\vec{y}$ is called *dynamic input*. The partial evaluator or *program specializer* is often named `mix`. Figure 3.1 taken from [JGS93] gives a schematic overview of partial evaluation.

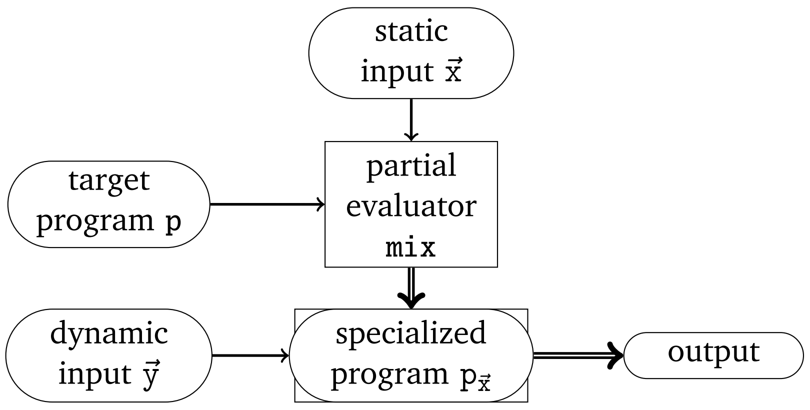

Figure 3.1: Partial evaluation schema.

The first efforts in partial evaluation date from the mid 1960s and were targeted towards Lisp. Due to the rise in popularity of functional and logic programming languages, the 1980s saw a large amount of research in partial evaluation of such languages. A seminal text on partial evaluation is the book by Jones et al. [JGS93].

There has been relatively little research on partial evaluation of Java. *JSpec* [SLC03] is the state-of-the-art program specializer for Java. It worked by cross-translation to C as an intermediate language. In fact, JSpec does not support full Java but a subset without concurrency, exception, reflection. JSpec uses an *offline* partial evaluation technique that depends on *binding time analysis*, which in general is not as precise as *online* partial evaluation. *Civet* [SC11] is a recent partial evaluator for Java based on *hybrid* partial evaluation, which performs offline-style specialization using an online approach without static binding time analysis. The programmer needs to explicitly identify to which parts of the programs partial evaluation should be applied. There is one other (commercial) Java partial evaluator called JPE[1], but its capabilities and underlying theory is not documented.

The application context of partial evaluation is rather different from that of symbolic execution: in practice, partial evaluation is not only employed to boost the efficiency of individual programs, but often used in meta-applications such as parser/compiler generation.

We illustrate the main principles of partial evaluation by a small SiJa program depicted in Figure 3.2 on the left. The program approximates the value of variable y to a given threshold with accuracy eps by repeatedly increasing or decreasing it as appropriate.

```
y = 80;
threshold = 100;

if (y > threshold) {
   decrease = true;
} else {
   decrease = false;
}

while (|y-threshold| > eps) {
  if (decrease) {
     y-1;
  } else {
     y+1;
  }
}
```

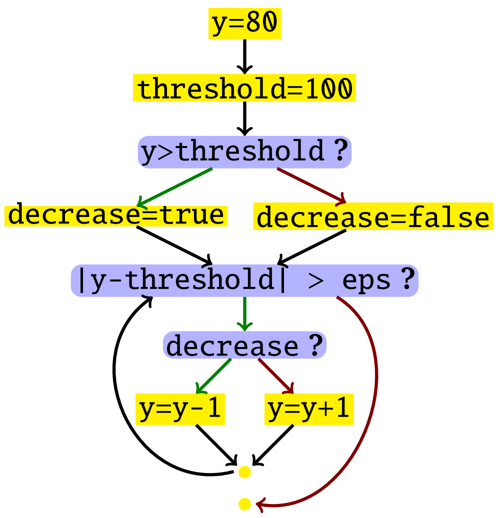

Figure 3.2: A simple control circuit SiJa program and its control flow graph.

We can imagine to walk a partial evaluator through the control flow graph (for the example on the right of Figure 3.2) while maintaining a table of concrete (i.e., constant) values for the program locations. In the example, that table is empty at first. After processing the two initial

[1] http://www.gradsoft.ua/products/jpe_eng.html

assignments it contains $\mathcal{U} = \{\mathtt{y} := 80 \| \mathtt{threshold} := 100\}$ (using the update notation of Section 2.3).

Whenever a new constant value becomes known, the partial evaluator attempts to propagate it throughout the current control flow graph (CFG). For the example, this *constant propagation* results in the CFG depicted in Figure 3.3 on the left. Note that the occurrences of y that are part of the loop have *not* been replaced. The reason is that y might be updated in the loop so that these latter occurrences of y cannot be considered to be static. Likewise, the value of decrease after the first conditional is not static either. The check whether the value of a given program location can be considered to be static with respect to a given node in the CFG is called *binding time analysis* (BTA) in partial evaluation.

Partial evaluation of our example proceeds now to the guard of the first conditional. This guard became a *constant expression* which can be evaluated to *false*. As a consequence, one can perform *dead code elimination* on the left branch of the conditional. The result is depicted in Figure 3.3 in the middle. Now the value of decrease is static and can be propagated into the loop (note that decrease is not changed inside the loop). After further dead code elimination, the final result of partial evaluation is the CFG on the right of Figure 3.3.

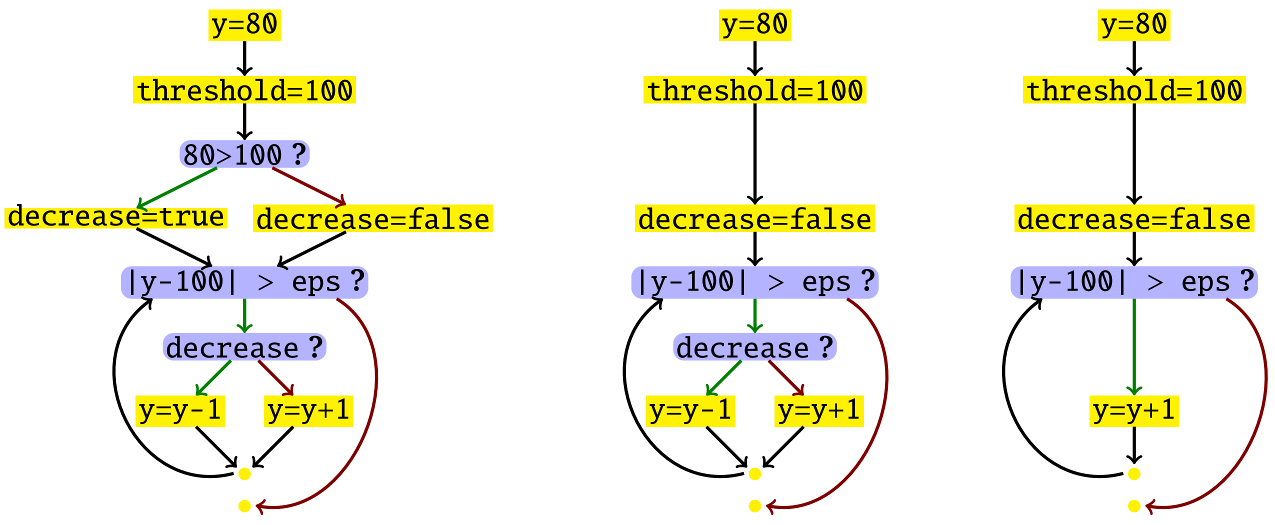

Figure 3.3: Partial evaluation example.

Partial evaluators necessarily approximate the target programming language semantics, because they are supposed to run fast and automatic. In the presence of such programming language features as exceptions, inheritance with complex localization rules (as in Java), and aliasing (e.g., references, array entries) BTA becomes very complex [SLC03].

3.2 Interleaving Symbolic Execution and Partial Evaluation

3.2.1 General Idea

Recall from Section 2.4 that a symbolic execution tree unwinds a program's control flow graph (CFG). As a consequence, identical code is (symbolically) executed in many branches, however, under differing path conditions and symbolic states. Merging back different nodes is usually not possible without approximation or abstraction [BHW09, Wei09].

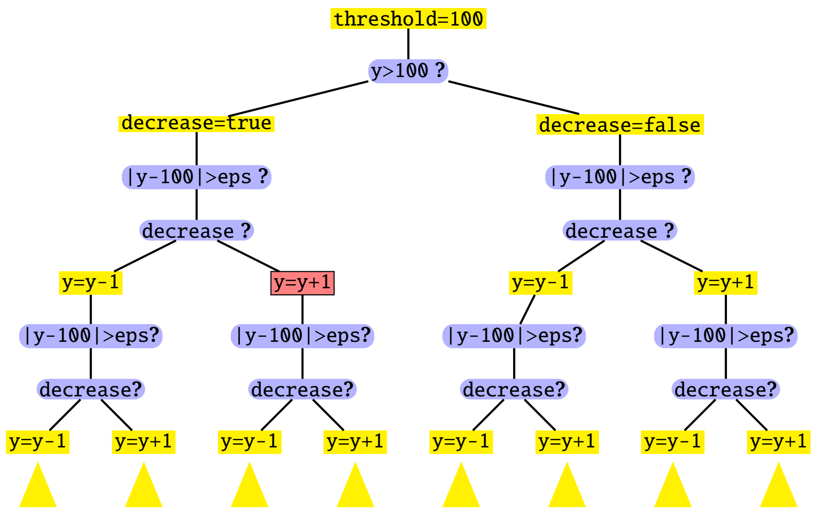

Figure 3.4: Symbolic execution tree of the control circuit program.

The hope with employing partial evaluation is that it is possible to factor out common parts of computations in different branches by evaluating them partially *before* symbolic execution takes place. The naïve approach, however, to *first* evaluate partially and *then* perform symbolic execution fails miserably. The reason is that for partial evaluation to work well the input space dimension of a program must be significantly reducible by identifying certain input variables to have static values.

Typical usage scenarios for symbolic execution like program verification are not of this kind. For example, in the program shown in Figure 3.2, it is unrealistic to classify the value of y as static. If we redo the example without the initial assignment $y = 80$ then partial evaluation can only perform one trivial constant propagation. The fact that input values for variables are not required to be static can even be considered to be one of the main advantages of symbolic execution and is the source of its generality: it is possible to cover all finite execution paths simultaneously and one can start execution at any given source code position without the need for initialization code.

The central observation that makes partial evaluation work in this context is that *during* symbolic execution static values are accumulated continuously as path conditions added to the current symbolic execution path. This suggests to perform partial evaluation *interleaved* with symbolic execution.

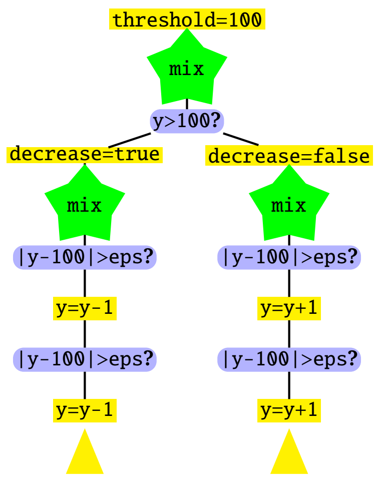

Figure 3.5: Symbolic execution with interleaved partial evaluation.

To be specific, we reconsider the example shown in Figure 3.2, but we remove the first statement assigning the static value 80 to y. As observed above, no noteworthy simplification of the program's CFG can be achieved by partial evaluation any longer. The structure of the CFG after partial evaluation remains exactly the same and only the occurrences of variable `threshold` are replaced by the constant value `100`. If we perform symbolic execution on this program, then the resulting execution tree spanned by two executions of the loop is shown in Figure 3.4. The first conditional divides the execution tree in two subtrees. The left subtree deals with the case that the value of y is too high and needs to be decreased. The right subtree with the complementary case.

All subsequent branches result from either the loop condition (omitted in Figure 3.4) or the conditional expression inside the loop body testing the value of `decrease`. As `decrease` is not modified within the loop, some of these branches are infeasible. For example the branch below the boxed occurrence of $\mathtt{y} = \mathtt{y} + 1$ (filled in red) is infeasible, because the value of `decrease` is `true` in that branch. Symbolic execution will not continue on these branches (at least for simple cases like that), but abandon them as infeasible by *proving* that the path condition is contradictory. Since the value of `decrease` is only tested *inside* the loop, however, the loop must still be first unwound and the proof that the current path condition is contradictory must be repeated. Partial evaluation can replace this potentially expensive proof search by *computation* which is drastically cheaper.

In the example, specializing the remaining program in each of the two subtrees after the first assignment to `decrease` eliminates the inner-loop conditional, see Figure 3.5 (the partial evaluation steps are labeled with `mix`). Hence, interleaving symbolic execution and partial evaluation promises to achieve a significant speed-up by removing redundancy from subsequent symbolic execution.

3.2.2 The Program Specialization Operator

We define a program specialization operator suitable for interleaving partial evaluation with symbolic execution in SiJa-DL. A soundness condition ensures that the operator can be safely integrated into the sequent calculus. This approach avoids to formalize the partial evaluator in SiJa-DL which would be tedious and inefficient.

Definition 12 (Program Specialization Operator)**.** *Let Σ be a signature and Σ' an extension of Σ as in Definition 7 containing countably infinite additional program variables and function symbols for any type and arity. Let σ be the embedding of Σ in Σ' ($\sigma(\Sigma) \subseteq \Sigma'$). The* program specialization operator

$$\downarrow_{\Sigma' \supseteq \Sigma} : \mathit{ProgramElement} \times \mathit{Updates}_{\Sigma'} \times \mathit{For}_{\Sigma'} \to \mathit{ProgramElement}$$

takes as arguments a SiJa-statement (-expression), an update and a SiJa-DL-formula and maps these to a SiJa-statement (-expression), where all arguments and the result are over Σ'.

The intention behind the above definition is that $\mathrm{p} \downarrow_{\Sigma' \supseteq \Sigma} (\mathcal{U}, \varphi)$ denotes a “simpler” but semantically equivalent version of p under the assumption that both are executed in a state coinciding with $\mathcal{U}$ and satisfying φ. The signature extension allows the specialization operator to introduce new temporary variables or function symbols.

A program specialization operator is *sound* if and only if Σ' is the signature extension of Σ and for all SiJa-DL-formulas $\psi \in For_{\Sigma}$, SiJa-DL-Kripke structures $\mathcal{K}_{\Sigma'}$, and states $s \in S_{\Sigma'}$

$$\mathcal{K}_{\Sigma'}, s \models \langle (\mathrm{p}) \downarrow_{\Sigma' \supseteq \Sigma} (\mathcal{U}, \varphi) \rangle \psi \Rightarrow \mathcal{K}_{\Sigma'}, s \models \mathcal{U}(\varphi \to \langle \mathrm{p} \rangle \psi) \ .$$

In words, the specialized program $\mathrm{p} \downarrow_{\Sigma' \supseteq \Sigma} (\mathcal{U}, \varphi)$ must be able to reach at least the same post-states as the original program p when started in a state coinciding with $\mathcal{U}$ in which (path condition) φ holds.

Interleaving partial evaluation and symbolic execution is achieved by introduction rules for the specialization operator. The simplest possibility is:

$$\text{introPE} \ \frac{\Gamma \Longrightarrow \mathcal{U}[(\mathrm{p}) \downarrow (\mathcal{U}, \mathit{true})]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathrm{p}]\phi, \Delta}$$

where $\downarrow$ is sound.

We instantiate the generic program specialization operator of Definition 12 with some possible actions. In each case we derive soundness conditions.

Specialization Operator Propagation.

The specialization operator needs to be propagated along the program as most of the different specialization operations work locally on single statements or expressions. During propagation of the operator, its knowledge base, the pair $(\mathcal{U}, \phi)$, needs to be updated by additional knowledge learned from executed statements or by erasing invalid knowledge about variables altered by the previous statement. Propagation of the specialization operator as well as updating the knowledge base is realized by the following rewrite rule

$$(\mathrm{p};\mathrm{q}) \downarrow (\mathcal{U}, \phi) \quad \rightsquigarrow \quad \mathrm{p} \downarrow (\mathcal{U}, \phi); \mathrm{q} \downarrow (\mathcal{U}', \phi')$$

This rule is unsound for arbitrarily chosen $\mathcal{U}'$, ϕ'. Soundness is ensured under a number of restrictions:

1. Let *mod* denote the set of all program locations possibly changed by p. Then we require that the SiJa-DL-formula "$\mathcal{U}$respectStrongModifies(p, mod)" is valid where the predicate respectStrongModifies abbreviates a formula that is valid if and only if p changes at most locations included in *mod*. "Strong" means that *mod* must contain even locations whose values are only changed temporarily. Such a formula is expressible in SiJa-DL, see [ERSW09] for details.

2. Let $\mathcal{V}_{mod}$ be the anonymizing update for *mod* (Definition 8). By fixing $\mathcal{U}' := \mathcal{U}\mathcal{V}_{mod}$ we ensure that the program state reached by executing p is covered by at least one interpretation and variable assignment over the extended signature.

3. ϕ' must be chosen in such a way that if $\mathcal{K}_\Sigma \models \mathcal{U}\langle \mathrm{p}\rangle\phi$ then there exists also an extended SiJa-DL-Kripke structure $\mathcal{K}_{\Sigma'}$ over an extended signature Σ' such that $\mathcal{K}_{\Sigma'} \models \mathcal{U}'\phi'$. This ensures that the post condition of p is correctly represented by ϕ'. One possible heuristic to obtain ϕ' consists of symbolic execution of p and applying the resulting update to ϕ. This yields a formula ϕ'' from which we obtain a candidate for ϕ' by "anonymizing" all occurrences of locations in it that occur in mod.

Constant propagation.

Constant propagation is one of the most basic operations in partial evaluation and often a prerequisite for more complex rewrite operations. Constant propagation entails that if the value of a variable v is known to have a constant value c within a certain program region (typically, until the variable is potentially reassigned) then usages of v can be replaced by c. The rewrite rule

$$(v)\downarrow(\mathcal{U},\varphi) \rightsquigarrow c$$

models the replacement operation. To ensure soundness the rather obvious condition $\mathcal{U}(\varphi \rightarrow v \doteq c)$ has to be proved where c is a rigid constant.

Dead-Code Elimination.

Constant propagation and constant expression evaluation result often in specializations where the guard of a conditional (or loop) becomes constant. In this case, unreachable code in the current state and path condition can be easily located and pruned.

A typical example for a specialization operation eliminating an infeasible symbolic execution branch is the rule

$$(\texttt{if (b) \{p\} else \{q\}}) \downarrow (\mathcal{U},\phi) \qquad \rightsquigarrow \qquad \texttt{p} \downarrow (\mathcal{U},\phi)$$

which eliminates the `else` branch of a conditional if the guard can be proved true. The soundness condition of the rule is straightforward and self-explaining: $\mathcal{U}(\phi \rightarrow \texttt{b} \doteq \texttt{TRUE})$.

Another case is

$$(\texttt{if (b) \{p\} else \{q\}}) \downarrow (\mathcal{U},\phi) \qquad \rightsquigarrow \qquad \texttt{q} \downarrow (\mathcal{U},\phi)$$

where the soundness condition is: $\mathcal{U}(\phi \rightarrow \texttt{b} \doteq \texttt{FALSE})$.

Safe Field Access.

Partial evaluation can be used to mark expressions as safe that contain field accesses or casts that may otherwise cause non-termination. We use the notation @(`e`) to mark an expression `e` as safe, for example, if we can ensure that $\texttt{o} \neq \texttt{null}$, then we can derive the annotation @(`o.a`) for any field `a` in the type of `o`. The advantage of safe annotations is that symbolic execution can assume that safe expressions terminate normally and needs not to spawn side proofs that ensure it. The rewrite rule for safe field accesses is

$$\texttt{o.a} \downarrow (\mathcal{U},\phi) \quad \rightsquigarrow \quad @(\texttt{o.a}) \downarrow (\mathcal{U},\phi) \ .$$

Its soundness condition is $\mathcal{U}(\phi \rightarrow \neg(\texttt{o} \doteq \texttt{null}))$.

Type Inference.

For deep type hierarchies dynamic dispatch of method invocations may cause serious performance issues in symbolic execution, because a long cascade of method calls is created by the method invocation rule (Section 2.4). To reduce the number of implementation candidates we use information from preceding symbolic execution to narrow the static type of the callee as far as possible and to (safely) cast the reference to that type. The method invocation rule can then determine the implementation candidates more precisely:

$$\begin{aligned}&\mathtt{res} = \mathtt{o}.m(\mathtt{a}_1,\ldots,\mathtt{a}_n);\downarrow(\mathcal{U},\phi) \quad \rightsquigarrow\\ &\quad \mathtt{res} = @((Cl)\mathtt{o}\downarrow(\mathcal{U},\phi)).m(\mathtt{a}_1\downarrow(\mathcal{U},\phi),\ldots,\mathtt{a}_n\downarrow(\mathcal{U},\phi));\end{aligned}$$

The accompanying soundness condition $\mathcal{U}(\phi \rightarrow \exists\, Cl\ x;(\mathtt{o} \doteq x))$ ensures that the type of `o` is compatible with Cl in any state specified by $\mathcal{U}$, ϕ.

3.3 Example

As an application of interleaving symbolic execution and partial evaluation, consider the verification of a GUI library. It includes standard visual elements such as `Window`, `Icon`, `Menu` and `Pointer`. An element has different implementations for different platforms or operating systems. Consider the following program snippet involving dynamic method dispatch:

```
framework.ui.Button button = radiobuttonX11;
button.paint();
```

The element `Button` is implemented in one way for Max OS X, while it is implemented in a different way for the X Window System. The method `paint()` is defined in `Button` which is extended by `CheckBox`, `Component`, and `Dialog`. Altogether, `paint()` is implemented in 16 different classes including `ButtonX11`, `ButtonMPC`, `RadioButtonX11`, `MenuItemX11`, etc. The type hierarchy is shown in Figure 3.6. In the code above `button` is assigned an object with type `RadioButtonX11` which implements `paint()`. As a consequence, it should always terminate and the SiJa-DL-formula $\langle\mathtt{gui}\rangle true$ should be provable where `gui` abbreviates the code above.

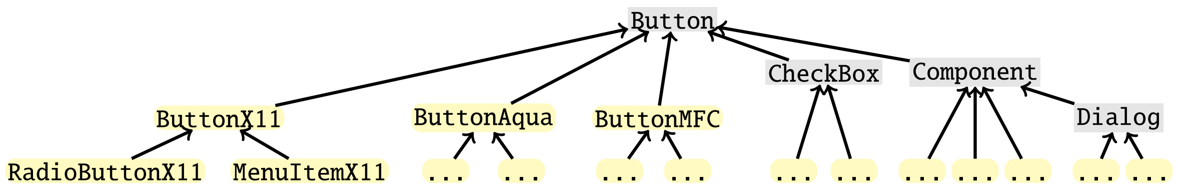

Figure 3.6: Type hierarchy for the GUI example.

First, we employ symbolic execution alone to do the proof. During this process, `button.paint()` is unfolded into 16 different cases by the method invocation rule (Section 2.4), each corresponding to a possible implementation of `button` in one of the subclasses of `Button`. The proof is constructed automatically in KeY 1.6 with 161 nodes and 10 branches in the proof tree.

In a second experiment, we interleave symbolic execution and partial evaluation to prove the same claim. The partial evaluator propagates with the help of the TypeInference rule in the previous section the information that the run-time type of `button` is `RadioButtonX11` and the only possible implementation of `button.paint()` is `RadioButtonX11.paint()`. All other possible implementations are pruned. Only 24 nodes and 2 branches occur in the proof tree when running KeY integrated with a partial evaluator.

3.4 Evaluation

We implemented a simple partial evaluator for SiJa and interleaved it with symbolic execution in the KeY system as described above. We formally verified a number of Java programs with KeY 1.6 with and without partial evaluation.

Table 3.1 shows the experimental results for a number of small Java programs. The column "Program" shows the name of the program we prove, the column "Strategy" shows the strategy we choose to perform the proof where "SE" means symbolic execution and "SE+PE" means interleaving symbolic execution and partial evaluation; the column "#Nodes" shows the total number of nodes in the proof; the column "#Branches" shows the total number of branches in the proof. The results show that interleaving symbolic execution with partial evaluation significantly speeds up the proof for `complexEval`, `constantPropagation`, `dynamicDispatch`, `safeAccess`, and `safeTypeCast` which can all be considered to be amenable to partial evaluation.

Program	Strategy	#Nodes	#Branches
`complexEval`	SE	261	15
	SE+PE	158	3
`constantPropagation`	SE	65	1
	SE+PE	56	1
`dynamicDispatch`	SE	161	10
	SE+PE	24	2
`methodCall`	SE	113	4
	SE+PE	108	3
`safeAccess`	SE	28	4
	SE+PE	24	3
`safeTypeCast`	SE	73	5
	SE+PE	45	3

Table 3.1: Symbolic execution and partial evaluation for small Java programs.

Table 3.2 shows the experimental results of verifying a larger and more realistic Java e-banking application used in [BHS06, Ch. 10]. The column “Proof Obligation” shows which property we prove; the remaining columns are as in Table 3.1. The results show that symbolic

Proof Obligation	Strategy	#Nodes	#Branches
`ATM.insertCard (EnsuresPost)`	SE	949	20
	SE+PE	805	13
`ATM.insertCard (PreservesInv)`	SE	2648	89
	SE+PE	2501	79
`ATM.enterPIN (EnsuresPost)`	SE	661	7
	SE+PE	654	8
`ATM.enterPIN (PreservesInv)`	SE	1524	45
	SE+PE	1501	44
`ATM.confiscateCard (EnsuresPost)`	SE	260	2
	SE+PE	255	2
`ATM.confiscateCard (PreservesInv)`	SE	739	19
	SE+PE	695	19
`ATM.accountBalance (EnsuresPost)`	SE	1337	35
	SE+PE	1271	29
`ATM.accountBalance (PreservesInv)`	SE	2233	57
	SE+PE	2223	59
`Account.checkAndWithdraw (EnsuresPost)`	SE	16174	136
	SE+PE	17023	135
`Account.checkAndWithdraw (PreservesInv)`	SE	14076	89
	SE+PE	10478	78

Table 3.2: Symbolic execution and partial evaluation for an e-banking application.

execution interleaved with partial evaluation can speed up verification proofs even for larger applications. As is to be expected, depending on the structure of the program the benefit varies. It is noteworthy that none of the programs and proof obligations used in the present chapter have been changed in order to make them more amenable to partial evaluation. In no case we have to pay a significant performance penalty which seems to indicate that partial evaluation is a generally useful technology for symbolic execution and should generally be applied.

The case study in Section 3.3 suggests that it could pay off to take partial evaluation into account when designing programs, specifications, and proof obligations.

4 Program Transformation

The structure of a symbolic execution tree makes it possible to synthesize a program by bottom-up traversal. The idea is to apply the sequent calculus rules reversely and generate the program step-by-step. This requires to extend the sequent calculus rules with means for program synthesis. Obviously, the synthesized program should behave exactly as the original one, at least for the *observable locations*. To this end we introduce the notion of *weak bisimulation* for SiJa programs and show its soundness for program transformation. Although this chapter assumes that the target language is the same as the source, the concept can be easily generalized to pairs of different languages, e.g., Java (SiJa) source code and bytecode, which is discussed in Chapter 6.

4.1 Weak Bisimulation Relation of Programs

Definition 13 (Location sets, observation equivalence)**.** *A* location set *is a set containing program variables* $\mathtt{x}$ *and attribute expressions* $o.\mathtt{a}$ *with* $\mathtt{a} \in \mathrm{Attr}$ *and* o *being a term of the appropriate sort. Let loc be the set of all program locations, given two states* s_1, s_2 *and a location set obs,* $obs \subseteq loc$*. A relation* $\approx: loc \times S \times S$ *is an* observation equivalence *if and only if for all* $ol \in obs$*,* $val_{D,s_1,\beta}(ol) = val_{D,s_2,\beta}(ol)$ *holds. It is written as* $s_1 \approx_{obs} s_2$*. We call obs* observable locations.

The semantics of a SiJa program p (Figure 2.6) is a state transformation. Executing p from a start state s results in a set of end states S', where S' is a singleton $\{s'\}$ if p terminates, or $\emptyset$ otherwise. We identify a singleton with its only member, so in case of termination, $val_{D,s}(\mathtt{p})$ is evaluated to s' instead of $\{s'\}$.

A *transition relation* $\longrightarrow: \Pi \times S \times S$ relates two states s, s' by a program p if and only if p starts in state s and terminates in state s', written $s \xrightarrow{\mathtt{p}} s'$. We have: $s \xrightarrow{\mathtt{p}} s'$, where $s' = val_{D,s}(\mathtt{p})$. If p does not terminate, we write $s \xrightarrow{\mathtt{p}}$.

Since a complex statement can be decomposed into a set of simple statements, which is done during symbolic execution, we can assume that a program p consists of simple statements. Execution of p leads to a sequence of state transitions: $s \xrightarrow{\mathtt{p}} s' \equiv s_0 \xrightarrow{\mathtt{sSt_0}} s_1 \xrightarrow{\mathtt{sSt_1}} \ldots \xrightarrow{\mathtt{sSt_{n-1}}} s_n \xrightarrow{\mathtt{sSt_n}} s_{n+1}$, where $s = s_0$, $s' = s_{n+1}$, s_i a *program state* and $\mathtt{sSt_i}$ a simple statement $(0 \leq i \leq n)$. A program state has the same semantics as the *state* defined in a Kripke structure, so we use both notations without distinction.

Some simple statements reassign values (write) to a location *ol* in the observable locations that affects the evaluation of *ol* in the final state. We distinguish these simple statements from those that do not affect the observable locations.

Definition 14 (Observable and internal statement/transition)**.** *Consider states* s, s'*, a simple statement* $\mathtt{sSt}$*, a transition relation* $\longrightarrow$*, where* $s \xrightarrow{\mathtt{sSt}} s'$*, and the observable locations obs; we* *call* `sSt` *an* observable statement *and* $\longrightarrow$ *an* observable transition*, if and only if there exists* $ol \in obs$*, and* $val_{D,s',\beta}(ol) \neq val_{D,s,\beta}(ol)$*. We write* $\xrightarrow{\texttt{sSt}}_{obs}$*. Otherwise,* `sSt` *is called an* internal statement *and* $\longrightarrow$ *an* internal transition*, written* $\longrightarrow_{int}$*.*

In this definition, observable/internal transitions are *minimal* transitions that relate two states with a simple statement. We indicate the simple statement `sSt` in the notion of the observable transition $\xrightarrow{\texttt{sSt}}_{obs}$, since `sSt` reflects the changes of the observable locations. In contrast, an internal statement does not appear in the notion of the internal transition.

Example 5. *Given the set of observable locations obs=*{`x`, `y`}*, the simple statement "*`x = 1 + z;`*" is observable, because* `x`*'s value is reassigned. The statement "*`z = x + y;`*" is internal, since the evaluation of* `x`*,* `y` *are not changed, even though the value of each variable is read by* `z`*.*

Remark. *An observable transition is defined by observing the changes of obs in the* final *state after the transition. For a program that consists of many statements, the observable locations for the final state may differ from that for some internal state. Assume an observable transition* $s \xrightarrow{\texttt{sSt}}_{obs} s'$ *changes the evaluation of some location* $ol \in obs$ *in state* s'*. The set of observable locations* obs_1 *in state* s *should also contain the locations* ol_1 *that is* read *by* ol*, because the change to* ol_1 *can lead to a change of* ol *in the final state* s'*.*

Example 6. *Consider the set of observable locations obs=*{`x`, `y`} *and program fragment "*`z = x + y; x = 1 + z;`*". The statement* `z = x + y;` *becomes observable because the value of* `z` *is changed and it will be used later in the observable statement* `x = 1 + z;`*. The observable location set* obs_1 *should contain* `z` *after the execution of* `z = x + y;` *.*

Definition 15 (Weak transition)**.** *Given a set of observable locations obs, the transition relation* $\Longrightarrow_{int}$ *is the reflexive and transitive closure of* $\longrightarrow_{int}$*:* $s \Longrightarrow_{int} s'$ *holds if and only if for states* $s_0,\ldots,s_n$*,* $n\geq 0$*, we have* $s = s_0$*,* $s' = s_n$ *and* $s_0 \longrightarrow_{int} s_1 \longrightarrow_{int} \cdots \longrightarrow_{int} s_n$*. In the case of* $n = 0$*,* $s \Longrightarrow_{int} s$ *holds. The transition relation* $\overset{\texttt{sSt}}{\Longrightarrow}_{obs}$ *is the composition of the relations* $\Longrightarrow_{int}$*,* $\xrightarrow{\texttt{sSt}}_{obs}$ *and* $\Longrightarrow_{int}$*:* $s \overset{\texttt{sSt}}{\Longrightarrow}_{obs} s'$ *holds if and only if there are states* s_1 *and* s_2 *such that* $s \Longrightarrow_{int} s_1 \xrightarrow{\texttt{sSt}}_{obs} s_2 \Longrightarrow_{int} s'$*. The* weak transition $\overset{\widehat{\texttt{sSt}}}{\Longrightarrow}_{obs}$ *represents either* $\overset{\texttt{sSt}}{\Longrightarrow}_{obs}$*, if* `sSt` *observable or* $\Longrightarrow_{int}$ *otherwise.*

In other words, a weak transition is a sequence of minimal transitions that contains at most one observable transition.

Definition 16 (Weak bisimulation for states)**.** *Given two programs* $\texttt{p}_1, \texttt{p}_2$ *and observable locations obs, obs', let* $\texttt{sSt}_1$ *be a simple statement and* s_1, s_1' *two program states of* $\texttt{p}_1$*, and* $\texttt{sSt}_2$ *is a simple statement and* s_2, s_2' *are two program states of* $\texttt{p}_2$*. A relation* $\approx$ *is a* weak bisimulation *for states if and only if* $s_1 \approx_{obs} s_2$ *implies:*

- *if* $s_1 \overset{\widehat{\texttt{sSt}_1}}{\Longrightarrow}_{obs'} s_1'$*, then* $s_2 \overset{\widehat{\texttt{sSt}_2}}{\Longrightarrow}_{obs'} s_2'$ *and* $s_1' \approx_{obs'} s_2'$

- *if* $s_2 \overset{\widehat{\mathtt{sSt_2}}}{\Longrightarrow}_{obs'} s_2'$, *then* $s_1 \overset{\widehat{\mathtt{sSt_1}}}{\Longrightarrow}_{obs'} s_1'$ *and* $s_2' \approx_{obs'} s_1'$

where $val_{D,s_1}(\mathtt{sSt_1}) \approx_{obs'} val_{D,s_2}(\mathtt{sSt_2})$.

Definition 17 (Weak bisimulation for programs). *Let* $\mathtt{p_1}, \mathtt{p_2}$ *be two programs, obs and obs′ are observable locations, and* $\approx$ *is a weak bisimulation relation for states.* $\approx$ *is a* weak bisimulation *for programs, written* $\mathtt{p_1} \approx_{obs} \mathtt{p_2}$*, if for the sequence of state transitions:*

$s_1 \xrightarrow{\mathtt{p_1}} s_1' \equiv s_1^0 \xrightarrow{\mathtt{sSt_1^0}} s_1^1 \xrightarrow{\mathtt{sSt_1^1}} \ldots \xrightarrow{\mathtt{sSt_1^{n-1}}} s_1^n \xrightarrow{\mathtt{sSt_1^n}} s_1^{n+1}$, *with* $s_1 = s_1^0$, $s_1' = s_1^{n+1}$,

$s_2 \xrightarrow{\mathtt{p_2}} s_2' \equiv s_2^0 \xrightarrow{\mathtt{sSt_2^0}} s_2^1 \xrightarrow{\mathtt{sSt_2^1}} \ldots \xrightarrow{\mathtt{sSt_2^{m-1}}} s_1^m \xrightarrow{\mathtt{sSt_2^m}} s_2^{m+1}$, *with* $s_2 = s_2^0$, $s_2' = s_2^{m+1}$,

we have (i) $s_2' \approx_{obs} s_1'$*; (ii) for each state* s_1^i *there exists a state* s_2^j *such that* $s_1^i \approx_{obs'} s_2^j$ *for some obs′; (iii) for each state* s_2^j *there exists a state* s_1^i *such that* $s_2^j \approx_{obs'} s_1^i$ *for some obs′, where* $0 \le i \le n$ *and* $0 \le j \le m$.

The weak bisimulation relation for programs defined above requires a weak transition that relates two states with at most one observable transition. This definition reflects the *structural* properties of a program and can be characterized as a *small-step semantics* [Plo04]. It directly implies the lemma below that relates the weak bisimulation relation of programs to a *big-step semantics* [Kah87].

Lemma 2. *Let* $\mathtt{p}, \mathtt{q}$ *be programs and obs the set of observable locations. It holds* $\mathtt{p} \approx_{obs} \mathtt{q}$ *if and only if for any first-order structure D and state s,* $val_{D,s}(\mathtt{p}) \approx_{obs} val_{D,s}(\mathtt{q})$ *holds.*

4.2 The Weak Bisimulation Modality and Sequent Calculus Rules

We introduce a weak bisimulation modality which allows us to relate two programs that behave indistinguishably on the observable locations.

Definition 18 (Weak bisimulation modality—syntax). *The* bisimulation modality $[\ \mathtt{p} \between \mathtt{q}\ ]@(obs, use)$ *is a modal operator providing compartments for programs* $\mathtt{p}$, $\mathtt{q}$ *and location sets obs and use. We extend our definition of formulas: Let* ϕ *be a* **SiJa**-*DL formula and* $\mathtt{p}, \mathtt{q}$ *two* **SiJa** *programs and obs, use two location sets such that* $pv(\phi) \subseteq obs$ *where* $pv(\phi)$ *is the set of all program variables occurring in* ϕ*, then* $[\ \mathtt{p} \between \mathtt{q}\ ]@(obs, use)\phi$ *is also a* **SiJa**-*DL formula.*

The intuition behind the location set $usedVar(s, \mathtt{p}, obs)$ defined below is to capture precisely those locations whose value influences the final value of an observable location $l \in obs$ (or the evaluation of a formula ϕ) after executing a program $\mathtt{p}$. We approximate the set later by the set of all program variables in a program that are used before being redefined (i.e., assigned a new value).

Definition 19 (Used program variable). *A variable $v \in \mathrm{PV}$ is called* used by a program $\mathtt{p}$ *with respect to a location set obs, if there exists an $l \in obs$ such that*

$$D, s \models \forall v_l . \exists v_0 . ((\langle \mathtt{p} \rangle l = v_l) \rightarrow (\{v := v_0\} \langle \mathtt{p} \rangle l \neq v_l))$$

The set $usedVar(s, \mathtt{p}, obs)$ is defined as the smallest set containing all used program variables of $\mathtt{p}$ *with respect to obs.*

The formula defining a used variable v of a program $\mathtt{p}$ encodes that there is an interference with a location contained in *obs*. In Example 6, $\mathtt{z}$ is a used variable. We formalize the semantics of the weak bisimulation modality:

Definition 20 (Weak bisimulation modality—semantics). *With* $\mathtt{p}, \mathtt{q}$ *SiJa-programs, D, s, β, and obs, use as before, let $val_{D,s,\beta}([\ \mathtt{p} \between \mathtt{q}\ ]@(obs, use)\phi) = tt$ if and only if*

1. $val_{D,s,\beta}([\mathtt{p}]\phi) = tt$
2. $use \supseteq usedVar(s, \mathtt{q}, obs)$
3. *for all $s' \approx_{use} s$ we have $val_{D,s}(\mathtt{p}) \approx_{obs} val_{D,s'}(\mathtt{q})$*

Lemma 3. *Let obs be the set of all locations observable by ϕ and let $\mathtt{p}, \mathtt{q}$ be programs. If $\mathtt{p} \approx_{obs} \mathtt{q}$ then $val_{D,s,\beta}([\mathtt{p}]\phi) \leftrightarrow val_{D,s,\beta}([\mathtt{q}]\phi)$ holds for all D, s, β.*

Proof. Direct consequence of Definition 20 and Lemma 2. □

An extended sequent for the bisimulation modality is:

$$\Gamma \Longrightarrow \mathcal{U}[\ \mathtt{p} \between \mathtt{q}\ ]@(obs, use)\phi, \Delta$$

The following lemma gives an explicit meaning of used variable set *use*.

Lemma 4. *An extended sequent $\Gamma \Longrightarrow \mathcal{U}[\ \mathtt{p} \between \mathtt{q}\ ]@(obs, use)\phi, \Delta$ within a sequential block bl (see Definition 9) represents a certain state s_1, where* $\mathtt{P}$ *is the original program of bl,* $\mathtt{p}$ *is the original program to be executed in bl at state s_1, and* $\mathtt{p}'$ *is the original program already been executed in bl; while* $\mathtt{Q}$ *is the program to be generated of bl,* $\mathtt{q}$ *is the already generated program in bl, and* $\mathtt{q}'$ *is the remaining program to be generated in bl. The location set use are the* dynamic observable locations *that the following relations hold:*

(i) $\mathtt{p} \approx_{obs} \mathtt{q}$

(ii) $\mathtt{P} \approx_{obs} \mathtt{Q}$

(iii) $\mathtt{p}' \approx_{use} \mathtt{q}'$

Proof. The structure of this sequential block bl is illustrated in Figure 4.1.

(i) $\mathtt{p} \approx_{obs} \mathtt{q}$

It is the direct consequence of Definition 20.

(ii) $\mathtt{P} \approx_{obs} \mathtt{Q}$

Consider the initial state s_0 of this sequential block, where $use = use_0$, p=P and q=Q in the sequent, we have $s'_0 \approx_{use_0} s_0$, according to Definition 20 and Lemma 2, $\mathtt{P} \approx_{obs} \mathtt{Q}$ holds.

(iii) $\mathtt{p}' \approx_{use} \mathtt{q}'$

Consider the truncated sequential block bl_2 starting from the current state s_1 and ending with the final state s_2 According to Definition 19, if there is no program in bl_2, then we have $obs = use$. Now consider the truncated sequential block bl_1 starting from the initial state s_0 and ending with the current state s_1. We have $use = use_0$, p=p′, q=q′ and $obs = use$ in the sequent, according to Definition 20 and Lemma 2, $\mathtt{p}' \approx_{use} \mathtt{q}'$ holds. □

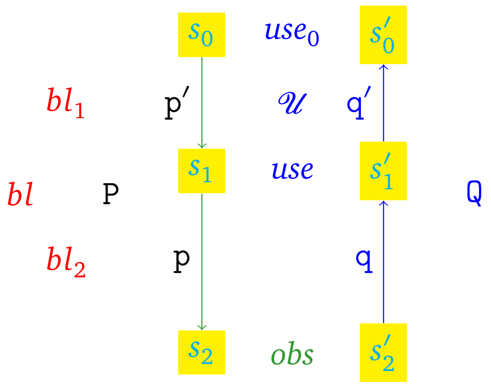

Figure 4.1: Program in a sequential block.

The sequent calculus rules for the bisimulation modality are of the following form:

$$\text{ruleName}\ \frac{\begin{array}{c}\Gamma_1 \Longrightarrow \mathcal{U}_1[\, \mathtt{p}_1 \between \mathtt{q}_1 \,]@(obs_1, use_1)\phi_1, \Delta_1 \\ \ldots \\ \Gamma_n \Longrightarrow \mathcal{U}_n[\, \mathtt{p}_n \between \mathtt{q}_n \,]@(obs_n, use_n)\phi_n, \Delta_n\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\, \mathtt{p} \between \mathtt{q} \,]@(obs, use)\phi, \Delta}$$

Figure 4.2 shows some extended sequent calculus rules, where $\overline{\omega}$ denotes the generated program that is weakly bisimilar to ω, and _ is a place holder for *empty*.

Unlike standard sequent calculus rules that are executed from root to leaves, sequent rule application for the bisimulation modality consists of two phases:

Phase 1. Symbolic execution of source program p as usual. In addition, the observable location sets obs_i are propagated, since they contain the locations observable by $\mathtt{p}_i$ and ϕ_i that will be used in the second phase. Typically, obs contains the return variables of a method and the locations used in the continuation of the program, e.g., program variables used after a loop

$$\text{emptyBox}\ \frac{\Gamma \Longrightarrow \mathcal{U}@(obs,\_)\phi,\Delta}{\Gamma \Longrightarrow \mathcal{U}[\ \_ \between \_\ ]@(obs,obs)\phi,\Delta}$$

$$\text{assignment}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{l} := \mathtt{r}\}[\ \omega \between \overline{\omega}\ ]@(obs,use)\phi,\Delta}{\left(\begin{array}{ll} \Gamma \Longrightarrow \mathcal{U}[\ \mathtt{l}=\mathtt{r};\omega \between \mathtt{l}=\mathtt{r};\overline{\omega}\ ]@(obs,use-\{\mathtt{l}\}\cup\{\mathtt{r}\})\phi,\Delta & \text{if } \mathtt{l}\in use \\ \Gamma \Longrightarrow \mathcal{U}[\ \mathtt{l}=\mathtt{r};\omega \between \overline{\omega}\ ]@(obs,use)\phi,\Delta & \text{otherwise} \end{array}\right)}$$

$$\text{ifElse}\ \frac{\begin{array}{l} \Gamma,\mathcal{U}\mathtt{b} \Longrightarrow \mathcal{U}[\ \mathtt{p};\omega \between \overline{\mathtt{p};\omega}\ ]@(obs,use_{\mathtt{p};\omega})\phi,\Delta \\ \Gamma,\mathcal{U}\neg\mathtt{b} \Longrightarrow \mathcal{U}[\ \mathtt{q};\omega \between \overline{\mathtt{q};\omega}\ ]@(obs,use_{\mathtt{q};\omega})\phi,\Delta \end{array}}{\begin{array}{l} \Gamma \Longrightarrow \mathcal{U}[\ \mathtt{if\ (b)\ \{p\}\ else\ \{q\}}\ \omega \between \\ \qquad \mathtt{if\ (b)\ \{}\overline{\mathtt{p};\omega}\mathtt{\}\ else\ \{}\overline{\mathtt{q};\omega}\mathtt{\}}\ ]@(obs,use_{\mathtt{p};\omega}\cup use_{\mathtt{q};\omega}\cup\{\mathtt{b}\})\phi,\Delta \end{array}}$$

(with b boolean variable)

$$\text{loopUnwind}\ \frac{\begin{array}{l} \Gamma \Longrightarrow \mathcal{U}[\ \mathtt{if\ (b)\ \{p;while\ (b)\ \{p\}\}}\ \omega \between \\ \qquad \overline{\mathtt{if\ (b)\ \{p;while\ (b)\ \{p\}\}}}\ \omega\ ]@(obs,use)\phi,\Delta \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\ \mathtt{while(b)\ \{p\}}\ \omega \between \overline{\mathtt{if\ (b)\ \{p;while(b)\ \{p\}\}}}\ \omega\ ]@(obs,use)\phi,\Delta}$$

$$\text{loopInvariant}\ \frac{\begin{array}{l} \Gamma \Longrightarrow \mathcal{U}inv,\Delta \\ \Gamma,\mathcal{U}\mathcal{V}_{mod}(\mathtt{b}\wedge inv) \Longrightarrow \mathcal{U}\mathcal{V}_{mod} \\ \qquad\qquad [\ \mathtt{p} \between \overline{\mathtt{p}}\ ]@(use_1\cup\{\mathtt{b}\},use_2)inv,\Delta \\ \Gamma,\mathcal{U}\mathcal{V}_{mod}(\neg\mathtt{b}\wedge inv) \Longrightarrow \mathcal{U}\mathcal{V}_{mod}[\ \omega \between \overline{\omega}\ ]@(obs,use_1)\phi,\Delta \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\ \mathtt{while(b)\{p\}}\ \omega \between \mathtt{while(b)\{}\overline{\mathtt{p}}\mathtt{\}}\ \overline{\omega}\ ]@(obs,use_1\cup use_2\cup\{b\})\phi,\Delta}$$

$$\begin{array}{l} \text{methodContract}_{C=(pre,post,mod)} \\ \dfrac{\begin{array}{l} \Gamma \Longrightarrow \mathcal{U}\{\mathtt{prm}_1 := \mathtt{v}_1\|\ldots\|\mathtt{prm}_n := \mathtt{v}_\mathtt{n}\}pre,\Delta \\ \Gamma \Longrightarrow \mathcal{U}\{\mathtt{prm}_1 := \mathtt{v}_1\|\ldots\|\mathtt{prm}_n := \mathtt{v}_\mathtt{n}\}\mathcal{V}_{mod} \\ \qquad\qquad (post \rightarrow \{\mathtt{r} := \mathtt{res}\}[\ \omega \between \overline{\omega}\ ]@(obs,use)\phi),\Delta \end{array}}{\begin{array}{l} \Gamma \Longrightarrow \mathcal{U}[\ \mathtt{r}=\mathtt{m}(\mathtt{v}_1,\ldots,\mathtt{v}_\mathtt{n});\ \omega \between \mathtt{r}=\mathtt{m}(\mathtt{v}_1,\ldots,\mathtt{v}_\mathtt{n});\overline{\omega}\ ]@(obs,use)\phi,\Delta \\ \text{(Contract } C \text{ is correct)} \end{array}} \end{array}$$

Figure 4.2: A collection of sequent calculus rules for program transformation.

must be reflected in the observable locations of the loop body. The result of this phase is a symbolic execution tree as illustrated in Figure 2.11.

Phase 2. We synthesize the target program $\mathtt{q}$ and used variable set *use* from $\mathtt{q}_i$ and $use_{\mathtt{i}}$ by applying the rules in a leave-to-root manner. One starts with a leaf node and generates the program within its sequential block first, e.g., bl_3, bl_4, bl_5, bl_6 in Figure 2.11. These are combined by rules corresponding to statements that contain a sequential block, such as loopInvariant (containing bl_3 and bl_4). One continues with the generalized sequential block containing the compound statements, e.g., $GSB(bl_2)$, and so on, until the root is reached. Note that the order of processing the sequential blocks matters, for instance, the program for the sequential block bl_4 must be generated before that for bl_3, because the observable locations in node n_3 depend on the used variable set of bl_4 according to the loopInvariant rule.

We explain some of the rules in details.

$$\text{emptyBox}\ \frac{\Gamma \Longrightarrow \mathcal{U} @(obs, \_)\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\ \_ \between \_\ ]@(obs, obs)\phi, \Delta}$$

The emptyBox rule is the starting point of program transformation in each sequential block. The location set *use* is set to *obs*, which is the direct result of Lemma 4.

$$\text{assignment}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{l} := \mathtt{r}\}[\ \omega \between \overline{\omega}\ ]@(obs, use)\phi, \Delta}{\begin{pmatrix} \Gamma \Longrightarrow \mathcal{U}[\ \mathtt{l} = \mathtt{r}; \omega \between \mathtt{l} = \mathtt{r}; \overline{\omega}\ ]@(obs, use - \{\mathtt{l}\} \cup \{\mathtt{r}\})\phi, \Delta & \text{if } \mathtt{l} \in use \\ \Gamma \Longrightarrow \mathcal{U}[\ \mathtt{l} = \mathtt{r}; \omega \between \overline{\omega}\ ]@(obs, use)\phi, \Delta & \text{otherwise} \end{pmatrix}}$$

In the assignment rule, the *use* set contains all program variables on which a read access might occur in the remaining program before being overwritten. In the first case, when the left side $\mathtt{l}$ of the assignment is among those variables, we have to update the *use* set by removing the newly assigned program variable $\mathtt{l}$ and adding the variable $\mathtt{r}$ which is read by the assignment. The second case makes use of the knowledge that the value of $\mathtt{l}$ is not accessed in the remaining program and skips the generation of the assignment.

$$\text{ifElse}\ \frac{\begin{array}{c}\Gamma, \mathcal{U}\mathtt{b} \Longrightarrow \mathcal{U}[\ \mathtt{p}; \omega \between \overline{\mathtt{p}; \omega}\ ]@(obs, use_{\mathtt{p};\omega})\phi, \Delta \\ \Gamma, \mathcal{U}\neg\mathtt{b} \Longrightarrow \mathcal{U}[\ \mathtt{q}; \omega \between \overline{\mathtt{q}; \omega}\ ]@(obs, use_{\mathtt{q};\omega})\phi, \Delta\end{array}}{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}[\ \mathtt{if\ (b)\ \{p\}\ else\ \{q\}}\ \omega \between \\ \qquad\qquad \mathtt{if\ (b)\ \{}\overline{\mathtt{p};\omega}\mathtt{\}\ else\ \{}\overline{\mathtt{q};\omega}\mathtt{\}}\ ]@(obs, use_{\mathtt{p};\omega} \cup use_{\mathtt{q};\omega} \cup \{\mathtt{b}\})\phi, \Delta\end{array}}$$

(with $\mathtt{b}$ boolean variable)

On encountering a conditional statement, symbolic execution splits into two branches, namely the `then` branch and `else` branch. The generation of the conditional statement will result in a conditional. The guard is the same as used in the original program, the `then` branch is the generated version of the source `then` branch continued with the rest of the program after the conditional, and the `else` branch is analogous to the `then` branch.

Note that the statements following the conditional statement are symbolically executed on both branches. This leads to duplicated code in the generated program, and, potentially to code size duplication at each occurrence of a conditional statement. One note in advance: code duplication can be avoided when applying a similar technique as presented later in connection with the loop translation rule. However, it is noteworthy that the application of this rule might have also advantages: as discussed in Chapter 3, symbolic execution and partial evaluation can be interleaved resulting in (considerably) smaller execution traces. Interleaving symbolic execution and partial evaluation is orthogonal to the approach presented here and can be combined easily. In several cases this can lead to different and drastically specialized and therefore smaller versions of the remainder program ω and $\overline{\omega}$. The *use* set is extended canonically by joining the *use* sets of the different branches and the guard variable.

$$\text{loopInvariant}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\, inv, \Delta \\ \Gamma, \mathcal{U}\mathcal{V}_{mod}(\mathtt{b} \wedge inv) \Longrightarrow \mathcal{U}\mathcal{V}_{mod} \\ \qquad\qquad\qquad [\,\mathtt{p} \between \overline{\mathtt{p}}\,]@(use_1 \cup \{\mathtt{b}\}, use_2) inv, \Delta \\ \Gamma, \mathcal{U}\mathcal{V}_{mod}(\neg \mathtt{b} \wedge inv) \Longrightarrow \mathcal{U}\mathcal{V}_{mod}[\,\omega \between \overline{\omega}\,]@(obs, use_1)\phi, \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\,\mathtt{while(b)\{p\}}\,\omega \between \mathtt{while(b)}\{\overline{\mathtt{p}}\}\,\overline{\omega}\,]@(obs, use_1 \cup use_2 \cup \{b\})\phi, \Delta}$$

On the logical side the loop invariant rule is as expected and has three premises. Here we are interested in compilation of the analyzed program rather than proving its correctness. Therefore, it is sufficient to use *true* as a trivial invariant or to use any automatically obtainable invariant. In this case the first premise (`init`) ensuring that the loop invariant is initially valid contributes nothing to the program compilation process and is ignored from here onward (if *true* is used as invariant then it holds trivially).

Two things are of importance: the third premise (`use case`) executes only the program following the loop. Furthermore, this code fragment is not executed by any of the other branches and, hence, we avoid unnecessary code duplication. The second observation is that variables read by the program in the third premise may be assigned in the loop body, but not read in the loop body. Obviously, we have to prevent that the assignment rule discards those assignments when compiling the loop body. Therefore, in the *obs* for the second premise (`preserves`), we must include the used variables of the `use case` premise and, for similar reasons, the program variable(s) read by the loop guard. In practice this is achieved by first executing the `use case` premise of the loop invariant rule and then including the resulting use_1 set in the *obs* of the `preserves` premise. The work flow of the synthesizing loop is shown in Figure 4.3.

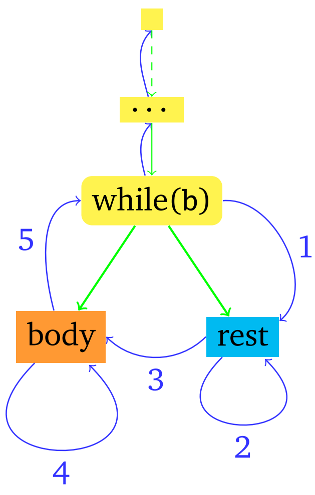

Figure 4.3: Work flow of synthesizing loop.

Now we show the program transformation in action.

Example 7. *Given observable locations obs*=$\{\mathrm{x}\}$, *we perform program transformation for the following SiJa program.*

```
y = y + z;
if (b) {
  y = z++;
  x = z;
}
else {
  z = 1;
  x = y + z;
  y = x;
  x = y + 2;
}
```

In the first phase, we do symbolic execution using the extended sequent calculus shown in Figure 4.2. We use sp_i to denote the program to be generated, and use_i to denote the used variable set. To ease the presentation, we omit postcondition ϕ, as well as unnecessary formulas Γ and Δ. The first active statement is an assignment, so the assignment rule is applied. A conditional is encountered. After the application of ifElse rule, the result is the symbolic execution tree shown in Figure 4.4.

Now the symbolic execution tree splits into 2 branches. $\mathcal{U}_1$ denotes the update computed in the previous steps: $\{\mathrm{y} := \mathrm{y} + \mathrm{z}\}$. We first concentrate on the `then` branch, where the condition b is *True*. The first active statement $\mathrm{y} = \mathrm{z}++;$ is a complex statement. We decompose it into

$$\dfrac{\dfrac{\mathcal{U}_1\mathtt{b} \Longrightarrow \mathcal{U}_1[\;\mathtt{y=z++;}\ldots \between sp_2\;]@(\{\mathtt{x}\},use_2) \qquad \mathcal{U}_1\neg\mathtt{b} \Longrightarrow \mathcal{U}_1[\;\mathtt{z=1;}\ldots \between sp_3\;]@(\{\mathtt{x}\},use_3)}{\Longrightarrow \{\mathtt{y:=y+z}\}[\;\mathtt{if(b)\{\ldots\}else\{\ldots\}} \between sp_1\;]@(\{\mathtt{x}\},use_1)}}{\Longrightarrow [\;\mathtt{y=y+z;}\ldots \between sp_0\;]@(\{\mathtt{x}\},use_0)}$$

Figure 4.4: Symbolic execution tree until conditional.

3 simple statements using the postInc rule introduced in Figure 2.10. Then after a few applications of the assignment rule followed by the emptyBox rule, the symbolic execution tree in this sequential block is shown in Figure 4.5.

$$\begin{array}{c}
\mathcal{U}_1\mathtt{b} \Longrightarrow \mathcal{U}_1\{\mathtt{t:=z}\}\{\mathtt{z:=z+1}\}\{\mathtt{y:=t}\}\{\mathtt{x:=z}\}@(\{\mathtt{x}\},\_) \\ \hline
\mathcal{U}_1\mathtt{b} \Longrightarrow \mathcal{U}_1\{\mathtt{t:=z}\}\{\mathtt{z:=z+1}\}\{\mathtt{y:=t}\}\{\mathtt{x:=z}\}[\;\between sp_8\;]@(\{\mathtt{x}\},use_8) \\ \hline
\mathcal{U}_1\mathtt{b} \Longrightarrow \mathcal{U}_1\{\mathtt{t:=z}\}\{\mathtt{z:=z+1}\}\{\mathtt{y:=t}\}[\;\mathtt{x=z;} \between sp_7\;]@(\{\mathtt{x}\},use_7) \\ \hline
\mathcal{U}_1\mathtt{b} \Longrightarrow \mathcal{U}_1\{\mathtt{t:=z}\}\{\mathtt{z:=z+1}\}[\;\mathtt{y=t;}\ldots \between sp_6\;]@(\{\mathtt{x}\},use_6) \\ \hline
\mathcal{U}_1\mathtt{b} \Longrightarrow \mathcal{U}_1\{\mathtt{t:=z}\}[\;\mathtt{z=z+1;\ y=t;}\ldots \between sp_5\;]@(\{\mathtt{x}\},use_5) \\ \hline
\mathcal{U}_1\mathtt{b} \Longrightarrow \mathcal{U}_1[\;\mathtt{int\ t=z;\ z=z+1;\ y=t;}\ldots \between sp_4\;]@(\{\mathtt{x}\},use_4) \\ \hline
\mathcal{U}_1\mathtt{b} \Longrightarrow \mathcal{U}_1[\;\mathtt{y=z++;}\ldots \between sp_2\;]@(\{\mathtt{x}\},use_2)
\end{array}$$

Figure 4.5: Symbolic execution tree of then branch.

Now the source program is empty, so we can start generating a program for this sequential block. By applying the emptyBox rule in the other direction, we get sp_8 as _ (empty program) and $use_8=\{\mathtt{x}\}$. The next rule application is assignment. Because $\mathtt{x} \in use_8$, the assignment $\mathtt{x=z;}$ is generated and the used variable set is updated by removing x but adding z. So we have sp_7: $\mathtt{x=z;}$ and $use_7=\{\mathtt{z}\}$. In the next step, despite another assignment rule application, no statement is generated because $\mathtt{y} \notin use_7$, and sp_6 and use_6 are identical to sp_7 and use_7. Following 3 more assignment rule applications, in the end we get sp_2: $\mathtt{z=z+1;x=z;}$ and $use_2=\{\mathtt{z}\}$. So $\mathtt{z=z+1;x=z;}$ is the program synthesized in this sequential block.

So far we have done the program transformation for the `then` branch. Analogous to this, we can generate the program for the `else` branch. After the first phase of symbolic execution, the symbolic execution tree is built as shown in Figure 4.6. In the second phase, the program is synthesized after applying a sequence of assignment rules. The resulting program for this sequential block is sp_3: $\mathtt{z=1;x=y+z;y=x;x=y+2;}$, while $use_3=\{\mathtt{y}\}$.

Now we have synthesized the program for both sequential blocks. Back to the symbolic execution tree shown in Figure 4.4, we can build a conditional by applying the ifElse rule. The result is sp_1: $\mathtt{if(b)\ \{z=z+1;x=z;\}\ else\ \{z=1;x=y+z;y=x;x=y+2;\}}$, and $use_1=\{\mathtt{b,z,y}\}$. After a final assignment rule application, the program generated is shown in Figure 4.7.

$$\begin{array}{c}
\mathcal{U}_1 \neg \mathtt{b} \Longrightarrow \mathcal{U}_1 \{\mathtt{z} := 1\}\{\mathtt{x} := \mathtt{y} + \mathtt{z}\}\{\mathtt{y} := \mathtt{x}\}\{\mathtt{x} := \mathtt{y} + 2\} @ (\{\mathtt{x}\}, \_) \\
\hline
\mathcal{U}_1 \neg \mathtt{b} \Longrightarrow \mathcal{U}_1 \{\mathtt{z} := 1\}\{\mathtt{x} := \mathtt{y} + \mathtt{z}\}\{\mathtt{y} := \mathtt{x}\}\{\mathtt{x} := \mathtt{y} + 2\}[\ \between\ sp_{12}\ ] @ (\{\mathtt{x}\}, use_{12}) \\
\hline
\mathcal{U}_1 \neg \mathtt{b} \Longrightarrow \mathcal{U}_1 \{\mathtt{z} := 1\}\{\mathtt{x} := \mathtt{y} + \mathtt{z}\}\{\mathtt{y} := \mathtt{x}\}[\ \mathtt{x} = \mathtt{y} + 2;\ \between\ sp_{11}\ ] @ (\{\mathtt{x}\}, use_{11}) \\
\hline
\mathcal{U}_1 \neg \mathtt{b} \Longrightarrow \mathcal{U}_1 \{\mathtt{z} := 1\}\{\mathtt{x} := \mathtt{y} + \mathtt{z}\}[\ \mathtt{y} = \mathtt{x}; \ldots\ \between\ sp_{10}\ ] @ (\{\mathtt{x}\}, use_{10}) \\
\hline
\mathcal{U}_1 \neg \mathtt{b} \Longrightarrow \mathcal{U}_1 \{\mathtt{z} := 1\}[\ \mathtt{x} = \mathtt{y} + \mathtt{z}; \ldots\ \between\ sp_{9}\ ] @ (\{\mathtt{x}\}, use_{9}) \\
\hline
\mathcal{U}_1 \neg \mathtt{b} \Longrightarrow \mathcal{U}_1 [\ \mathtt{z} = 1; \ldots\ \between\ sp_{3}\ ] @ (\{\mathtt{x}\}, use_{3})
\end{array}$$

Figure 4.6: Symbolic execution tree of else branch.

```
y = y + z;
if (b) {
  z = z + 1;
  x = z;
}
else {
  z = 1;
  x = y + z;
  y = x;
  x = y + 2;
}
```

Figure 4.7: The generated program for Example 7.

Remark. *Our approach to program transformation will generate a program that only consists of simple statements. The generated program is optimized to a certain degree, because the used variable set avoids generating unnecessary statements. In this sense, our program transformation framework can be considered as* program specialization*. In fact, during the symbolic execution phase, we can interleave partial evaluation actions, i.e., constant propagation, deadcode-elimination, safe field access and type inference (Section 3.2.2). It will result in a more optimized program.*

Example 8. *We specialize the program shown in Example 7. In the first phase, symbolic execution is interleaved with simple partial evaluation actions.*

In the first 2 steps of symbolic execution until conditional, no partial evaluation is involved. The resulting symbolic execution tree is identical to that shown in Figure 4.4.

There are 2 branches in the symbolic execution tree. Symbolical execution of the `then` branch is the same as in Example 7. It builds the same symbolic execution tree (Figure 4.5).

Notice that after executing the statement $\mathtt{t} = \mathtt{z};$, we did not propagate this information to the statement $\mathtt{y} = \mathtt{t};$ and rewrite it to $\mathtt{y} = \mathtt{z};$. The reason being $\mathtt{z}$ is reassigned in the statement $\mathtt{z} = \mathtt{z} + 1;$ before $\mathtt{y} = \mathtt{t};$, thus $\mathtt{z}$ is not a "constant" and we cannot apply constant propagation. In the program generation phase, we also get sp_2: $\mathtt{z = z + 1; x = z;}$ and $use_2 = \{\mathtt{z}\}$ for this sequential block.

The first step of symbolic execution of the `else` branch is the application of the assignment rule on $\mathtt{z = 1;}$. Now we can perform constant propagation and rewrite the following statement $\mathtt{x = y + z;}$ into $\mathtt{x = y + 1;}$. The next step is a normal application of the assignment rule on $\mathtt{x = y + 1;}$. Now we apply the assignment rule on $\mathtt{y = x;}$. Since neither x nor y is reassigned before the statement $\mathtt{x = y + 2;}$, x is considered as a "constant" and we do another step of constant propagation. The statement $\mathtt{x = y + 2;}$ is rewritten into $\mathtt{x = x + 2;}$. After final application of the assignment rule and emptyBox rule, we get the symbolic execution tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{U}_1 \neg b \Longrightarrow \mathcal{U}_1\{z := 1\}\{x := y+1\}\{y := x\}\{x := x+2\}@(\{x\},\_)}{\mathcal{U}_1 \neg b \Longrightarrow \mathcal{U}_1\{z := 1\}\{x := y+1\}\{y := x\}\{x := x+2\}[\ \between\ sp_{12}\ ]@(\{x\}, use_{12})}}{\mathcal{U}_1 \neg b \Longrightarrow \mathcal{U}_1\{z := 1\}\{x := y+1\}\{y := x\}[\ x = x + 2;\ \between\ sp_{11}\ ]@(\{x\}, use_{11})}}{\mathcal{U}_1 \neg b \Longrightarrow \mathcal{U}_1\{z := 1\}\{x := y+1\}[\ y = x; \ldots\ \between\ sp_{10}\ ]@(\{x\}, use_{10})}}{\mathcal{U}_1 \neg b \Longrightarrow \mathcal{U}_1\{z := 1\}[\ x = y + 1; \ldots\ \between\ sp_9\ ]@(\{x\}, use_9)}}{\mathcal{U}_1 \neg b \Longrightarrow \mathcal{U}_1[\ z = 1; \ldots\ \between\ sp_3\ ]@(\{x\}, use_3)}$$

In the second phase of program generation, after applying the emptyBox rule and 4 times assignment rules, we get sp_3: $\mathtt{x = y + 1; x = x + 2;}$ and $use_3 = \{\mathtt{y}\}$.

Combining both branches, we finally get the specialized version of the original, shown in Figure 4.8.

```
y = y + z;
if (b) {
  z = z + 1;
  x = z;
}
else {
  x = y + 1;
  x = x + 2;
}
```

Figure 4.8: The generated program for Example 8.

Compared to the result shown in Figure 4.7, we generated a more optimized program by interleaving partial evaluation actions during symbolic execution phase. Further optimization can be made by involving updates during program generation. This will be discussed later.

4.3 Soundness

Theorem 1. *The extended sequent calculus rules are sound.*

The deductive description of the presented program transformation rule system enables us to reuse standard proof techniques applied in soundness proofs for classical logic calculi.

The basic approach is to prove soundness for each rule. The soundness of the whole method is then a consequence of the soundness theorem for classical sequent calculi $\vdash$:

Theorem 2. *If all rules of the proof system $\vdash$ are sound, then the proof system is sound.*

The soundness proof for the classical calculus rules remains unchanged. The interesting part is the soundness proof for the rules dealing with the weak bisimulation modality. The soundness proof of these rules requires in particular to show, that the transformed program is equivalent to the original one up to weak bisimulation with respect to a specified set of observable locations *obs*.

We need first some lemmas which establish simple properties that are mostly direct consequences of the respective definitions given in the Section 4.1.

The following lemma allows us to extend the weak bisimulation relation for two states when we know that they coincide on the value of x.

Lemma 5. *Let $s_1, s_2 \in S$ be observation equivalent $s_1 \approx_{obs} s_2$ and $\mathtt{x} : T \in \mathsf{PV}$. If $s_1(\mathtt{x}) = s_2(\mathtt{x})$ then $s_1 \approx_{obs \cup \{\mathtt{x}\}} s_2$.*

Proof. Direct consequence of Definition 13. □

The next lemma states that two bisimilar states remain bisimular if both are updated by identical assignments:

Lemma 6. *Let $s_1, s_2 \in S$ be observation equivalent $s_1 \approx_{obs} s_2$. If s_1', s_2' are such that $s_1' = s_1[\mathtt{x} \leftarrow d]$ and $s_2' = s_2[\mathtt{x} \leftarrow d]$ for a program variable $\mathtt{x} : T$ and domain element $d \in D(T)$ then $s_1' \approx_{obs} s_2'$.*

Proof. Direct consequence of Definition 13. □

We need further that the bisimulation relation is anti-monotone with respect to the set of observable locations.

Lemma 7. *Given two programs $\mathtt{p}, \mathtt{q}$ and location sets loc_1, loc_2 with $loc_1 \subseteq loc_2$. If $\mathtt{p} \approx_{loc_2} \mathtt{q}$ then also $\mathtt{p} \approx_{loc_1} \mathtt{q}$.*

Proof. Direct consequence of Definition 17. □

Finally, we need the fact that changes to unobserved locations have no effect on the bisimulation relation between two states:

Lemma 8. *Let loc denote a set of locations, $\mathtt{l} : T \in \mathsf{PV}$ and $s_1, s_2 \in S$.*

If $\mathtt{l} \notin loc$ and $s_1 \approx_{loc} s_2$ then for all $d \in \mathscr{D}_T$:

$$s_1[\mathtt{l} \leftarrow d] \approx_{loc} s_2$$

Proof. Direct consequence of Definition 13. □

We can now turn to the soundness proof for the calculus rules. We prove here exemplarily that the assignment rule for local variables is sound. The rule is central to the approach as it performs a state change.

Lemma 9. *The rule*

$$\text{assignment}\quad \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{l} := \mathtt{r}\}[\ \omega \between \overline{\omega}\ ]@(obs, use)\phi, \Delta}{\left(\begin{array}{ll} \Gamma \Longrightarrow \mathcal{U}[\ \mathtt{l} = \mathtt{r}; \omega \between \mathtt{l} = \mathtt{r}; \overline{\omega}\ ]@(obs, use - \{\mathtt{l}\} \cup \{\mathtt{r}\})\phi, \Delta & \textit{if } \mathtt{l} \in use \\ \Gamma \Longrightarrow \mathcal{U}[\ \mathtt{l} = \mathtt{r}; \omega \between \overline{\omega}\ ]@(obs, use)\phi, \Delta & \textit{otherwise} \end{array}\right)}$$

(with l*,* r *local variables)*
is sound.

Proof. To check the soundness of the rule, we have to prove that if all premises of the rule are valid then its conclusion is also valid.

We fix a first-order structure D, a state s and a variable assignment β. Further, we assume that for all formulas $\gamma \in \Gamma$: $val_{D,s,\beta}(\gamma) = tt$ and for all formulas $\delta \in \Delta$: $val_{D,s,\beta}(\Delta) = ff$ holds. Otherwise, the conclusion is trivially satisfied by D, s, β. Hence, we can assume that

$$val_{D,s,\beta}(\mathcal{U}\{\mathtt{l} := \mathtt{r}\}[\ \omega \between \overline{\omega}\ ]@(obs, use)\phi) = tt$$

or, equivalently,

$$val_{D,\widehat{s},\beta}([\ \omega \between \overline{\omega}\ ]@(obs, use)\phi) = tt \tag{4.1}$$

where

$$s_{\mathcal{U}} := val_{D,s,\beta}(\mathcal{U})(s),\ \ \widehat{s} := val_{D,s_{\mathcal{U}},\beta}(\mathtt{l} := \mathtt{r})(s_{\mathcal{U}}) = val_{D,s,\beta}(\mathcal{U} \| \mathcal{U}(\mathtt{l} := \mathtt{r}))(s)$$

holds.

Case 1 ($\mathtt{l} \in use$):

We have to show that

$$\begin{aligned} & val_{D,s,\beta}(\mathcal{U}[\,\mathtt{l}=\mathtt{r};\omega \between \mathtt{l}=\mathtt{r};\overline{\omega}\,]@(obs,use')\phi) \\ = \; & val_{D,s_{\mathcal{U}},\beta}([\,\mathtt{l}=\mathtt{r};\omega \between \mathtt{l}=\mathtt{r};\overline{\omega}\,]@(obs,use')\phi) \\ = \; & tt \end{aligned}$$

with $use' := use - \{\mathtt{l}\} \cup \{\mathtt{r}\}$ holds.

To prove that $val_{D,s_{\mathcal{U}},\beta}([\,\mathtt{l}=\mathtt{r};\omega \between \mathtt{l}=\mathtt{r};\overline{\omega}\,]@(obs,use')\phi) = tt$ we need to check the three items of Definition 20:

Item 1 is satisfied if

$$val_{D,s,\beta}(\mathcal{U}[\mathtt{l}=\mathtt{r};\omega]\phi) = tt$$

holds. This is a direct consequence from the correctness of the sequent calculus presented in Section 2.4.

Item 2 $use' \supseteq usedVar(s,\mathtt{l}=\mathtt{r};\overline{\omega},obs)$ expresses that use' captures at least all used variables and it is a direct consequence of the definition of *usedVar*. By assumption *use* contains at least all variables actually read by $\overline{\omega}$. The program $\mathtt{l}=\mathtt{r};\overline{\omega}$ redefines $\mathtt{l}$ which can be safely removed from *use* while variable $\mathtt{r}$ is read and needs to be added.

Item 3 is the last remaining item that needs to be proven, i.e., that the two programs in the conclusion are actually weak bisimular with respect to the location set *obs*.

We have to show that for all $s_1 \approx_{use'} s_{\mathcal{U}}$:

$$val_{D,s_{\mathcal{U}}}(\mathtt{l}=\mathtt{r};\omega) \;\approx_{obs}\; val_{D,s_1}(\mathtt{l}=\mathtt{r};\overline{\omega})$$

holds. Following the semantics definitions given in Figure 2.6 we get

$$val_{D,s_{\mathcal{U}}}(\mathtt{l}=\mathtt{r};\omega) = \bigcup_{s' \in val_{D,s_{\mathcal{U}}}(\mathtt{l}=\mathtt{r};)} val_{D,s'}(\omega) = val_{D,\widehat{s}}(\omega)$$

and

$$val_{D,s_1}(\mathtt{l}=\mathtt{r};\overline{\omega}) = \bigcup_{s'_1 \in val_{D,s_1}(\mathtt{l}=\mathtt{r};)} val_{D,s'_1}(\overline{\omega}) = val_{D,\widehat{s_1}}(\overline{\omega}) \;\text{ with } \{\widehat{s_1}\} = val_{D,s_1}(\mathtt{l}=\mathtt{r};)$$

As use' contains $\mathtt{r}$ and because $s_1 \approx_{use'} s_{\mathcal{U}}$ we get

$$s_{\mathcal{U}}(\mathtt{r}) = s_1(\mathtt{r}) \tag{4.2}$$

and, hence,

$$\widehat{s}(\mathtt{l}) = \widehat{s_1}(\mathtt{l}) \tag{4.3}$$

Applying Lemma 6 we get

$$\begin{aligned}
&\widehat{s} \approx_{use'} \widehat{s_1} \\
\Longleftrightarrow\ &\widehat{s} \approx_{use-\{\mathtt{l}\}\cup\{\mathtt{r}\}} \widehat{s_1} \\
\underset{\text{Lemma 7}}{\Rightarrow}\ &\widehat{s} \approx_{use-\{\mathtt{l}\}} \widehat{s_1} \\
\underset{(4.3)}{\Rightarrow}\ &\widehat{s} \approx_{use} \widehat{s_1}
\end{aligned}$$

With assumption (4.1) and Definition 18, we get $val_{D,\widehat{s}}(\omega) \approx_{obs} val_{D,\widehat{s_1}}(\overline{\omega})$ and hence

$$val_{D,s_{\mathcal{U}}}(\mathtt{l}=\mathtt{r};\omega) = val_{D,\widehat{s}}(\omega) \approx_{obs} val_{D,\widehat{s_1}}(\overline{\omega}) = val_{D,s_1}(\mathtt{l}=\mathtt{r};\overline{\omega})$$

Case 2 ($\mathtt{l} \notin use$): As for case 1 we have to check all three items. The first item is identical to case 1 and the second item is trivial as the transformed program does not change. Item 3 remains to be checked, i.e., for an arbitrary s_1 with

$$s_1 \approx_{use'} s_{\mathcal{U}} \tag{4.4}$$

we have to prove that

$$val_{D,s_{\mathcal{U}}}(\mathtt{l}=\mathtt{r};\omega)\ \approx_{obs}\ val_{D,s_1}(\overline{\omega})$$

holds (i.e., that the final states are observation equivalent), we have to use the fact that $\mathtt{l} \notin use$ and that item 2 holds, i.e., that use contains at least all variables read by $\overline{\omega}$.

$$\begin{aligned}
&s_1 \approx_{use'} s_{\mathcal{U}} \\
\Rightarrow\ &s_1 \approx_{use} s_{\mathcal{U}} \\
\underset{Lemma\ 8}{\Rightarrow}\ &s_1 \approx_{use} \widehat{s} \\
\underset{(4.1)}{\Rightarrow}\ &val_{D,\widehat{s}}(\omega)\ \approx_{obs}\ val_{D,s_1}(\overline{\omega}) \\
\underset{(4.1)}{\Rightarrow}\ &val_{D,s_{\mathcal{U}}}(\mathtt{l}=\mathtt{r};\omega) = val_{D,\widehat{s}}(\omega)\ \approx_{obs}\ val_{D,s_1}(\overline{\omega})
\end{aligned}$$

□

We conclude this section with a short discussion of the loop invariant rule. The interesting aspect of the loop invariant rule is that the observable location set *obs* of the second premise differs from the others. This allows us to establish a connection to the notion of a program context as used in compositional correctness proofs.

Compositional compiler correctness proofs consider the context $C(\circ)$ in which the compiled entity p is *used*. A context C is a description contain the placeholder $\circ$ which can be instantiated by 'any' program entity q.

The idea is to formalize a stable interface on which p can rely on and with which p interacts. A compositional compiler must now be able to compile p such that a given correctness criteria are satisfied for the compilation $\mathtt{p}_{compiled}$ with respect to C.

The observable location set *obs* in the presented approach is similar to the context as described above. It specifies which effects must be preserved by the compiler (program transformer). E.g., when the program p to be transformed is a method body, then the observable set contains only the location which refers to the result value of the method and implicitly, all heap locations.

If the effect on these locations produced by the transformed program is indistinguishable from the respective effect of the original program, then the program transformer is considered correct. In case of the loop invariant rule, the loop body is transformed *independently* in the second branch. It would not be enough to just use the original context instead, we must demand that all effects on local variables used by the code following the loop statement as well as the loop guard variable are preserved.

4.4 Optimization

The previously introduced program transformation technique generates a program that consists only of simple statements. With the help of the used variable set, we avoid generating unnecessary statements, so the program is optimized to a certain level. An optimization can be made to interleave partial evaluation actions with symbolic execution in the first phase.

4.4.1 Sequentialized Normal Form of Updates

Updates reflect the state of program execution. In particular, the update in a sequential block records the evaluation of the locations in that sequential block. We can involve updates in the second phase of program generation, which leads to further optimization opportunities. As defined in Definition 5, updates in normal form are in the form of static single assignment (SSA). It is easy to maintain normal form of updates in a sequential block when applying the extended sequent calculus rules of Figure 4.2. This can be used for further optimization of the generated program.

Take the assignment rule for example: after each forward rule application, we do an update simplification step to maintain the normal form of the update for that sequential block; when a statement is synthesized by applying the rule backwards, we use the *update* instead of the executed assignment statement, to obtain the value of the location to be assigned; then we generate the assignment statement with that value.

Example 9. *Consider the following program:*

```
i = j + 1;
j = i;
i = j + 1;
```

After executing the first two statements and update simplification, we obtain the normal form update $\mathcal{U}_2^{nf} = \{\mathtt{i} := \mathtt{j} + 1 \| \mathtt{j} := \mathtt{j} + 1\}$*. Doing the same with the third statement results in* $\mathcal{U}_3^{nf} = \{\mathtt{j} := \mathtt{j} + 1 \| \mathtt{i} := \mathtt{j} + 2\}$*, which implies that in the final state* $\mathtt{i}$ *has value* $\mathtt{j} + 2$ *and* $\mathtt{j}$ *has value* $\mathtt{j} + 1$.

Let $\mathtt{i}$ *be the only observable location, for which a program is now synthesized bottom-up, starting with the third statement. The rules in Figure 4.2 would allow to generate the statement* $\mathtt{i} = \mathtt{j} + 1$;*. But, reading the value of location* $\mathtt{i}$ *from* $\mathcal{U}_3^{nf}$ *as sketched above, the statement* $\mathtt{i} = \mathtt{j} + 2$; *is generated. This reflects the current value of* $\mathtt{j}$ *along the sequential block and saves an assignment.*

A first attempt to formalize our ideas is the following assignment rule:

$$\frac{\Gamma \Longrightarrow \mathcal{U}_1^{nf}[\ \omega \between \overline{\omega}\ ]@(obs, use)\phi, \Delta}{\begin{pmatrix} \Gamma \Longrightarrow \mathcal{U}^{nf}[\ \mathtt{l} = \mathtt{r};\omega \between \mathtt{l} = \mathtt{r}_1;\overline{\omega}\ ]@(obs, use - \{\mathtt{l}\} \cup \{\mathtt{r}\})\phi, \Delta & \text{if } \mathtt{l} \in use \\ \Gamma \Longrightarrow \mathcal{U}^{nf}[\ \mathtt{l} = \mathtt{r};\omega \between \overline{\omega}\ ]@(obs, use)\phi, \Delta & \text{otherwise} \end{pmatrix}}$$
$$\text{(with } \mathcal{U}_1^{nf} = \{\ldots \| \mathtt{l} := \mathtt{r}_1\} \text{ being the normal form of } \mathcal{U}^{nf}\{\mathtt{l} := \mathtt{r}\})$$

However, this rule is not sound. If we continue Example 9 with synthesizing the first two assignments, we obtain $\mathtt{j} = \mathtt{j} + 1; \mathtt{i} = \mathtt{j} + 2;$ by using the new rule, which is clearly incorrect, because $\mathtt{i}$ has final value $\mathtt{j} + 3$ instead of $\mathtt{j} + 2$. The problem is that the values of locations in the normal form update are independently synthesized from each other and do not reflect how one statement is affected by the execution of previous statements in sequential execution. To ensure correct usage of updates in program generation, we introduce the concept of a *sequentialized normal form* (SNF) of an update. Intuitively, it is the update of the normal form in which every involved assignment statement is independent of each other.

Definition 21 (Elementary update independence)**.** *An elementary update* $\mathtt{l}_1 := exp_1$ *is* independent *from another elementary update* $\mathtt{l}_2 := exp_2$*, if* $\mathtt{l}_1$ *does not occur in* exp_2 *and* $\mathtt{l}_2$ *does not occur in* exp_1.

Definition 22 (Sequentialized normal form update). *An update is in* sequentialized normal form, *denoted by* $\mathscr{U}^{snf}$, *if it has the shape of a sequence of two parallel updates* $\{u_1^a \| \dots \| u_m^a\}\{u_1 \| \dots \| u_n\}$, $m \geq 0, n \geq 0$.

$\{u_1 \| \dots \| u_n\}$ *is the* core *update, denoted by* $\mathscr{U}^{snf_c}$, *where each* u_i *is an* elementary update *of the form* $\mathtt{l}_i := exp_i$, *and all* u_i, u_j $(i \neq j)$ *are independent and have no conflict.*

$\{u_1^a \| \dots \| u_m^a\}$ *is the* auxiliary *update, denoted by* $\mathscr{U}^{snf_a}$, *where (i) each* u_i^a *is of the form* $\mathtt{l}^k := \mathtt{l}$ $(k \geq 0)$*; (ii)* $\mathtt{l}$ *is a program variable; (iii)* $\mathtt{l}^k$ *is a* fresh *program variable not occurring anywhere else in* $\mathscr{U}^{snf_a}$ *and not occurring in the location set of the core update* $\mathtt{l}^k \notin \{\mathtt{l}_i | 0 \leq i \leq n\}$*; (iv) there is no conflict between* u_i^a *and* u_j^a *for all* $i \neq j$.

Any normal form update whose elementary updates are independent is also an SNF update that has only a core part.

Example 10 (SNF update). *For the following updates,*

- $\{\mathtt{i}^0 := \mathtt{i} \| \mathtt{i}^1 := \mathtt{i}\}\{\mathtt{i} := \mathtt{i}^0 + 1 \| \mathtt{j} := \mathtt{i}^1\}$ *is in sequentialized normal form.*
- $\{\mathtt{i}^0 := \mathtt{j} \| \mathtt{i}^1 := \mathtt{i}\}\{\mathtt{i} := \mathtt{i}^0 + 1 \| \mathtt{j} := \mathtt{i}^1\}$ *and* $\{\mathtt{i}^0 := \mathtt{i} + 1 \| \mathtt{i}^1 := \mathtt{i}\}\{\mathtt{i} := \mathtt{i}^0 + 1 \| \mathtt{j} := \mathtt{i}^1\}$ *are not in sequentialized normal form:* $\mathtt{i}^0 := \mathtt{j}$ *has different base variables on the left and right, while* $\mathtt{i}^0 := \mathtt{i} + 1$ *has a complex term on the right, both contradicting (i).*
- $\{\mathtt{i}^0 := \mathtt{i} \| \mathtt{i}^1 := \mathtt{i}\}\{\mathtt{i} := \mathtt{i}^0 + 1 \| \mathtt{j} := \mathtt{i}\}$ *is not in sequentialized normal form, because* $\mathtt{i} := \mathtt{i}^0 + 1$ *and* $\mathtt{j} := \mathtt{i}$ *are not independent.*

To compute the SNF of an update, in addition to the rules given in Figure 2.7 we need two more rules shown in Figure 4.9.

(*associativity*) $\{u_1\}\{u_2\}\{u_3\} \rightsquigarrow \{u_1\}(\{u_2\}\{u_3\})$

(*introducing auxiliary*) $\{u\} \rightsquigarrow \{\mathtt{x}^0 := \mathtt{x}\}(\{\mathtt{x} := \mathtt{x}^0\}\{u\})$, where $\mathtt{x}^0 \notin pv$

Figure 4.9: Rules for computing SNF updates.

Lemma 10. *The associativity rule and introducing auxiliary rule are sound.*

Proof. We use the update simplification rules defined in Figure 2.7 to prove these two rules.
Associativity

The left hand side:

$$\begin{gathered}\{u_1\}\{u_2\}\{u_3\} \\ \rightsquigarrow \{u_1 \| \{u_1\}u_2\}\{u_3\} \\ \rightsquigarrow \{u_1 \| \{u_1\}u_2 \| \{u_1 \| \{u_1\}u_2\}u_3\}\end{gathered}$$

The right hand side:

$$\begin{aligned}
&\{u_1\}(\{u_2\}\{u_3\})\\
\rightsquigarrow\ &\{u_1\}\{u_2\|\{u_2\}u_3\}\\
\rightsquigarrow\ &\{u_1\|\{u_1\}(u_2\|\{u_2\}u_3)\}\\
\rightsquigarrow\ &\{u_1\|\{u_1\}u_2\|\{u_1\}\{u_2\}u_3\}\\
\rightsquigarrow\ &\{u_1\|\{u_1\}u_2\|\{u_1\|\{u_1\}u_2\}u_3\}
\end{aligned}$$

So, $\{u_1\}\{u_2\}\{u_3\} = \{u_1\}(\{u_2\}\{u_3\})$. We have proved the associativity rule.

Introducing auxiliary

The right hand side:

$$\begin{aligned}
&\{\mathtt{x}^0 := \mathtt{x}\}(\{\mathtt{x} := \mathtt{x}^0\}\{u\})\\
\rightsquigarrow\ &\{\mathtt{x}^0 := \mathtt{x}\}\{\mathtt{x} := \mathtt{x}^0\}\{u\}\ \text{(associativity)}\\
\rightsquigarrow\ &\{\mathtt{x}^0 := \mathtt{x}\|\{\mathtt{x}^0 := \mathtt{x}\}\mathtt{x} := \mathtt{x}^0\}\{u\}\\
\rightsquigarrow\ &\{\mathtt{x}^0 := \mathtt{x}\|\mathtt{x} := \mathtt{x}\}\{u\}\\
\rightsquigarrow\ &\{\mathtt{x} := \mathtt{x}\}\{u\}\ \text{(since } \mathtt{x}^0 \notin pv)\\
\rightsquigarrow\ &\{u\}
\end{aligned}$$

So the introducing auxiliary rule is proven. □

We can maintain the SNF of an update on a sequential block as follows: after executing a program statement, apply the associativity rule and compute the core update; if the newly added elementary update $\mathtt{l} := \mathtt{r}$ is not independent from some update in the core, then apply introducing auxiliary rule to introduce $\{\mathtt{l}^0 := \mathtt{l}\}$, then compute the new auxiliary update and core update.

4.4.2 Sequent Calculus Rules Involving Updates

With the help of the SNF of an update, a sound assignment rule can be given as follows:

assignment

$$\frac{\Gamma \Longrightarrow \mathcal{U}_1^{snf}[\ \omega \between \overline{\omega}\ ]@(obs, use)\phi, \Delta}{\begin{pmatrix} \Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{l} = \mathtt{r};\omega \between \mathtt{l} = \mathtt{r}_1;\overline{\omega}\ ]@(obs, use - \{\mathtt{l}\} \cup \{\mathtt{r}\})\phi, \Delta & \text{if } \mathtt{l} \in use \\ \Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{l} = \mathtt{r};\omega \between \overline{\omega}\ ]@(obs, use)\phi, \Delta & \text{otherwise} \end{pmatrix}}$$

(where $\mathcal{U}_1^{snf} = \mathcal{U}_1^{snf_a}\{\ldots\|\mathtt{l} := \mathtt{r}_1\}$ is the SNF of$\mathcal{U}^{snf}\{\mathtt{l} := \mathtt{r}\}$)

Whenever the core update is empty, the following auxAssignment rule is used, which means the auxiliary assignments are always generated in the beginning of a sequential block.

auxAssignment

$$\frac{\Gamma \Longrightarrow \mathcal{U}_1^{snf_a}[\ \omega\ \between\ \overline{\omega}\ ]@(obs, use)\phi, \Delta}{\left(\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}^{snf_a}[\ \omega\ \between\ \mathtt{T_l}\ \mathtt{l}^0 = \mathtt{l};\overline{\omega}\ ]@(obs, use - \{\mathtt{l}^0\} \cup \{\mathtt{l}\})\phi, \Delta & \text{if } \mathtt{l}^0 \in use \\ \Gamma \Longrightarrow \mathcal{U}^{snf_a}[\ \omega\ \between\ \overline{\omega}\ ]@(obs, use)\phi, \Delta & \text{otherwise}\end{array}\right)}$$

(where $\mathcal{U}^{snf_a} = \{u\}$ and $\mathcal{U}_1^{snf_a} = \{u \| \mathtt{l}^0 := \mathtt{l}\}$ being the auxiliary updates)

Most of the other rules are obtained by replacing $\mathcal{U}$ with $\mathcal{U}^{snf}$. Some are shown in Figure 4.10.

Example 11. *We demonstrate that the program from Example 9 is now handled correctly. After executing the first two statements and simplifying the update, we get the normal form update* $\mathcal{U}_2^{nf} = \{\mathtt{i} := \mathtt{j} + 1 \| \mathtt{j} := \mathtt{j} + 1\}$. *Here a dependency issue occurs, so we introduce the auxiliary update* $\{\mathtt{j}^0 := \mathtt{j}\}$ *and simplify to the sequentialized normal form update* $\mathcal{U}_2^{snf} = \{\mathtt{j}^0 := \mathtt{j}\}\{\mathtt{i} := \mathtt{j}^0 + 1 \| \mathtt{j} := \mathtt{j}^0 + 1\}$. *Continuing with the third statement and performing update simplification results in the SNF update* $\mathcal{U}_3^{snf} = \{\mathtt{j}^0 := \mathtt{j}\}\{\mathtt{j} := \mathtt{j}^0 + 1 \| \mathtt{i} := \mathtt{j}^0 + 2\}$. *By applying the rules above, we synthesize the program* `int` $\mathtt{j}^0 = \mathtt{j}; \mathtt{i} = \mathtt{j}^0 + 2;$*, which still saves one assignment and is sound.*

Remark. *Remember that the program is synthesized within a sequential block first and then constructed. The SNF updates used in the above rules are the SNF updates in the current sequential block. A program execution path may contain several sequential blocks. We do keep the SNF update for each sequential block without simplifying them further into a bigger SNF update for the entire execution path. For example in Figure 2.11, the execution path from node n_0 to n_4 involves 3 sequential blocks bl_0, bl_1 and bl_4. When we synthesize the program in bl_4, more precisely, we should write $\mathcal{U}_0^{snf}\mathcal{U}_2^{snf}\mathcal{U}_4^{snf}$ to represent the update used in the rules. However, we just care about the SNF update of bl_4 when generating the program for bl_4, so in the above rules, $\mathcal{U}^{snf}$ refers to $\mathcal{U}_4^{snf}$ and the other SNF updates are omitted.*

Theorem 3. *The extended sequent calculus rules involving updates are sound.*

Proof. Follows from the soundness of the extended sequent calculus rules (Theorem 1), the update simplification rules (Figure 2.7) and Lemma 10. □

Now we revisit Example 7 and show how to generate a more optimized program.

Example 12. *Given observable locations obs=*$\{\mathtt{x}\}$*, specialize the following* SiJa *program by the approach involving updates in the program generation phase.*

$$\text{emptyBox}\ \dfrac{\Gamma \Longrightarrow \mathcal{U}^{snf} @(obs, \_)\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}^{snf}[\ \_ \between \_\ ]@(obs, obs)\phi, \Delta}$$

assignment

$$\dfrac{\Gamma \Longrightarrow \mathcal{U}_1^{snf}[\ \omega \between \overline{\omega}\ ]@(obs, use)\phi, \Delta}{\left(\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{l} = \mathtt{r};\omega \between \mathtt{l} = \mathtt{r}_1;\overline{\omega}\ ]@(obs, use - \{\mathtt{l}\} \cup \{\mathtt{r}\})\phi, \Delta & \text{if } \mathtt{l} \in use \\ \Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{l} = \mathtt{r};\omega \between \overline{\omega}\ ]@(obs, use)\phi, \Delta & \text{otherwise}\end{array}\right)}$$

(where $\mathcal{U}_1^{snf} = \mathcal{U}_1^{snf\,a}\{\ldots \| \mathtt{l} := \mathtt{r}_1\}$ is the SNF of$\mathcal{U}^{snf}\{\mathtt{l} := \mathtt{r}\}$)

auxAssignment

$$\dfrac{\Gamma \Longrightarrow \mathcal{U}_1^{snf\,a}[\ \omega \between \overline{\omega}\ ]@(obs, use)\phi, \Delta}{\left(\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}^{snf\,a}[\ \omega \between \mathtt{T}_1\ \mathtt{l}^0 = \mathtt{l};\overline{\omega}\ ]@(obs, use - \{\mathtt{l}^0\} \cup \{\mathtt{l}\})\phi, \Delta & \text{if } \mathtt{l}^0 \in use \\ \Gamma \Longrightarrow \mathcal{U}^{snf\,a}[\ \omega \between \overline{\omega}\ ]@(obs, use)\phi, \Delta & \text{otherwise}\end{array}\right)}$$

(where $\mathcal{U}^{snf\,a} = \{u\}$ and $\mathcal{U}_1^{snf\,a} = \{u \| \mathtt{l}^0 := \mathtt{l}\}$ being the auxiliary updates)

$$\text{ifElse}\ \dfrac{\begin{array}{l}\Gamma, \mathcal{U}^{snf}\mathtt{b} \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{p};\omega \between \overline{\mathtt{p};\omega}\ ]@(obs, use_{\mathtt{p};\omega})\phi, \Delta \\ \Gamma, \mathcal{U}^{snf}\neg\mathtt{b} \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{q};\omega \between \overline{\mathtt{q};\omega}\ ]@(obs, use_{\mathtt{q};\omega})\phi, \Delta\end{array}}{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{if\ (b)\ \{p\}\ else\ \{q\}}\ \omega \between \\ \qquad\qquad \mathtt{if\ (b)}\ \{\overline{\mathtt{p};\omega}\}\ \mathtt{else}\ \{\overline{\mathtt{q};\omega}\}\ ]@(obs, use_{\mathtt{p};\omega} \cup use_{\mathtt{q};\omega} \cup \{\mathtt{b}\})\phi, \Delta\end{array}}$$

(with $\mathtt{b}$ boolean variable)

$$\text{loopUnwind}\ \dfrac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{if\ (b)\ \{p; while\ (b)\ \{p\}\}}\ \omega \between \\ \qquad\qquad \overline{\mathtt{if\ (b)\ \{p; while\ (b)\ \{p\}\}}\ \omega}\ ]@(obs, use)\phi, \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{while(b)\ \{p\}}\ \omega \between \overline{\mathtt{if\ (b)\ \{p; while(b)\ \{p\}\}}\ \omega}\ ]@(obs, use)\phi, \Delta}$$

$$\text{loopInvariant}\ \dfrac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}^{snf} inv, \Delta \\ \Gamma, \mathcal{U}^{snf}\mathcal{V}_{mod}(\mathtt{b} \wedge inv) \Longrightarrow \mathcal{U}^{snf}\mathcal{V}_{mod} \\ \qquad\qquad\qquad [\ p \between \overline{p}\ ]@(use_1 \cup \{b\}, use_2)inv, \Delta \\ \Gamma, \mathcal{U}^{snf}\mathcal{V}_{mod}(\neg\mathtt{b} \wedge inv) \Longrightarrow \mathcal{U}^{snf}\mathcal{V}_{mod}[\ \omega \between \overline{\omega}\ ]@(obs, use_1)\phi, \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{while(b)\{p\}}\,\omega \between \mathtt{while(b)}\{\overline{\mathtt{p}}\}\,\overline{\omega}\ ]@(obs, use_1 \cup use_2 \cup \{\mathtt{b}\})\phi, \Delta}$$

methodContract$_{C=(pre,post,mod)}$

$$\dfrac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}^{snf}\{\mathtt{prm}_1 := \mathtt{v}_1 \| \ldots \| \mathtt{prm}_n := \mathtt{v}_n\} pre, \Delta \\ \Gamma \Longrightarrow \mathcal{U}^{snf}\{\mathtt{prm}_1 := \mathtt{v}_1 \| \ldots \| \mathtt{prm}_n := \mathtt{v}_n\} \mathcal{V}_{mod} \\ \qquad\qquad\qquad (post \to \{\mathtt{r} := \mathtt{res}\}[\ \omega \between \overline{\omega}\ ]@(obs, use)\phi), \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{r} = \mathtt{m}(\mathtt{v}_1, \ldots, \mathtt{v}_n);\ \omega \between \mathtt{r} = \mathtt{m}(\mathtt{v}_1, \ldots, \mathtt{v}_n);\overline{\omega}\ ]@(obs, use)\phi, \Delta}$$

(Contract C is correct)

Figure 4.10: A collection of sequent calculus rules for program transformation using SNF update.

```
y = y + z;
if (b) {
  y = z++;
  x = z;
}
else {
  z = 1;
  x = y + z;
  y = x;
  x = y + 2;
}
```

In the first phase, we do symbolic execution using the extended sequent calculus rules involving updates given in Figure 4.10. We ignore the postcondition ϕ and unnecessary formulas Γ and Δ. To ease the presentation, we do not mention the update simplification step all the time, but keep in mind that updates within a sequential block are always simplified after each rule application. Also, we just show the sequents computed after sequent calculus rule application and update simplification, but hide the intermediate ones before simplifying the updates. As usual, sp_i denotes the program to be generated, and use_i denotes the used variable set.

The first active statement is an assignment, we apply the assignment rule. After the application of the ifElse rule, the result is the symbolic execution tree shown in Figure 4.11. Here, $\mathcal{U}_1^{snf}$ denotes the sequentialized normal formed update $\{\mathtt{y} := \mathtt{y} + \mathtt{z}\}$. Note that in the path condition, now we only have $\mathtt{b}$ (or $\neg\mathtt{b}$) instead of $\mathcal{U}_1^{snf}\mathtt{b}$ (or $\mathcal{U}_1^{snf}\neg\mathtt{b}$). It is the result of update simplification after applying the ifElse rule.

$$\dfrac{\dfrac{\mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}[\,\mathtt{y = z++;\ldots} \between sp_2\,]@(\{\mathtt{x}\}, use_2) \qquad \neg\mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}[\,\mathtt{z = 1;\ldots} \between sp_3\,]@(\{\mathtt{x}\}, use_3)}{\Longrightarrow \{\mathtt{y := y + z}\}[\,\mathtt{if(b)\{\ldots\}else\{\ldots\}} \between sp_1\,]@(\{\mathtt{x}\}, use_1)}}{\Longrightarrow [\,\mathtt{y = y + z;\ldots} \between sp_0\,]@(\{\mathtt{x}\}, use_0)}$$

Figure 4.11: Symbolic execution tree until conditional.

Now the symbolic execution tree splits into 2 branches.

We symbolically execute the then branch first. The complex statement $\mathtt{y = z++;}$ is decomposed into 3 simple statements using the postInc rule. After the application of the assignment rule on $\mathtt{t = z;}$, the resulting update is $\{\mathtt{t := z}\}$. It is an SNF update that only contains the core part. Then we apply the assignment rule on $\mathtt{z = z + 1;}$. The update we get before simplification is $\{\mathtt{t := z}\}\{\mathtt{z := z + 1}\}$. To simplify this update, we first transform it into parallel form $\{\mathtt{t := z} \| \mathtt{z := z + 1}\}$ using the rules given in Figure 2.7. Notice that $\mathtt{z}$, on the left hand side of $\mathtt{z := z + 1}$, occurs on the right hand side of $\mathtt{t := z}$, so the elementary updates $\mathtt{t := z}$ and $\mathtt{z := z + 1}$ are not independent. To obtain an SNF update, we use the introducing auxiliary rule defined in Figure 4.9. So the update is rewritten as $\{z^0 := z\}(\{z := z^0\}\{t := z \| z := z + 1\})$, where z^0 is a fresh variable and the auxiliary update $\{z^0 := z\}$ is introduced. After simplifying the core part, we finally get the SNF update $\{z^0 := z\}\{t := z^0 \| z := z^0 + 1\}$. From now on, after a few steps application of assignment rule followed by the emptyBox rule, the symbolic execution tree in this sequential block is shown in Figure 4.12.

$$
\begin{array}{c}
\neg b \Longrightarrow \mathcal{U}_1^{snf}\{z^0 := z\}\{t := z^0 \| z := z^0 + 1 \| y := z^0 \| x := z^0 + 1\}@(\{x\}, \_) \\
\hline
b \Longrightarrow \mathcal{U}_1^{snf}\{z^0 := z\}\{t := z^0 \| z := z^0 + 1 \| y := z^0 \| x := z^0 + 1\}[\ \between\ sp_8\ ]@(\{x\}, use_8) \\
\hline
b \Longrightarrow \mathcal{U}_1^{snf}\{z^0 := z\}\{t := z^0 \| z := z^0 + 1 \| y := z^0\}[\ x = z;\ \between\ sp_7\ ]@(\{x\}, use_7) \\
\hline
b \Longrightarrow \mathcal{U}_1^{snf}\{z^0 := z\}\{t := z^0 \| z := z^0 + 1\}[\ y = t; \ldots\ \between\ sp_6\ ]@(\{x\}, use_6) \\
\hline
b \Longrightarrow \mathcal{U}_1^{snf}\{t := z\}[\ z = z + 1;\ y = t; \ldots\ \between\ sp_5\ ]@(\{x\}, use_5) \\
\hline
b \Longrightarrow \mathcal{U}_1^{snf}[\ \mathtt{int}\ t = z;\ z = z + 1;\ y = t; \ldots\ \between\ sp_4\ ]@(\{x\}, use_4) \\
\hline
b \Longrightarrow \mathcal{U}_1^{snf}[\ y = z{+}{+}; \ldots\ \between\ sp_2\ ]@(\{x\}, use_2)
\end{array}
$$

Figure 4.12: Symbolic execution tree of then branch.

Now we start generating the program for this sequential block. By applying the emptyBox rule in the other direction, we get sp_8 as _ and $use_8 = \{x\}$. In the next step, since $x \in use_8$, the assignment $x = z^0 + 1;$ is generated according to the assignment rule involving SNF update. The used variable set is updated by removing x but adding z^0. So we have sp_7: $x = z^0 + 1;$ and $use_7 = \{z^0\}$. The application of 4 more assignment rules generates no more new statement. Now the core update is empty and we can generate the auxiliary assignment according to the auxAssignment rule. In the end, we get for this sequential branch sp_2 : $\mathtt{int}\ z^0 = z; x = z^0 + 1;$ and $use_2 = \{z\}$.

Analogous to this, we can generate the program for the `else` branch. After the first phase of symbolic execution while maintaining the SNF update, Figure 4.13 shows the resulting symbolic execution tree.

$$
\begin{array}{c}
\neg b \Longrightarrow \mathcal{U}_1^{snf}\{y^0 := y\}\{z := 1 \| y := y^0 + 1 \| x := y^0 + 3\}@(\{x\}, \_) \\
\hline
\neg b \Longrightarrow \mathcal{U}_1^{snf}\{y^0 := y\}\{z := 1 \| y := y^0 + 1 \| x := y^0 + 3\}[\ \between\ sp_{12}\ ]@(\{x\}, use_{12}) \\
\hline
\neg b \Longrightarrow \mathcal{U}_1^{snf}\{y^0 := y\}\{z := 1 \| x := y^0 + 1 \| y := y^0 + 1\}[\ x = y + 2;\ \between\ sp_{11}\ ]@(\{x\}, use_{11}) \\
\hline
\neg b \Longrightarrow \mathcal{U}_1^{snf}\{z := 1 \| x := y + 1\}[\ y = x; \ldots\ \between\ sp_{10}\ ]@(\{x\}, use_{10}) \\
\hline
\neg b \Longrightarrow \mathcal{U}_1^{snf}\{z := 1\}[\ x = y + z; \ldots\ \between\ sp_9\ ]@(\{x\}, use_9) \\
\hline
\neg b \Longrightarrow \mathcal{U}_1^{snf}[\ z = 1; \ldots\ \between\ sp_3\ ]@(\{x\}, use_3)
\end{array}
$$

Figure 4.13: Symbolic execution tree of else branch.

In the second phase, the program is synthesized after applying a sequence of assignment rules and a final auxAssignment rule. The result program for this sequential block is $\mathtt{int}\ \mathtt{y}^0 = \mathtt{y}; \mathtt{x} = \mathtt{y}^0 + 2;$, and use_3=$\{\mathtt{y}\}$.

Now the programs for both sequential blocks are synthesized. We can generate the whole program by applying the ifElse rule and assignment rule. The specialized program is shown in Figure 4.14.

```
y = y + z;
if (b) {
  int z⁰ = z;
  x = z⁰ + 1;
}
else {
  int y⁰ = y;
  x = y⁰ + 3;
}
```

Figure 4.14: The generated program for Example 12.

Compared to the specialization results from Example 7 and 8, we get a more optimized program by involving SNF updates during the generation phase. The specialized program introduces auxiliary variables and is not necessarily containing only simple statements (although there are only simple statements in this example). This is more like a real-world program compared to the programs only containing simple statements.

For easier reference of these program transformation approaches, we call the normal approach PTr; the approach interleaving partial evaluation actions PTr + PE; and the approach involving SNF updates PTr + SNF. Obviously, we can also interleave partial evaluation actions during symbolic execution phase, as well as involving SNF updates during generation phase, denoted as PTr + PE + SNF. This will achieve the most optimization.

We show the application of the program transformation and optimization techniques introduced before on some larger examples.

Example 13. *Specialize the following* SiJa *program using* PTr + PE + SNF.

```
public class OnLineShopping {
  boolean cpn;
  public int read() { /* read price of item */ }
  public int sum(int n) {
    int i = 1;
    int count = n;
    int tot = 0;
    while(i <= count) {
      int m = read();
      if(i >=3 && cpn) {
        tot = tot + m * 9 / 10;
        i++; }
      else {
        tot = tot + m;
        i++; }
    }
    return tot;
  }
}
```

Our purpose is to specialize the sum() method which consists of non-trivial constructs such as attributes, a conditional, loop, and method call. We ignore the postcondition ϕ and unnecessary formulas Γ and Δ. To ease the presentation, we do not mention partial evaluation and the update simplification steps all the time, but keep in mind that after each rule application the partial evaluation actions are performed and the updates within a sequential block are always simplified. Also, we just show the final sequents computed after sequent calculus rule application, partial evaluation and update simplification, but hide the intermediate results. As usual, sp_i denotes the program to be generated, and use_i denotes the used variable set. The symbolic execution rules used here are the rules involving SNF updates that are defined in Figure 4.10.

The return value tot is the only observable location, i.e., $obs = \{\mathtt{tot}\}$. The first phase starts symbolically executing method sum(). The first statements of the method declare and initialize variables. These statements are executed similar to assignments. Altogether the assignment rule is applied three times, we end up with

$$\begin{array}{c}
\Longrightarrow \{\mathtt{i} := 1 \| \mathtt{count} := \mathtt{n} \| \mathtt{tot} := 0\}[\,\mathtt{while(i <= n)\ldots} \between sp_3\,]@(\{\mathtt{tot}\}, use_3) \\
\hline
\Longrightarrow \{\mathtt{i} := 1 \| \mathtt{count} := \mathtt{n}\}[\,\mathtt{tot = 0;\ while(i <= n)\ldots} \between sp_2\,]@(\{\mathtt{tot}\}, use_2) \\
\hline
\Longrightarrow \{\mathtt{i} := 1\}[\,\mathtt{count = n;\ldots} \between sp_1\,]@(\{\mathtt{tot}\}, use_1) \\
\hline
\Longrightarrow [\,\mathtt{i = 1;\ldots} \between sp_0\,]@(\{\mathtt{tot}\}, use_0)
\end{array}$$

We use $\mathcal{U}_1^{snf}$ to denote the SNF update computed in this sequential block: $\mathcal{U}_1^{snf} = \{\texttt{i} := \texttt{1} \| \texttt{count} := \texttt{n} \| \texttt{tot} := \texttt{0}\}$.

The next statement to be symbolically executed is the while loop computing the total sum.

Instead of immediately applying the loop invariant rule, we unwind the loop once using the loopUnwind rule. Partial evaluation allows to simplify the guard `i <= n` and `i >= 3 && cpn` of the introduced conditional to `1 <= n` and `1 >= 3 && cpn` by applying constant propagation. Furthermore, the `then` branch is eliminated because the guard `1 >= 3 && cpn` can be evaluated to *false*. The result is as follows:

$$\Longrightarrow \mathcal{U}_1^{snf}[\ \texttt{if(1 <= n)\{int m = read(); tot = m; i = 2; while...\}} \between sp_3\ ]@(\{\texttt{tot}\}, use_3)$$

$$\Longrightarrow \mathcal{U}_1^{snf}[\ \texttt{if(1 <= n)\{... tot = 0 + m; i = 2; while...\}} \between sp_3\ ]@(\{\texttt{tot}\}, use_3)$$

$$\Longrightarrow \mathcal{U}_1^{snf}[\ \texttt{if(1 <= n)\{... if(1 >= 3 \&\& cpn)...; i = 1 + 1; while...\}} \between sp_3\ ]@(\{\texttt{tot}\}, use_3)$$

$$\Longrightarrow \mathcal{U}_1^{snf}[\ \texttt{if(i <= n)\{... if(i >= 3 \&\& cpn)...; i++; while...\}} \between sp_3\ ]@(\{\texttt{tot}\}, use_3)$$

$$\Longrightarrow \mathcal{U}_1^{snf}[\ \texttt{while(i <= n)...} \between sp_3\ ]@(\{\texttt{tot}\}, use_3)$$

Application of the ifElse rule creates two branches. The `else` branch contains no program so it is synthesized right away by applying the emptyBox rule. We symbolically execute the `then` branch by applying the assignment rule three times until we reach the while loop again. We use $\mathcal{U}_2^{snf}$ to denote the SNF update for the sequential block in the `then` branch until the while loop. $\mathcal{U}_2^{snf} = \{\texttt{m} := \texttt{res}_1 \| \texttt{tot} := \texttt{res}_1 \| \texttt{i} := \texttt{2}\}$. Here, in the update we use $\texttt{res}_1$ to denote the return value of `read()`. We decide to unwind the loop a second time. The symbolic execution follows then the same pattern as before until we reach the loop for a third time. Figure. 4.15(a) shows the relevant part of the symbolic execution tree of the second loop unwinding.

Instead of unwinding the loop once more, we apply the loopInvariant rule with *true* as the invariant. The rule creates three new goals. The goal for the `init` branch is not of importance for the specialization itself, hence, we ignore it in the following. The anonymizing update $\mathcal{V}_{mod}$ is also ignored.

The *used variables* set *use* of the `preserves` branch depends on the instantiation of the *use* set in the `use case` branch. To resolve the dependency we continue with the latter. In this case, the `use case` branch contains no program, so it is trivially synthesized by applying the emptyBox rule which results in _ as the specialized program and the only element `tot` in *obs* becomes the *use* set. Based on this, the *use* set of the `preserves` branch is the union of $\{\texttt{tot}\}$ and the locations used in the loop guard: $\{\texttt{tot}, \texttt{i}\}$. The program in the `preserves` branch is then symbolically executed by applying suitable rules until it is empty. This process is similar to that when executing the program in the `then` branch of the conditional generated by loopUnwind. $\mathcal{U}_3^{snf}$ denotes the SNF update $\{\texttt{m} := \texttt{res}_2 \| \texttt{tot} := \texttt{tot} + \texttt{res}_2 \| \texttt{i} := \texttt{3}\}$. $\mathcal{U}_4^{snf}$ denotes the SNF update $\{\texttt{m} := \texttt{res}_3\}$. The symbolic execution tree resulting from the application of the loop invariant rule is shown in Figure. 4.15(b).

$$\dfrac{\neg(1 \le \mathtt{n}) \Longrightarrow \mathcal{U}_1^{snf}[\between \_\,]@(\{\mathtt{tot}\},\{\mathtt{tot}\}) \qquad \dfrac{\begin{array}{c}\dfrac{\dfrac{\dfrac{\dfrac{2 \le \mathtt{n} \Longrightarrow \mathcal{U}_1^{snf}\mathcal{U}_2^{snf}\{\mathtt{m} := \mathtt{res}_2 \| \mathtt{tot} := \mathtt{tot} + \mathtt{res}_2 \| \mathtt{i} := 3\}[\,\mathtt{while(i <= n)}\ldots \between \mathit{sp}_6\,]@(\{\mathtt{tot}\},\mathit{use}_6)}{\cdots}}{1 \le \mathtt{n} \Longrightarrow \mathcal{U}_1^{snf}\mathcal{U}_2^{snf}[\,\mathtt{if(i <= n)}\ldots;\mathtt{while}\ldots \between \mathit{sp}_5\,]@(\{\mathtt{tot}\},\mathit{use}_5)}}{1 \le \mathtt{n} \Longrightarrow \mathcal{U}_1^{snf}\{\mathtt{m} := \mathtt{res}_1 \| \mathtt{tot} := \mathtt{res}_1 \| \mathtt{i} := 2\}[\,\mathtt{while(i <= n)}\ldots \between \mathit{sp}_5\,]@(\{\mathtt{tot}\},\mathit{use}_5)}}{\cdots}\end{array}}{1 \le \mathtt{n} \Longrightarrow \mathcal{U}_1^{snf}[\,\mathtt{int\ m = read();}\ldots \between \mathit{sp}_4\,]@(\{\mathtt{tot}\},\mathit{use}_4)}}{\Longrightarrow \mathcal{U}_1^{snf}[\,\mathtt{if(1 <= n)\{int\ m = read(); tot = m; i = i + 1; while}\ldots\} \between \mathit{sp}_3\,]@(\{\mathtt{tot}\},\mathit{use}_3)}$$

(a) Specialization of the `while` loop via unwinding

$$\dfrac{\neg(\mathtt{i} \le \mathtt{n}) \Longrightarrow [\,\between \_\,]@(\{\mathtt{tot}\},\{\mathtt{tot}\}) \qquad \dfrac{\dfrac{\begin{array}{c}\cdots\\ \ldots,\mathtt{cpn} \Longrightarrow \mathcal{U}_1^{snf}\ldots\mathcal{U}_4^{snf}[\ldots \between \mathit{sp}_9\,]@(\{\mathtt{tot},\mathtt{i}\},\mathit{use}_9)\end{array} \qquad \dfrac{\dfrac{\ldots \Longrightarrow \mathcal{U}_1^{snf}\ldots\mathcal{U}_4^{snf}\{\mathtt{tot} := \mathtt{tot} + \mathtt{m} \| \mathtt{i} := \mathtt{i} + 1\}[\,\between \mathit{sp}_{12}\,]@(\{\mathtt{tot},\mathtt{i}\},\mathit{use}_{12})}{\ldots \Longrightarrow \mathcal{U}_1^{snf}\ldots\mathcal{U}_4^{snf}\{\mathtt{tot} := \mathtt{tot} + \mathtt{m}\}[\,\mathtt{i}{+}{+};\ \between \mathit{sp}_{11}\,]@(\{\mathtt{tot},\mathtt{i}\},\mathit{use}_{11})}}{\ldots,\neg\mathtt{cpn} \Longrightarrow \mathcal{U}_1^{snf}\ldots\mathcal{U}_4^{snf}[\ldots \between \mathit{sp}_{10}\,]@(\{\mathtt{tot},\mathtt{i}\},\mathit{use}_{10})}}{\mathtt{i} \le \mathtt{n} \Longrightarrow \mathcal{U}_1^{snf}\mathcal{U}_2^{snf}\mathcal{U}_3^{snf}\{\mathtt{m} := \mathtt{res}_3\}[\,\mathtt{if(cpn)}\ldots \between \mathit{sp}_8\,]@(\{\mathtt{tot},\mathtt{i}\},\mathit{use}_8)}}{\mathtt{i} \le \mathtt{n} \Longrightarrow \mathcal{U}_1^{snf}\mathcal{U}_2^{snf}\mathcal{U}_3^{snf}[\,\mathtt{int}\ \ldots \between \mathit{sp}_7\,]@(\{\mathtt{tot},\mathtt{i}\},\mathit{use}_7)}}{2 \le \mathtt{n} \Longrightarrow \mathcal{U}_1^{snf}\mathcal{U}_2^{snf}\mathcal{U}_3^{snf}[\,\mathtt{while(i <= n)}\ldots \between \mathit{sp}_6\,]@(\{\mathtt{tot}\},\mathit{use}_6)}$$

(b) Specialization of the `while`-loop using the loop invariant rule

Figure 4.15: Specialization of the `while`-loop by different means.

After symbolic execution we enter the second phase of our approach in which the specialized program is generated. Recall that when applying the loopInvariant rule, the procedure of synthesizing the loop starts with the use case branch. In our example, we have already performed this step and could already determine the instantiation of the observable location set *obs* of the `preserves` branch.

We show how the loop body is synthesized in the `preserves` branch: applying the emptyBox rule instantiates the placeholders sp_{12} and use_{12} with _ and $\{\texttt{tot}, \texttt{i}\}$. Going backwards, the assignment rule tells us how to derive the instantiations for sp_{11}: $\texttt{i} = \texttt{i} + \texttt{1};$ and $use_{11} = \{\texttt{tot}, \texttt{i}\}$. The instantiations for sp_{10} and use_{10} can be derived as $\texttt{tot} = \texttt{tot} + \texttt{m}; \texttt{i} = \texttt{i} + \texttt{1};$ and $\{\texttt{tot}, \texttt{i}\}$. Before we can continue, the instantiations of sp_9 and use_9 need to be determined. Similar to the derivation of sp_{10} and use_{10}, applying the assignment rule a few times, we get sp_9: $\texttt{tot} = \texttt{tot} + \texttt{m} * \texttt{9}/\texttt{10}; \texttt{i} = \texttt{i} + \texttt{1};$ and $use_9 = \{\texttt{tot}, \texttt{i}\}$. We have now reached the node where the ifElse rule was previously applied. This rule allows us to derive sp_8 as

```
if (cpn) { tot = tot + m * 9 / 10;
  i = i + 1; }
else { tot = tot + m;
  i = i + 1; }
```

and $use_8 = \{\texttt{tot}, \texttt{i}, \texttt{cpn}\}$.

Applying suitable rules, we end up with the specialized program sp_6 as

```
while (i<=n) {
  int m = read();
  if (cpn) {
    tot = tot + m * 9 / 10;
    i = i + 1;
  }
  else {
    tot = tot + m;
    i = i + 1;
  }
}
```

and the used variable set $use_6 = \{\texttt{tot}, \texttt{i}, \texttt{cpn}\}$.

Following the symbolic execution tree backwards and applying the corresponding rules, we finally synthesize the specialized program for sum() as shown in Figure 4.16.

```
public int sum(int n) {
  int i;
  int tot;
  tot = 0;
  if (1 <= n) {
    tot = read();
    if (2 <= n) {
      tot = tot + read();
      i = 3;
      while(i <= n) {
        int m = read();
        if (cpn) {
          tot = tot + m * 9 / 10;
          i = i + 1;
        } else {
          tot = tot + m;
          i = i + 1;
        }
      }
    }
  }
  return tot;
}
```

Figure 4.16: The result of program transformation.

4.5 Implementation and Evaluation

We have a prototype implementation of the program transformation framework introduced in this chapter. It is named PE-KeY, which is an extension based on KeY including the following efforts:

- An information collector along with the symbolic execution of the source Java program. It keeps track of the observable variables and constructs the working stack that is used in the synthesize phase.
- An integrated partial evaluator which performs some simple partial evaluation operations such as constant propagation and dead code elimination. It is used in the symbolic execution phase.
- The extended calculus rules that are used to generate programs in the second phase. KeY's sequent calculus has around 1200 rules of which around 100-150 rules are used for sym-

bolic execution of programs. Around half of them have been implemented in the current version of PE-KeY, but a considerable effort is required to get a complete coverage.

- An update analyzer used to extract symbolic values of program variables from preceding updates to achieve a higher degree of specialization.

The current version of PE-KeY supports basic Java features such as assignment, comparison, conditional, loop, method call inlining, integer arithmetics. Array data structure and field access are also supported to some extent. Multi-threading and floating point arithmetics are not supported due to limitations of KeY.

We have tried PE-KeY with a set of example programs. Although in an early stage, the examples indicate the potential of PE-KeY once full Java is supported. For instance, the (simplified) formula

$$\mathtt{i} > \mathtt{j} \rightarrow [\mathtt{if(i>j)\ max\ =\ i;\ else\ max\ =\ j;}]\mathtt{POST}$$

leads to the following specialization of the conditional statement:

$$\mathtt{max} = \mathtt{i};$$

because of the precondition $\mathtt{i} > \mathtt{j}$ and thanks to the integrated first-order reasoning mechanism in PE-KeY. Here, POST is an unspecified predicate which can neither be proven nor disproved.

For the same reason,

$$\mathtt{i} \doteq 5 \rightarrow [\mathtt{i++;}]\mathtt{POST}$$

results in the specialized statement $\mathtt{i} = 6$.

In fact, the program can be specialized according to the given specification from a general implementation. Figure 4.17 shows a fragment of a bank account implementation. A bank account includes the current available balance and the credit line (normally fixed) that can be used when the balance is negative. Cash withdraw can be done by calling the `withdraw` method. If the withdraw amount does not exceed the available balance, the customer will get the cash without any extra service fee; if the available balance is less than the amount to be withdrawn, the customer will use the credit line to cover the difference with 5 extra cost; if the withdrawn amount could not be covered by both the available balance and the credit line, the withdraw does not succeed. In every case, the information of the new available balance will be printed (returned). This is a general implementation of the cash withdrawal process, but some banks (or ATMs) only allow cash withdrawal when the balance is above 0. In this case, the precondition of the `withdraw` method is restricted to $\mathtt{withdrawAmt} <= \mathtt{availableBal}$. Then, with help of PE-KeY, the implementation of method `withdraw` is specialized to:

$$\mathtt{return\ availableBal} - \mathtt{withdrawAmt};$$

```
public class BankAccount {
  int availableBal;
  int creditLn;

  BankAccount( int availableBal, int creditLn ) {
    this.availableBal = availableBal;
    this.creditLn = creditLn;
  }

  public int withdraw(int withdrawAmt) {
    if (withdrawAmt <= availableBal) {
      availableBal = availableBal - withdrawAmt;
      return availableBal;
    } else {
      if(withdrawAmt - availableBal <= creditLn) {
        availableBal = availableBal - withdrawAmt - 5;
        return availableBal;
      } else {
        return availableBal;
      }
    }
  }
  ...
}
```

Figure 4.17: Code fragment of bank account.

We applied our prototype partial evaluator also on some examples stemming from the JSpec test suite [SLC03]. One of them is concerned with the computation of the power of an arithmetic expression, as shown in Figure 4.18.

```
class Power extends Object{
 int exp;
 Binary op;
 int neutral;

 Power(int exp, Binary op,
       int neutral) {
   super();
   this.exp = exp;
   this.op = op;
   this.neutral = neutral;
 }

 int raise(int base) {
   int res = neutral;
   for (int i=0; i<exp; i++) {
     res = op.eval( base, res );
   }
   return res;
 }
}
```

```
class Binary extends Object {
  Binary() { super(); }
  int eval(int x, int y) {
    return this.eval(x, y);
  }
}

class Add extends Binary {
 Add() { super(); }
 int eval(int x, int y) {
    return x+y;
 }
}

class Mult extends Binary {
 Mult() { super(); }
 int eval(int x, int y) {
   return x*y;
 }
}
```

Figure 4.18: Source code of the Power example as found in the JSpec suite.

The interesting part is that the arithmetic expression is represented as an abstract syntax tree (AST) structure. The abstract class `Binary` is the superclass of the two concrete binary operators `Add` and `Mult` (the strategies). The `Power` class can be used to apply a `Binary` operator op and a `neutral` value for y times to a base value x, as illustrated by the following expression:

$$\texttt{power} = \texttt{new Power}(\texttt{y}, \texttt{new op}(), \texttt{neutral}).\texttt{raise}(\texttt{x})$$

The actual computation for concrete values is performed on the AST representation. To be more precise, the task was to specialize the program

$$\texttt{power} = \texttt{new Power}(\texttt{y}, \texttt{new Mult}(), 1).\texttt{raise}(\texttt{x});$$

The ac under the assumption that the value of y is constant and equal to 16.

As input formula for PE-KeY we use:

$$\begin{aligned}&\texttt{y} \doteq 16 \rightarrow\\ &\quad [\texttt{power} = \texttt{new Power}(\texttt{y}, \texttt{new Mult}(), 1).\texttt{raise}(\texttt{x});\ \between\ \texttt{sp}_{\texttt{res}}\ ]@(\mathit{obs}, \mathit{use})\texttt{POST}\end{aligned}$$

PE-KeY then executes the program symbolically and extracts the specialized program $\texttt{sp}_{\texttt{res}}$ as $\texttt{power} = (\ldots((\texttt{x} * \texttt{x}) * \texttt{x}) * \ldots) * \texttt{x};$ (or $\texttt{power} = \texttt{x}^{16}$). The achieved result is a simple int-typed expression without the intermediate creation of the abstract syntax tree and should provide a significantly better performance than executing the original program.

5 Information Flow Security

5.1 Introduction

Information flows more freely in modern society. For instance, internet can deliver your information anywhere in the world, to any place whether you intend to or not. The preservation of *confidentiality* becomes a growing concern. The confidentiality of information refers to *secrets*, or *privacy* when it is personal information.

Since nowadays software is used in many places moving information, it is important to preserve the confidentiality on the program level. Information enters a program on *sources* and exits on *sinks*. When a program runs, if an output in a sink depends on an input in a source, then there is an *information flow* from that source to that sink. If this flow of information is undesired, then an *information leak* has occurred. The traditional approaches of preserving confidential data by using access control do not apply here, because access control only checks restrictions on the release of information, but not its propagation. *Information flow control* [Den82] tracks the flow of information in programs. Since each program is written in a programming language with rigorous semantics, we can apply language-based techniques to analyze information flow occurring in a program, to *enforce* that the program satisfies a *security policy* of interest [SM03].

A *security policy* places restrictions on the permitted dependencies between sources and sinks. To specify the allowed dependencies, the sources and sinks are labeled with (partially) ordered *confidentiality levels* [Den76]. The most common example of the confidentiality level is `Low` (public) and `High` (secret). These are considered to be part of a security lattice, ordered as `{(Low, Low), (Low, High), (High, High)}`. If an output on a `Low` sink depends on input from a `High` source, an information leak occurs.

The baseline for the security policies is the notion of *non-interference* [Coh77, GM82]. Non-interference states that any two runs of a program with the same `Low` inputs will produce the same `Low` outputs, regardless of what `High` inputs are. In other words, an *observer* can derive the information from the `High` variables by using only the information from the `Low` variables. Figure 5.1 illustrates the non-interference policy.

Example 14. *Let* $\mathtt{l}_1$ *be a* `Low` *variable and* $\mathtt{h}_1, \mathtt{h}_2$ *be* `High` *variables in a program.*

- $\mathtt{l}_1 = \mathtt{h}_1;$
  This program violates the non-interference policy because $\mathtt{l}_1$ *is assigned the value of* $\mathtt{h}_1$.

- $\mathtt{l}_1 = \mathtt{h}_1 - \mathtt{h}_2;$
  The non-interference policy is violated although $\mathtt{l}_1$ *can not learn the values of* $\mathtt{h}_1$ *or* $\mathtt{h}_2$*, but the difference of them is leaked.*

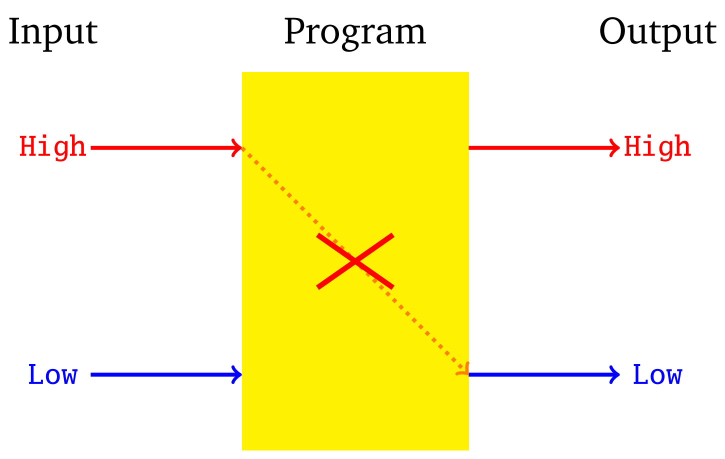

Figure 5.1: Non-interference.

In the above example, the (partial) information from the value of the `High` variables flows directly to the `Low` variables (*explicit* flow). It is also possible that information flows indirectly from `High` to `Low` variables (*implicit* flow), as shown in the following example:

Example 15. *Let* $\mathtt{l}_1, \mathtt{l}_2$ *be* `Low` *variables and* $\mathtt{h}_1, \mathtt{h}_2$ *be* `High` *variables in a program.*

- $\mathtt{if}(\mathtt{h}_1 > 0)\ \{\mathtt{l}_1 = 1;\}\ \mathtt{else}\ \{\mathtt{l}_1 = 0;\}$
  The value of $\mathtt{l}_1$ *leaks the information of whether* $\mathtt{h}_1$ *is greater than 0.*

- $\mathtt{l}_1 = 0; \mathtt{l}_2 = 0;\ \mathtt{if}(\mathtt{h}_1 > \mathtt{h}_2)\ \{\mathtt{l}_1 = 1;\}\ \mathtt{else}\ \{\mathtt{l}_2 = 1;\}$
  Partial information on the comparison of $\mathtt{h}_1$ *and* $\mathtt{h}_2$ *is leaked by observing either* $\mathtt{l}_1$ *or* $\mathtt{l}_2$ *has been set to 1.*

In Examples 14 and 15, `High` variables appear in the program by assigning to `Low` variables explicitly or determining the values of `Low` variables implicitly. However, such a use of `High` variables does not necessarily entail an information leak, as shown in Example 16.

Example 16. *Let* $\mathtt{l}_1, \mathtt{l}_2$ *be* `Low` *variables and* $\mathtt{h}_1, \mathtt{h}_2$ *be* `High` *variables in a program.*

- $\mathtt{l}_1 = \mathtt{h}_1; \mathtt{l}_1 = 1;$
  No information is leaked because after termination the value of $\mathtt{l}_1$ *is set to 1 although it was assigned to* h_1 *in the intermediate state.*

- $\mathtt{l}_1 = 0; \mathtt{l}_2 = 0;\ \mathtt{if}(\mathtt{h}_1 > \mathtt{h}_2)\ \{\mathtt{l}_1 = 1;\}\ \mathtt{else}\ \{\mathtt{l}_1 = 1;\}$
  The non-interference policy is valid because the value of $\mathtt{l}_1$ *is set to 1 no matter which value of* $\mathtt{h}_1$ *and* $\mathtt{h}_2$ *is greater. In fact, this program is equivalent to:* $\mathtt{l}_1 = 0; \mathtt{l}_2 = 0; \mathtt{l}_1 = 1;$.

The condition of non-interference requires `Low` outputs to be independent of `High` inputs. Devising an *enforcement mechanism* for this condition which is sound and permissive is an ongoing challenge. In practice, this condition is not always necessary. Many programs actually *intend* to leak some information about the `High` variables. Take the bank account for example. When a user logs in to an online banking session or an ATM machine, after a failure attempt, the correct password must not be displayed directly to the user. However, the correctness of the password can always be derived when the system allows you to log in or not, hence this partial information is allowed to leak. This leads to the notion of information *declassification* [SS05] that certain parts of `High` variables can be declassified. For example, the variable `correctPassword` should not flow into a `Low` variable, but the result of the operation `correctPassword == providedPassword` is allowed to flow, thus declassified to `Low`.

In information flow control, a security policy is accompanied by a permissive *enforcement mechanism*, proven sound with respect to the given security policy. On running a program, if the enforcement reports a positive result, then the soundness proof implies that the program satisfies the policy. Several approaches to enforce the security policies rely on the semantics of the *language constructs*.

One way is to use *static analysis* that analyzes the program before executing it. These often take the form of a *security type system* [VIS96, HS06] which, by tracking the confidentiality level of information contained in variables and program context, (over-)approximates information flows occurring in (an over-approximation of) the control flow paths the program can take. It is possible to guarantee the nonexistence of leaky control flow paths. One advantage of static enforcement is that the policy is enforced before running the program. It thus avoids the runtime overhead. Another is the ability to reason about all control flow paths. It can ensure that `Low` outputs observers cannot learn about `High` inputs by inferring which control flow path was not taken. Since analysis is performed before the program is run, the enforcement has no access to runtime information. A static enforcement cannot permissively enforce programs using highly dynamic language constructs, because a large control flow branching occurs at these control points and the coarse approximations have to be made.

Another way to enforce security policies is using *dynamic analysis*, or more precisely *security monitors* [Vol99, AS09] in this setting. At run time, input data is labeled with the confidentiality level that propagates through the channels. When the monitor detects an output of data containing a `High` label on a `Low` sink, the monitor prevents the leak by blocking the program. Although this blocking of the program can also leak information, an advantage of dynamic analysis is the ability to treat highly dynamic language constructs in a permissive manner. The dynamic enforcement has runtime overhead, and cannot guarantee the absence of leaks for the control flow paths that are not taken.

Research of information flow control addresses many other aspects of security policies and enforcement. The reader is referred to [SM03] for a detailed survey.

5.2 Enforcing Information Flow Security by Program Transformation

In this section, we show how to apply our program transformation framework to enforce information flow security for SiJa programs. We concentrate on the non-interference policy.

Recall the weak bisimulation modality $[\ \mathtt{p}\ \between\ \mathtt{q}\ ]@(obs, use)$ defined in Section 4.2. In a program specialization setting, as discussed in Chapter 4, q is the same programming language as p, and *obs* are normally the return variables and the variables used in the postcondition. Running program q is equivalent to running program p except that q is more optimized.

In fact, q can also be viewed as a *dependency flow* of *obs*. This is because the extended sequent calculus rules for the weak bisimulation modality (involving updates or not) only allow to generate statements that will interfere with the evaluation of the *obs* variables in the final state. For instance, the assignment rule will only generate an assignment in which the assigned location belongs to *use*, so this assignment has an interference with *obs* (see Definition 19). This observation gives us the opportunity to apply our program transformation approaches to a information flow security setting.

Instead of generating a meaningful program q that is equivalent to p in a real program execution by fixing *obs* as its return variables, we can choose *obs* freely and generate a dependency flow q of *obs*, which may not be as meaningful as the original program p. This will not affect the soundness of our program transformation framework, because *obs* is not required to be fixed as the return variables in the related definitions and proofs. So the generated program q is also weakly bisimilar to p with respect to an arbitrary choice of obs. We can choose obs as the variables with confidentiality level Low. By doing this, after program transformation, we get a dependency flow q of Low variables. If q does not contain High variables, the non-interference policy is enforced. If High variables occur in q, there is *possibly* an information leak. This is formalized in Lemma 11.

Lemma 11. *Given SiJa programs* p, q, *a set of* High *variables* H *and a set of* Low *variables* L *such that* $\mathtt{p} \approx_L \mathtt{q}$. *If for all* $\mathtt{h} \in H$, $\mathtt{h} \notin pv(\mathtt{q})$, *then the non-interference policy for program* p *is enforced.*

Example 17. *Let* $\mathtt{l}_1, \mathtt{l}_2$ *be* Low *variables and* $\mathtt{h}_1, \mathtt{h}_2$ *be* High *variables. Consider the following SiJa programs:*

(i) $\mathtt{l}_1 = \mathtt{h}_1; \mathtt{l}_1 = 1;$
Fixing obs as $\{\mathtt{l}_1\}$, *program transformation results in:* $\mathtt{l}_1 = 1$;. *According to Lemma 11, non-interference is enforced.*

(ii) $\mathtt{l}_1 = \mathtt{h}_1 - \mathtt{h}_2;$
Fixing obs as $\{\mathtt{l}_1\}$, *program transformation ends with the same program. Non-interference cannot be determined by Lemma 11. By inspecting the generated program, we find an explicit information leak.*

(iii) $\mathtt{l}_1 = 0; \mathtt{l}_2 = 0;$ $\mathtt{if}(\mathtt{h}_1 > \mathtt{h}_2)\ \{\mathtt{l}_1 = 1;\}\ \mathtt{else}\ \{\mathtt{l}_2 = 1;\}$
Fixing obs as $\{\mathtt{l}_1, \mathtt{l}_2\}$*, the specialized program is:* $\mathtt{if}(\mathtt{h}_1 > \mathtt{h}_2)\ \{\mathtt{l}_1 = 1;\}\ \mathtt{else}\ \{\mathtt{l}_2 = 1;\}$.
Non-interference cannot be determined by Lemma 11. By inspecting the generated program, we find an implicit information leak.

(iv) $\mathtt{l}_1 = 0; \mathtt{l}_2 = 0;$ $\mathtt{if}(\mathtt{h}_1 > \mathtt{h}_2)\ \{\mathtt{l}_1 = 1;\}\ \mathtt{else}\ \{\mathtt{l}_1 = 1;\}$
Fixing obs as $\{\mathtt{l}_1, \mathtt{l}_2\}$*, the specialized program is:* $\mathtt{l}_2 = 0; \mathtt{if}(\mathtt{h}_1 > \mathtt{h}_2)\ \{\mathtt{l}_1 = 1;\}\ \mathtt{else}\ \{\mathtt{l}_1 = 1;\}$.
Non-interference cannot be determined by Lemma 11. By inspecting the generated program, non-interference is enforced.

The above example shows the application of program transformation to enforce non-interference policy. The first example can be determined directly by Lemma 11. In the other examples, the information leak has to be checked by other enforcement approaches on the generated program. The generated program is optimized with respect to the `Low` variables, so it is easier to check. In fact, Lemma 11 gives no conclusion when some `High` variables occur in the generated program.

If we can give a suitable notion of *explicit* flow and *implicit* flow, then Lemma 11 can be strengthened. A first attempt is the following definition:

Definition 23 (Explicit and implicit flow — first attempt). *Given **SiJa** programs* p, q*, a set of* `High` *variables H and a set of* `Low` *variables L such that* $\mathtt{p} \approx_L \mathtt{q}$.

- *If there exists* $\mathtt{h} \in H$ *and some non-boolean expression exp of program* q *such that* $\mathtt{h} \in pv(exp)$*, then there is an* explicit *flow in program* p.
- *If there exists* $\mathtt{h} \in H$ *and some boolean expression* exp_B *of program* q *such that* $\mathtt{h} \in pv(exp_B)$*, then there is an* implicit *flow in program* p.

By this definition, we can conclude that in Example 17, the second program has an explicit information flow leak, and the third program has an implicit information flow leak.

However, Definition 23 is not accurate enough.

Example 18. *Let* `l` *be* `Low` *and* `h` *be* `High`*. Consider the non-interference policy of a **SiJa** program:*

```
t = h;
h = l;
l = t;
t = l;
l = h;
h = t;
```

This program swaps the values of `l` and `h` twice. In the end, `l` is assigned its original value and cannot learn any information of `h`, so the non-interference policy is enforced. However, fixing *obs* as {l}, if we do program transformation using the normal approach PTr introduced in Section 4.2, the resulting program is: h = l; l = h;, which has an explicit information leak. So the above definition is not valid if we generate a program using PTr approach. We redo program transformation involving partial evaluation actions and SNF updates (PTr + PE + SNF). This time we end up with program: $l^0 = l; l = l^0$; that leaks no information.

In order to generate a program for the purpose of information flow security enforcement, it is better to use the most optimized approach PTr + PE + SNF introduced in Section 4.4. This is because PTr + PE + SNF takes into account the SNF update in a sequential block to generate the statements with up-to-date information without showing the intermediate assignments. Intermediate assignments are exactly the reason why Definition 23 would fail.

Based on this observation, on the second try, we give a more precise definition of explicit and implicit flow:

Definition 24 (Explicit and implicit flow – second attempt). *Given SiJa programs* p*, the set of* High *variables H and the set of* Low *variables L; program* q *is generated by* PTr + PE + SNF *approach, and* $p \approx_L q$.

- *If there exists* $h \in H$ *and some non-boolean expression exp of program* q *such that* $h \in pv(exp)$*, then there is an* explicit *flow in program* p.
- *If there exists* $h \in H$ *and some boolean expression* exp_B *of program* q *such that* $h \in pv(exp_B)$*, then there is an* implicit *flow in program* p.

This definition is based on PTr + PE + SNF approach thus the update in a sequential block helps to generate a more optimized program. However, so far we have not simplified the different SNF updates for different sequential blocks along an execution path and made use of this for program generation, but the non-interference policy is enforced in all execution paths as studied in the static enforcement approaches. This results in the inaccuracy of Definition 24. And the issue is, as in Definition 23, the generated intermediate assignments, this time in different sequential blocks.

Example 19. *Let* l_1, l_2 *be* Low *and* h *be* High*. Consider the non-interference policy for SiJa program:*

```
h = l₁ + l₂;
if (h > 0) {
  l₁ = h + 1;
} else {
  l₂ = h + 2;
}
```

Fixing *obs* as $\{\texttt{l}_1, \texttt{l}_2\}$, $\mathsf{PTr} + \mathsf{PE} + \mathsf{SNF}$ will generate an almost identical program without much specialization. According to Definition 24, it has both an explicit and implicit information flow leak. However, after the real execution of this program, either $\texttt{l}_1$ is set to $\texttt{l}_1 + \texttt{l}_2 + 1$, or $\texttt{l}_2$ is set to $\texttt{l}_1 + \texttt{l}_2 + 2$ depending on the comparison of $\texttt{l}_1$ and $\texttt{l}_2$. In either case, this program is secure. The reason is already discussed above. To avoid this issue, one possible way is to simplify the different SNF updates for different sequential blocks along an execution path, which will lead to another category of sequent calculus rules. This is not a trivial extension.

So far we are focusing on the generated *program* to find a suitable notion of information leak. In fact, along with program generation, we also obtain the *used variable set*, denoted as use_0 here. When program generation is finished, according to Lemma 4 and Definition 19, use_0 are the observable locations in the initial state and each variable that belongs to use_0 will interfere with *obs* in the final state. In other words, in the information flow security setting, every input variable that belongs to use_0 will interfere with output `Low` variables. According to the definition of non-interference, we only need to guarantee that `High` variables do not occur in use_0. If so, the non-interference policy can be enforced. Then we have the following theorem.

Theorem 4 (Non-interference Enforcement). *Given a* ***SiJa*** *program* `p`, *a set of* `High` *variables* H *and a set of* `Low` *variables* L*; program* `q` *and used variable set* use_0 *is generated, and* $\texttt{p} \approx_L \texttt{q}$. *The non-interference policy is enforced if for all* $\texttt{h} \in H$, $\texttt{h} \notin use_0$.

Proof. Direct result of Lemma 4, Def. 19 and the notion of non-interference. □

We show that now Example 19 works properly. After program transformation, we achieve the used variable set as $\{\texttt{l}_1, \texttt{l}_2\}$. Since no `High` variables are involved, the non-interference policy is enforced. By inspecting the final used variable set use_0, we can check information flow security quickly. If no `High` variables occur in use_0, then non-interference policy is enforced; otherwise, we can also use other existing approaches to check the generated program, which is still better than checking the original program.

Example 20. *Let* `l` *be* `Low` *variables and* `h` *be* `High` *variables in a program. We discuss whether the standard security policy, as stated in the introduction, holds for some example programs:*

(i) $\texttt{h} = 0; \texttt{l} = \texttt{h};$
Fixing obs as $\{\texttt{l}\}$, *program transformation results in:* $use_0 = \{\emptyset\}$. *According to Theorem 4, non-interference is enforced.*

(ii) $\texttt{l} = \texttt{h}; \texttt{l} = \texttt{l} - \texttt{h};$
Fixing obs as $\{\texttt{l}\}$, *program transformation by* $\mathsf{PTr} + \mathsf{PE} + \mathsf{SNF}$ *approach results in:* $use_0 = \{\emptyset\};$. *According to Theorem 4, non-interference is enforced.*

(iii) $\texttt{if}(\texttt{h} > 0)\ \{\texttt{h} = \texttt{l}; \texttt{l} = \texttt{h};\}$
Fixing obs as $\{\texttt{l}\}$, *program transformation by* $\mathsf{PTr} + \mathsf{PE} + \mathsf{SNF}$ *approach results in:* $use_0 =$

$\{\mathtt{h}\}$ and program: $\mathtt{if(h > 0)\ \{l = l;\}}$. Non-interference cannot be determined by Theorem 4. By inspecting the generated program, non-interference is enforced.

(iv) $\mathtt{if(h > 0)\ \{l = 1;\}\ else\ \{l = 2;\}\ l = 0;}$
Fixing obs as $\{\mathtt{l}\}$, program transformation by PTr + PE + SNF *approach results in: $use_0 = \{\mathtt{h}\}$ and program: $\mathtt{if(h > 0)\ \{l = 0;\}\ else\ \{l = 0;\}}$. Non-interference cannot be determined by Theorem 4. By inspecting the generated program, non-interference is enforced.*

Example 20 shows that Theorem 4 is still not precise enough to classify non-interference policies for some cases. For (*iii*), `l` is assigned to itself in the `then` branch. If we ignore this self-assignment, the final result is unchanged. For (*iv*), we have the identical program $\mathtt{l = 0;}$ in both the `then` and `else` branches. In this case, the conditional does not affect the result, so it can be safely ignored.

To achieve a more precise result, we need some extended sequent rules tailored to information flow analysis, as shown in Figure 5.2:

assignNotSelf

$$\frac{\Gamma \Longrightarrow \mathcal{U}_1^{snf}[\ \omega \between \overline{\omega}\ ]@(obs, use)\phi, \Delta}{\left(\begin{array}{ll} \Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{l = r;}\omega \between \mathtt{l = r_1;}\overline{\omega}\ ]@(obs, use - \{\mathtt{l}\} \cup \{\mathtt{r}\})\phi, \Delta & \text{if } \mathtt{l} \in use \wedge \mathtt{r_1} \neq \mathtt{l} \\ \Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{l = r;}\omega \between \overline{\omega}\ ]@(obs, use)\phi, \Delta & \text{otherwise} \end{array}\right)}$$

(where $\mathcal{U}_1^{snf} = \mathcal{U}_1^{snf_a}\{\ldots \| \mathtt{l} := \mathtt{r_1}\}$ is the SNF of $\mathcal{U}^{snf}\{\mathtt{l} := \mathtt{r}\}$)

$$\text{ifElseUnify}\ \frac{\begin{array}{l}\Gamma, \mathcal{U}^{snf}\mathtt{b} \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{p;}\omega \between \overline{\mathtt{p;}\omega}\ ]@(obs, use_{\mathtt{p};\omega})\phi, \Delta \\ \Gamma, \mathcal{U}^{snf}\neg\mathtt{b} \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{q;}\omega \between \overline{\mathtt{q;}\omega}\ ]@(obs, use_{\mathtt{q};\omega})\phi, \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{if\ (b)\ \{p\}\ else\ \{q\};}\omega \between \overline{\mathtt{p;}\omega}\ ]@(obs, use_{\mathtt{p};\omega})\phi, \Delta}$$

(with `b` boolean variable, $\overline{\mathtt{p;}\omega} \approx_{obs} \overline{\mathtt{q;}\omega}$, and $use_{\mathtt{p};\omega} = use_{\mathtt{q};\omega}$)

loopInvNoBody

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}^{snf} inv, \Delta \\ \Gamma, \mathcal{U}^{snf}\mathcal{V}_{mod}(\mathtt{b} \wedge inv) \Longrightarrow \mathcal{U}^{snf}\mathcal{V}_{mod} \\ \qquad\qquad\qquad\qquad [\ \mathtt{p} \between \overline{\mathtt{p}}\ ]@(use_1 \cup \{\mathtt{b}\}, use_2) inv, \Delta \\ \Gamma, \mathcal{U}^{snf}\mathcal{V}_{mod}(\neg\mathtt{b} \wedge inv) \Longrightarrow \mathcal{U}^{snf}\mathcal{V}_{mod}[\ \omega \between \overline{\omega}\ ]@(obs, use_1)\phi, \Delta\end{array}}{\left(\begin{array}{ll} \Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{while(b)\{p\}}\,\omega \between \overline{\omega}\ ]@(obs, use_1)\phi, \Delta & \text{if } use_1 = \emptyset \\ \Gamma \Longrightarrow \mathcal{U}^{snf}[\ \mathtt{while(b)\{p\}}\,\omega \between \mathtt{while(b)\{\overline{p}\}}\,\overline{\omega}\ ]@(obs, use_1 \cup use_2 \cup \{\mathtt{b}\})\phi, \Delta & \text{otherwise} \end{array}\right)}$$

Figure 5.2: Some extended sequent calculus rules tailored to information flow analysis.

The assignNotSelf rule avoids the generation of self assignments $\mathtt{l = l;}$. The ifElseUnify rule checks whether the `then` branch and `else` branch have the same effect, if so, we do not generate a conditional block. The loopInvNoBody rule avoids the generation of a loop body, if the used variable set obtained in the continuation of the loop is $\emptyset$. Because in this case, the loop does not affect the values of the observable locations at all.

Now programs (*iii*) and (*iv*) in Example 20 can be classified properly. For (*iii*), according to assignNotSelf, we do not generate any program in the `then` branch, then apply ifElseUnify rule (both branches are empty), we obtain the empty program, with used variable set $use = \{\mathtt{l}\}$. According to Theorem 4, non-interference is enforced. For (*iv*), we generate the program $\mathtt{l} = 0;$ and $use = \emptyset$, non-interference is enforced.

Example 21. *Consider the following program with loop invariant* $\mathtt{l} > 0$ *and post condition* $\mathtt{l} \doteq 2$. *Let* `l` *be* `Low` *and* `h` *be* `High`.

```
l = 1;
while (h > 0) {
  l ++;
  h --;
}
if (l > 0) {
  l = 2;
}
```

After symbolic execution of the loop we have three branches. In the branch that continues after the loop, we encounter a conditional. With the loop invariant we can infer that the guard holds, so we only execute the then *branch with* $\mathtt{l} = 2;$. *Every open goal is closeable, so the program is proven. We start to analyze information flow security with* $obs = \{\mathtt{l}\}$. *In the first step, the statement* $\mathtt{l} = 2;$ *is generated empty used variable set. According to* loopInvNoBody, *we do not generate loop body code. Continuing with* $\mathtt{l} = 1;$, *we obtain the program* $\mathtt{l} = 2;$ *and an empty used variable set. According to Theorem 4, non-interference is enforced.*

Remark. *We can perform the program transformation without suitable loop invariants (just use true), as discussed previously (e.g, in Section 4.2). This achieves a higher degree of automation, which is desirable in the context of program specialization. However, proper loop invariants will increase the precision of the information flow analysis. Without the loop invariant* $\mathtt{l} > 0$ *in Example 21, we have to generate the conditional as well as the loop body, and then we cannot classify this program.*

Because the program transformation process employs first-order reasoning and partial evaluation in the symbolic execution phase, as well as using updates during program generation, we achieve a more precise information flow analysis than security type systems.

6 Deductive Compilation

6.1 Introduction

Can you trust your compiler? In general, compilers should preserve the semantical behavior of the source program and compiled program (*bytecode*). Complicated symbolic transformations are performed during compilation, especially for the case of optimizing compilers. Compilers may be buggy, resulting in a crash at compile-time, or even introducing errors to the correct source program. Those errors introduced by compilers are notoriously difficult to expose and track down. Nowadays, most effort of formal verification of programs is applied to the source code level. However, a buggy compiler may invalidate the correctness properties that have been formally verified for source code. We also need to guarantee the correctness of bytecode. The widely used technique used for this purpose is *compiler verification*, that proves the correctness of compilers.

Compiler verification has been a research topic for more than 40 years [MP67, MW72]. Since then, many proofs have been conducted, ranging from single-pass compilers for toy languages to sophisticated code optimizations [Dav03]. Recently, the *CompCert* project [Ler06, Ler09a, Ler09b] has been the most successful story in compiler verification. In that project, a complete compilation tool chain has been verified from a subset of C source code to PowerPC assembly language in Coq. CompCert focuses on low-level details and language features such as memory layout, register allocation and instruction selection. As part of the *Verisoft* project, a nonoptimizing compiler from C0, a subset of C, directly to DLX assembly has been verified in Isabelle/HOL [Lei08]. Like CompCert, it focuses on low-level details and proves a weak simulation theorem for sequential executions. The paper [Loc10] presents a rigorous formalization (in the proof assistant Isabelle/HOL) of concurrent Java source and byte code together with an executable compiler and its correctness proof.

Previous works have shown that compiler verification is an expensive task. In this chapter, we present our approach to guarantee the correctness of bytecode. Instead of verifying a compiler, we generate bytecode step by step using the program transformation techniques introduced in Chapter 4. The soundness of the extended sequent calculus rules entails the correctness of the generated program. No further verification of bytecode is needed.

6.2 Sequent Calculus for Bytecode Generation

The weak bisimulation modality $[\ \mathtt{p} \between \mathtt{q}\ ]@(obs, use)$ defined in Section 4.2 is the core concept for program transformation and information flow security. For both scenarios, $\mathtt{q}$ is the program in the same language as p. In fact, it is not necessary to restrict p and q to the same programming language. Choosing different languages for p and q will result in other applications of program transformation. For instance, fixing p as the Java language and q as the C language will result in a Java-to-C *translator*. To ensure the soundness of this translation, the correctness of the corresponding weak bisimulation modalities and accompanied extended sequent calculus rules need to be proven. Here we focus on the transformation from Java source code (or SiJa to be precise) to Java bytecode, which works as a Java (SiJa) compiler. The soundness of compilation is entailed by the sound bisimulation modality and sequent calculus rules.

We target the version of Java bytecode that can be executed by a *Java Virtual Machine (JVM)* [LY97]. The Java Virtual Machine is a conventional stack-based abstract machine. Most instructions pop their arguments off the stack, and push back their results on the stack. A set of *registers* (also called *local variables*) is provided. They can be accessed via a *load* instruction that pushes the value of a given register on the stack, and a *store* instruction that stores the top of the stack in the given register. Most Java compilers use registers to store the values of source-level local variables and method parameters, and the stack to hold temporary results during evaluation of expressions. Both the stack and the registers are preserved across method calls. Control is handled by a variety of branch instructions: unconditional branch (`goto`), conditional branches (e.g., `ifeq`), and multiway branches (corresponding to `switch`). In the JVM, most instructions are typed. For instance, the `iadd` instruction (integer addition) requires that the stack initially contains at least two elements and that these two elements are of type `int`; it then pushes back a result of type `int`. Table 6.1 shows some commonly used instructions with descriptions, a complete list can be found in [LY97].

Example 22. *Consider the following Java bytecode:*

-
  ```
  iload_0 istore_1
  ```
  Reads the value of variable 0 and stores it to variable 1.

-
  ```
  iload_0 iload_1 iadd
  ```
  Adds the int values of variables 0 and 1.

-
  ```
  iload_0 iload_1 if_icmplt lbelse
  iload_0 istore_2 goto lbelse
  lbelse:  iload_1 istore_2
  ```
  Compares the values of variable 0 and variable 1, if variable 0 is less than variable 1, then goto label `lbelse` *and write the value of variable 1 to variable 2; otherwise continue and set variable 2 with the value of variable 0. This program computes the max of 2 variables.*

Now we define the bisimulation modality for different programming languages. In Section 4.1, we have defined observable equivalence, observable locations, and weak bisimulation

Mnemonic	Other bytes	Description
`iload`	`1:index`	load an *int* value from a local variable #index
`iload_0`		load an *int* value from local variable 0
`istore`	`1:index`	store an *int* value into variable #index
`istore_0`		store an *int* value into variable 0
`iconst_0`		load the int value 0 onto the stack
`aload`	`1:index`	load a reference onto the stack from a local variable #index
`aload_0`		load a reference onto the stack from local variable 0
`astore`	`1:index`	store a reference into a local variable #index
`astore_0`		store a reference into local variable 0
`bipush`	`1:byte`	push a byte onto the stack as an integer value
`getfield`	`2: index1, index2`	get a field value of an object objectref, where the field is identified by field reference in the constant pool index
`putfield`	`2: indexbyte1, indexbyte2`	set field to value in an object objectref, where the field is identified by a field reference index in constant pool
`iadd`		add two ints
`isub`		subtract two ints
`imul`		multiply two ints
`idiv`		divide two ints
`iinc`	`2: index, const`	increment local variable #index by signed byte const
`ifeq`	`2: branchbyte1, branchbyte2`	if value is 0, branch to instruction at branchoffset
`ifne`	`2: branchbyte1, branchbyte2`	if value is not 0, branch to instruction at branchoffset
`ifge`	`2: branchbyte1, branchbyte2`	if value is no less than 0, branch to instruction at branchoffset
`ifgt`	`2: branchbyte1, branchbyte2`	if value is greater than 0, branch to instruction at branchoffset
`ifle`	`2: branchbyte1, branchbyte2`	if value is no greater than 0, branch to instruction at branchoffset
`iflt`	`2: branchbyte1, branchbyte2`	if value is less than 0, branch to instruction at branchoffset
`ifnull`	`2: branchbyte1, branchbyte2`	if value is null, branch to instruction at branchoffset
`ifnonnull`	`2: branchbyte1, branchbyte2`	if value is not null, branch to instruction at branchoffset
`if_icmpeq`	`2: branchbyte1, branchbyte2`	if ints are equal, branch to instruction at branchoffset
`if_icmpne`	`2: branchbyte1, branchbyte2`	if ints are not equal, branch to instruction at branchoffset
`if_icmpge`	`2: branchbyte1, branchbyte2`	if value1 is no less than value2, branch to instruction at branchoffset
`if_icmpgt`	`2: branchbyte1, branchbyte2`	if value1 is greater than value2, branch to instruction at branchoffset
`if_icmple`	`2: branchbyte1, branchbyte2`	if value1 is no greater than value2, branch to instruction at branchoffset
`if_icmplt`	`2: branchbyte1, branchbyte2`	if value1 is less than value2, branch to instruction at branchoffset
`goto`	`2: branchbyte1, branchbyte2`	goes to another instruction at branchoffset (signed short constructed from unsigned bytes branchbyte1 « 8 + branchbyte2)
`return`		return void from method
`ireturn`		return an integer from a method

Table 6.1: A collection of Java bytecode instructions.

for programs (Definitions 13-17). All the definitions there are on the semantic level. The semantics of Java bytecode is normally defined as an operational semantics in the form of an abstract machine (JVM). We ignore the technical details here, but show the relation of the semantics of a SiJa program and the semantics of Java bytecode. For the formal definition of Java bytecode semantics, readers can refer to e.g., [LY97, FM03].

Since the JVM is a stack-based abstract machine, execution of Java bytecode is a sequence of stack operations on the JVM. A *state* in Java bytecode is defined as a snapshot of the status of the registers (variables) and the stack. We define a mapping function ξ that relates the universe of SiJa source code and Java bytecode.

Definition 25 (Mapping function). *For a SiJa program, St is the set of statements, S is a set of states,* PV *is a set of program variables. And for Java bytecode,* $\mathtt{Inst}$ *is the set of instructions, S_B is a set of states,* PV *is a set of variables. A mapping function ξ maps:*

(i) *every* $\mathtt{pv} \in \mathsf{PV}$ *to a distinct* $\mathtt{pv_B} \in \mathsf{PV_B}$. $\xi(\mathtt{pv}) = \mathtt{pv_B}$.

(ii) *every* $s \in S$ *to an* $s_B \in S_B$. $\xi(s) = s_B$.

(iii) *every* $\mathtt{st} \in \mathtt{St}$ *to a sequence of instructions:* $\mathtt{inst_1} \ldots \mathtt{inst_n}$ *where for* $0 \leq i \leq n$ $\mathtt{inst_i} \in \mathtt{Inst}$ *and* $\xi(\mathtt{st}) = \mathtt{inst_1} \cdots \mathtt{inst_n}$.

ξ^{-1} *is the* inverse *of* ξ.

Figure 6.1 shows some SiJa statements and Java bytecode related by the mapping function ξ. We also maintain a *program counter pc* (initially 0) to indicate the label of bytecode instructions, and pc_i has the value of $pc + i$.

SiJa statement	Java bytecode

`l=r`	`iload_`ξ`(r)`
	`istore_`ξ`(l)`

$\mathtt{p_1;p_2}$	$\xi(\mathtt{p_1})$ $\xi(\mathtt{p_2})$
`if(b) {p} else {q}`	`iload_`ξ`(b)` `ifeq` pc_1 ξ`(p)` `goto` pc_2 pc_1: ξ`(q)` pc_2: _
`while(b) {p}`	pc_1: `iload_`ξ`(b)` `ifeq` pc_2 ξ`(p)` `goto` pc_1 pc_2: _

Figure 6.1: Mapping of SiJa programs to Java bytecode.

As a property of the mapping function ξ, the following lemma gives the relation of the semantics of SiJa programs to the semantics of Java bytecode. We assume the Java bytecode is evaluated by a evaluation function *valB*, logic structure D_B and a state s. The actual representation of D_B is not of importance.

Lemma 12. *Given the evaluation function val, the first-order structure D and a state $s \in S$ of SiJa program* $\mathtt{p}$*, and the corresponding evaluation function valB and logic structure D_B of Java bytecode* $\mathtt{q}$*. If $\xi(\mathtt{p}) = \mathtt{q}$, then $val_{D,s}(\mathtt{p}) = valB_{D_B,\xi(s)}(\mathtt{q})$.*

Lemma 12 shows that to evaluate the Java bytecode, we can evaluate its ξ^{-1}-mapped SiJa program. This gives us an opportunity to define the weak bisimulation modality for a SiJa program and Java bytecode by adding a mapping function ξ to Definitions. 13-17 in Section 4.1. For example, the definition of weak bisimulation for SiJa program and Java bytecode is given below. The other definitions are analogous.

Definition 26 (Weak bisimulation for SiJa program and Java bytecode)**.** *Let $\mathtt{p}_1, \mathtt{p}_2$ be two SiJa programs,* $\mathtt{q}$ *is Java bytecode, and ξ is a mapping function such that $\xi(\mathtt{p}_2) = \mathtt{q}$. Assume obs and obs′ are observable locations, and $\approx$ is a weak bisimulation relation for states. Then $\approx$ is a* weak bisimulation *for a SiJa program $\mathtt{p}_1$ and Java bytecode $\mathtt{q}$, written $\mathtt{p}_1 \approx_{obs} \mathtt{q}$, if for the sequence of state transitions:*

$$s_1 \xrightarrow{\mathtt{p_1}} s_1' \equiv s_1^0 \xrightarrow{\mathtt{sSt}_1^0} s_1^1 \xrightarrow{\mathtt{sSt}_1^1} \dots \xrightarrow{\mathtt{sSt}_1^{n-1}} s_1^n \xrightarrow{\mathtt{sSt}_1^n} s_1^{n+1}, \text{ with } s_1 = s_1^0,\ s_1' = s_1^{n+1},$$

$$s_2 \xrightarrow{\mathtt{p_2}} s_2' \equiv s_2^0 \xrightarrow{\mathtt{sSt}_2^0} s_2^1 \xrightarrow{\mathtt{sSt}_2^1} \dots \xrightarrow{\mathtt{sSt}_2^{m-1}} s_1^m \xrightarrow{\mathtt{sSt}_2^m} s_2^{m+1}, \text{ with } s_2 = s_2^0,\ s_2' = s_2^{m+1},$$

(i) $s_2' \approx_{obs} s_1'$; (ii) for each state s_1^i there exists a state s_2^j such that $s_1^i \approx_{obs'} s_2^j$ for some obs′; (iii) for each state s_2^j there exists a state s_1^i such that $s_2^j \approx_{obs'} s_1^i$ for some obs′, where $0 \le i \le n$ and $0 \le j \le m$.

The weak bisimulation modality for a SiJa program and Java bytecode can be defined similarly to that for SiJa programs only (Definition 18 and 20).

Definition 27 (Weak bisimulation modality for SiJa program and Java bytecode—syntax)**.** *The* bisimulation modality *$[\ \mathtt{p} \between \mathtt{q}\ ]@(obs, use)$ is a modal operator providing compartments for a SiJa program* $\mathtt{p}$*, Java bytecode* $\mathtt{q}$ *and location sets obs and use. We extend our definition of formulas: Let ϕ be a SiJa-DL formula and* $\mathtt{p}$ *a SiJa program,* $\mathtt{q}$ *Java bytecode and obs, use two location sets such that $pv(\phi) \subseteq obs$ where $pv(\phi)$ is the set of all program variables occurring in ϕ, then $[\ \mathtt{p} \between \mathtt{q}\ ]@(obs, use)\phi$ is also a SiJa-DL formula.*

The *used program variable* set $usedVar(s, \mathtt{p}, obs)$ is defined similarly as in Definition 19. We formalize the semantics of the weak bisimulation modality for a SiJa program and Java bytecode:

Definition 28 (Weak bisimulation modality for a SiJa program and Java bytecode—semantics)**.** *With $\mathtt{p}, \mathtt{p}_1$ SiJa-programs,* $\mathtt{q}$ *a Java bytecode program, D, s, β, and obs, use are as before, ξ is a mapping function and $\xi(p_2) = q$. Let $val_{D,s,\beta}([\ \mathtt{p} \between \mathtt{q}\ ]@(obs, use)\phi) = tt$ if and only if*

1. $val_{D,s,\beta}([\mathrm{p}]\phi) = tt$
2. $use \supseteq usedVar(s, \mathrm{q}, obs)$
3. *for all* $s' \approx_{use} s$ *we have* $val_{D,s}(\mathrm{p}) \approx_{obs} val_{D,s'}(\mathrm{p}_1) = valB_{D_B,\xi(s')}(q)$

The sequent calculus rules for Java bytecode generation can be defined based on the weak bisimulation modality for a SiJa program and Java bytecode. The starting point is the rules defined in Figure 4.2 that are used in the PTr method of program transformation. In most cases, by changing the generated SiJa program part to its ξ-mapped Java bytecode in the rules presented there, we obtain the rules for bytecode generation, shown in Figure 6.2. The symbol $\overline{\omega}$ represents the generated Java bytecode for SiJa program ω and ξ is the mapping function, and we need to update the program counter after the application of the ifElse and loopInvariant rules to obtain a correct compilation result.

Lemma 13. *The extended sequent calculus rules given in Figure 6.2 are sound.*

The soundness of this lemma is entailed by Lemma 1, Definition 25 and Lemma 12.

Remark. *By introducing a mapping function ξ, we avoid to the semantics of Java bytecode directly but relate it to the semantics of SiJa programs, which results in a better integration of the new weak bisimulation modality with the ones introduced before. In fact, ξ can also be viewed as the* compilation *function since it maps the source code to the bytecode. However, instead of operating on the original source program like a normal compiler would do, ξ is applied on the generated source code and the bytecode is generated based on that already specialized code. So it works as an* optimizing compiler.

6.3 Example

We demonstrate our approach of bytecode generation on an example. The method to be compiled is shown in Figure 6.3.

This program could possibly be used in an online store. It calculates the total amount the customer has to pay if buying `i` items at an item price of 20 EUR. The total sum is stored in both `tot` and `atot`. If the customer can provide a coupon (`cpn`), then a reduction of 50 EUR will be applied. Finally, the total cost is returned as `tot`.

We begin with symbolic execution of our example program. The first statements are simple variable declarations and initializations that are treated similar to assignments. The first steps until reaching the loop are shown below:

$$\dfrac{\dfrac{\Longrightarrow \{\mathtt{tot} := 0 \| \mathtt{atot} := 0\}[\,\mathtt{while(i > 0)} \ldots \between bc_2\,]@(\{\mathtt{tot}\}, use_2)}{\Longrightarrow \{\mathtt{tot} := 0\}[\,\mathtt{atot = 0;} \ldots \between bc_1\,]@(\{\mathtt{tot}\}, use_1)}}{\Longrightarrow [\,\mathtt{tot = 0;} \ldots \between bc_0\,]@(\{\mathtt{tot}\}, use_0)}$$

$$\text{emptyBox}\ \frac{\Gamma \Longrightarrow \mathscr{U}@(obs,\_)\phi,\Delta}{\Gamma \Longrightarrow \mathscr{U}[\ \_ \between \_\ ]@(obs,obs)\phi,\Delta}$$

assignment

$$\frac{\Gamma \Longrightarrow \mathscr{U}\{\mathtt{l} := \mathtt{r}\}[\ \omega \between \overline{\omega}\ ]@(obs,use)\phi,\Delta}{\left(\begin{array}{ll} \Gamma \Longrightarrow \mathscr{U}[\ \mathtt{l}=\mathtt{r};\omega \between \begin{array}{l}\mathtt{iload\_}\xi(\mathtt{r})\\ \mathtt{istore\_}\xi(\mathtt{l})\\ \overline{\omega}\end{array}\ ]@(obs,use-\{\mathtt{l}\}\cup\{\mathtt{r}\})\phi,\Delta & \text{if } \mathtt{l}\in use \\ \Gamma \Longrightarrow \mathscr{U}[\ \mathtt{l}=\mathtt{r};\omega \between \overline{\omega}\ ]@(obs,use)\phi,\Delta & \text{otherwise}\end{array}\right)}$$

ifElse

$$\frac{\begin{array}{c}\Gamma,\mathscr{U}\mathsf{b} \Longrightarrow \mathscr{U}[\ \mathsf{p};\omega \between \overline{\mathsf{p};\omega}\ ]@(obs,use_{\mathsf{p};\omega})\phi,\Delta \\ \Gamma,\mathscr{U}\neg\mathsf{b} \Longrightarrow \mathscr{U}[\ \mathsf{q};\omega \between \overline{\mathsf{q};\omega}\ ]@(obs,use_{\mathsf{q};\omega})\phi,\Delta\end{array}}{\Gamma \Longrightarrow \mathscr{U}[\ \mathtt{if}\ (\mathsf{b})\ \{\mathsf{p}\}\ \mathtt{else}\ \{\mathsf{q}\};\omega \between \begin{array}{l}\mathtt{iload\_}\xi(\mathsf{b})\\ \mathtt{ifeq}\ pc_1\\ \overline{\mathsf{p};\omega}\\ \mathtt{goto}\ pc_2\\ pc_1:\ \overline{\mathsf{q};\omega}\\ pc_2:\ \_\end{array}\ ]@(obs,use_{\mathsf{p};\omega}\cup use_{\mathsf{q};\omega}\cup\{\mathsf{b}\})\phi,\Delta}$$

(after rule application: $pc = pc + 2$)

$$\text{loopUnwind}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \mathscr{U}[\ \mathtt{if}\ (\mathsf{b})\ \{\mathsf{p};\mathtt{while}\ (\mathsf{b})\ \{\mathsf{p}\}\}\ \omega \between \\ \qquad\qquad \overline{\mathtt{if}\ (\mathsf{b})\ \{\mathsf{p};\mathtt{while}\ (\mathsf{b})\ \{\mathsf{p}\}\}\ \omega}\ ]@(obs,use)\phi,\Delta\end{array}}{\Gamma \Longrightarrow \mathscr{U}[\ \mathtt{while}(\mathsf{b})\ \{\mathsf{p}\}\ \omega \between \overline{\mathtt{if}\ (\mathsf{b})\ \{\mathsf{p};\mathtt{while}(\mathsf{b})\ \{\mathsf{p}\}\}\ \omega}\ ]@(obs,use)\phi,\Delta}$$

loopInvariant

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \mathscr{U}inv,\Delta \\ \Gamma,\mathscr{U}\mathscr{V}_{mod}(\mathsf{b}\wedge inv) \Longrightarrow \mathscr{U}\mathscr{V}_{mod}[\ \mathtt{p} \between \overline{\mathtt{p}}\ ]@(use_1\cup\{\mathsf{b}\},use_2)inv,\Delta \\ \Gamma,\mathscr{U}\mathscr{V}_{mod}(\neg\mathsf{b}\wedge inv) \Longrightarrow \mathscr{U}\mathscr{V}_{mod}[\ \omega \between \overline{\omega}\ ]@(obs,use_1)\phi,\Delta\end{array}}{\Gamma \Longrightarrow \mathscr{U}[\ \mathtt{while}(\mathsf{b})\{\mathsf{p}\}\ \omega \between \begin{array}{l}pc_1:\ \mathtt{iload\_}\xi(\mathsf{b})\\ \mathtt{ifeq}\ pc_2\\ \overline{\omega}\\ \mathtt{goto}\ pc_1\\ pc_2:\ \_\end{array}\ ]@(obs,use_1\cup use_2\cup\{\mathsf{b}\})\phi,\Delta}$$

(after rule application: $pc = pc + 2$)

Figure 6.2: A collection of sequent calculus rules for generating Java bytecode.

```
int tot = 0;
int atot = 0;
int i;
boolean cpn;
while (i > 0) {
   tot = tot + 20;
   atot = tot;
   i = i - 1;
}
if (cpn) {
   tot = tot - 50;
   if (tot < 0) {
     tot = 0;
   }
}
return tot;
```

Figure 6.3: Program to be compiled into bytecode.

Notice that *obs* is instantiated with $\{\mathtt{tot}\}$. And as usual, we ignore the postcondition ϕ and unnecessary formulas Γ and Δ. We use bc_i to denote the bytecode to be generated, and use_i to denote the used variable set.

Applying the loop invariant rule creates two new goals (we ignore the `init` branch). The rule application and the resulting goals are shown below.

$$\begin{array}{l}
\mathcal{U}_1\mathcal{V}_a(inv \wedge (i > 0)) \Longrightarrow \mathcal{U}_1\mathcal{V}_a[\,\mathtt{tot} = \mathtt{tot} + 20; \ldots \between bc_4\,]@(\{\mathtt{tot}\} \cup use_3 \cup \{\mathtt{i}\}, use_4) \\
\mathcal{U}_1\mathcal{V}_a(inv \wedge \neg(i > 0)) \Longrightarrow \mathcal{U}_1\mathcal{V}_a[\,\mathtt{if(cpn)} \ldots \between bc_3\,]@(\{\mathtt{tot}\}, use_3) \\
\hline
\Longrightarrow \{\mathtt{tot} := 0 \| \mathtt{atot} := 0\}[\,\mathtt{while(i > 0)\{tot = tot + 20;} \ldots \mathtt{\}if(cpn)} \ldots \between bc_2\,]@(\{\mathtt{tot}\}, use_2)
\end{array}$$

As the used variable set use_4 in the `preserves` branch depends on the instantiation of the used variable set use_3 of the `use case` branch, we continue with the `use case` branch. During symbolic execution of the `use case` branch, two conditional statements have to be executed until reaching the end of the method. The resulting symbolic execution tree is shown below, where updates $\mathcal{U}_1 = \{\mathtt{tot} := 0 \| \mathtt{atot} := 0\}$, $\mathcal{U}_2 = \{\mathtt{tot} := \mathtt{tot} - 50\}$, and path condition $\Gamma_1 = \mathcal{U}_1\mathcal{V}_a(inv \wedge \neg(i > 0))$.

$$\dfrac{\ldots[\ \between\ bc_6\ ]@(\{\mathtt{tot}\},use_6) \qquad \dfrac{\dfrac{\ldots[\ \between\ bc_9\ ]@(\{\mathtt{tot}\},use_9) \quad \dfrac{\Gamma_1,\mathtt{cpn},\mathcal{U}_1\mathcal{V}_a\mathcal{U}_2(\mathtt{tot}<0) \Longrightarrow \mathcal{U}_1\mathcal{V}_a\mathcal{U}_2\{\mathtt{tot} := 0\}[\ \between\ bc_{10}\ ]@(\{\mathtt{tot}\},use_{10})}{\Gamma_1,\mathtt{cpn},\mathcal{U}_1\mathcal{V}_a\mathcal{U}_2(\mathtt{tot}<0) \Longrightarrow \mathcal{U}_1\mathcal{V}_a\mathcal{U}_2[\ \mathtt{tot}=0;\ \between\ bc_8\ ]@(\{\mathtt{tot}\},use_8)}}{\Gamma_1,\mathtt{cpn} \Longrightarrow \mathcal{U}_1\mathcal{V}_a\{\mathtt{tot} := \mathtt{tot}-50\}[\ \mathtt{if(tot<0)}\ldots\ \between\ bc_7\ ]@(\{\mathtt{tot}\},use_7)}}{\Gamma_1,\mathtt{cpn} \Longrightarrow \mathcal{U}_1\mathcal{V}_a[\ \mathtt{tot}=\mathtt{tot}-50;\ldots\ \between\ bc_5\ ]@(\{\mathtt{tot}\},use_5)}}{\mathcal{U}_1\mathcal{V}_a(inv \wedge \neg(i>0)) \Longrightarrow \mathcal{U}_1\mathcal{V}_a[\ \mathtt{if(cpn)}\ldots\ \between\ bc_3\ ]@(\{\mathtt{tot}\},use_3)}$$

Java bytecode is to be synthesized after symbolic execution. Starting with the application of the emptyBox rule, bc_{10} and use_{10} are instantiated as _ and $\{\mathtt{tot}\}$.

Going backwards we can now derive the instantiations for bc_8: `iconst_0 istore_1` (assuming $\xi(\mathtt{tot}) = 1$ for variable tot), and $use_8 = \{\mathtt{tot}\}$ according to assignment rule. The previous rule application was executing a conditional statement. Before we can continue, we have first to derive the instantiations for the other premise. By similar steps as before, we end up with $bc_9 =$ _ and $use_9 = \{\mathtt{tot}\}$. Having now determined all required instantiations, we can continue with the compilation of the conditional statement. As a result we derive for the used variable set $use_7 = \{\mathtt{tot}\}$ and bc_7 :

```
  iload_1
  ifle 1
  iconst_0
  istore_1
1:
  ireturn
```

Applying the remaining rules, we end up with instantiations for bc_3 and use_3 representing the bytecode compilation for the remaining program following the loop and the set of variables used in it.

Assume now that we can derive that cpn is FALSE, i.e., that the customer does not possess a coupon. Partial evaluation allows the translation of both conditional statements to be omitted. This results in faster symbolic execution (as shown below) and in an optimized version of the compiled program. In this case, $bc_3 =$ _, $use_3 = \{\mathtt{tot},\mathtt{cpn}\}$.

$$\dfrac{\Gamma_1,\neg\mathtt{cpn} \Longrightarrow \mathcal{U}_1\mathcal{V}_a[\ \between\ bc_6\ ]@(\{\mathtt{tot}\},use_6)}{\mathcal{U}_1\mathcal{V}_a(inv \wedge \neg(i>0)) \Longrightarrow \mathcal{U}_1\mathcal{V}_a[\ \mathtt{if(cpn)}\ldots\ \between\ bc_3\ ]@(\{\mathtt{tot}\},use_3)}$$

After synthesizing the use case branch, we turn towards the preserves branch, as the required instantiation for use_3 is now known as $\{\mathtt{tot}, \mathtt{cpn}\}$. The symbolic execution tree of the loop body looks like:

$$\dfrac{\dfrac{\dfrac{\mathcal{U}_1\mathcal{V}_a(inv \wedge (i > 0)) \Longrightarrow \mathcal{U}_1\mathcal{V}_a\{\ldots \| \mathtt{i} := \mathtt{i} - 1\}[\ \between\ bc_{13}\ ]@(\{\mathtt{tot}, \mathtt{cpn}, \mathtt{i}\}, use_{13})}{\mathcal{U}_1\mathcal{V}_a(inv \wedge (i > 0)) \Longrightarrow \mathcal{U}_1\mathcal{V}_a\{\ldots \| \mathtt{atot} := \mathtt{tot}\}[\ \mathtt{i} = \mathtt{i} - 1;\ \between\ bc_{12}\ ]@(\{\mathtt{tot}, \mathtt{cpn}, \mathtt{i}\}, use_{12})}}{\mathcal{U}_1\mathcal{V}_a(inv \wedge (i > 0)) \Longrightarrow \mathcal{U}_1\mathcal{V}_a\{\mathtt{tot} := \mathtt{tot} + 20\}[\ \mathtt{atot} = \mathtt{tot}; \ldots\ \between\ bc_{11}\ ]@(\{\mathtt{tot}, \mathtt{cpn}, \mathtt{i}\}, use_{11})}}{\mathcal{U}_1\mathcal{V}_a(inv \wedge (i > 0)) \Longrightarrow \mathcal{U}_1\mathcal{V}_a[\ \mathtt{tot} = \mathtt{tot} + 20; \ldots\ \between\ bc_4\ ]@(\{\mathtt{tot}, \mathtt{cpn}, \mathtt{i}\}, use_4)}$$

Synthesis of the bytecode follows the same pattern as described for the use case branch. The Java bytecode generated under the assumption that $\mathtt{cpn} = \mathtt{FALSE}$ by our approach is in Figure 6.4. Here, $\xi(\mathtt{tot}) = 1$ and $\xi(\mathtt{i}) = 2$ for variable tot and i.

```
  iconst_0
  istore_1
1:
  iload_2
  ifle 2
  iload_1
  bipush 20
  iadd
  istore_1
  iinc 2, -1
  goto 1
2:
  iload_1
  ireturn
```

Figure 6.4: Generated Java bytecode.

We can see that the resulting Java bytecode is sound and also more optimized than that obtained by a normal line-by-line compiler. For instance, the bytecode for the statement $\mathtt{atot} = \mathtt{tot}$ is not generated because it will not affect the final result of the observable locations (return variable). And the bytecode for the conditional is ignored thanks to partial evaluation.

If one is only interested in sound compilation, but not in functional verification, then the trivial postcondition *true* is sufficient. As a consequence, it suffices to supply *true* as well for the invariant of the loopInvariant rule and symbolic execution becomes fully automatic. The resulting first-order proof obligations are no problem for state-of-art solvers.

7 Conclusion

7.1 Summary

In this thesis, we are concerned with the safety and security of programs. The problems addressed here are the correctness of SiJa (a subset of Java) source code and Java bytecode, and the information flow security of SiJa programs. A lot of research has been made on these topics, but almost all of them study each topic independently and no approach can handle all of these aspects. We proposed a uniform framework that integrates the effort of proving correctness and security into one process. The core concept for this uniform approach is sound program transformation based on symbolic execution and deduction.

Symbolic execution is used to execute a source program symbolically, so that it reveals the information on all program execution paths which can be used further for optimization and verification. We use the state-of-the-art Java verification system KeY to perform symbolic execution of SiJa programs. The first-order reasoning capabilities of KeY analyzes variable dependencies, aliasing, and eliminates infeasible execution paths. The symbolic execution tree (or: proof tree) can be reduced further by interleaving partial evaluation with symbolic execution. This speeds up the verification process for a SiJa program, as shown in Chapter 3.

Program transformation is performed based on the symbolic execution tree, which is achieved as a side product of SiJa source code verification using KeY, or it can be built explicitly for the purpose of program transformation which does not require strong loop invariants, postconditions, etc. The sequent calculus rules used to perform symbolic execution are extended with bisimulation modalities. An extended sequent with bisimulation modalities has the form: $\Gamma \Longrightarrow \mathcal{U}[\,\mathrm{p} \between \mathrm{q}\,]@(obs, use)\phi, \Delta$. It means at the current state, we get program p and q that are bisilmilar with respect to the observable locations *obs*. On one hand, this extension does not affect the normal symbolic execution, nor the verification result achieved with the normal sequent calculus rules. On the other hand, the additional information, namely observable locations *obs*, used variables *use* and the program q generated-so-far, recorded in the extended sequent, contributes to a sound program transformation by the step-wise inverse application of the extended calculus rules. To be more precise, *obs* is the set of observable locations that matters to the output of a program; the used variables set *use* tracks all the locations in the current state that may affect the result of *obs* after program execution; program q accumulates the program generated so far in a generalized sequential block, and it represents the program obtained after the generation is done. The update $\mathcal{U}$ records the symbolic state resulting from program execution along a certain path.

The program transformation process is a bottom-up traversal of the symbolic execution tree. Starting with the nodes where the program is empty, it first synthesizes the program in each sequential block, and then builds a program in a generalized sequential block by combining sibling blocks, until the whole program is generated. Our basic approach to program transformation, called PTr, generates a program with the granularity of simple statements. The resulting program is optimized up to a certain point considering that some assignments that are not relevant to the final values of *obs* are not generated. Soundness of this approach, which is entailed by the soundness of the extended sequent calculus rules, has been proven. This approach can be optimized in two directions. The first direction, called PTr + PE, is to optimize the symbolic execution tree, by interleaving partial evaluation and symbolic execution. The second direction, called PTr + SNF, is to take into account the updates in the generation phase and synthesizes a program as optimal as possible in each sequential block. Combining these two directions, PTr + PE + SNF, we obtain the most optimized program transformation approach studied in this thesis. Chapter 4 discussed the above program transformation approaches in detail and showed their soundness.

An interesting observation about the extended sequent with bisimulation modalities $\Gamma \Longrightarrow \mathcal{U}[\ \mathtt{p}\ \between\ \mathtt{q}\ ]@(obs, use)\phi, \Delta$ is the following: *obs* can be chosen freely, as its choice will not affect the soundness of the program transformation. Then we can include information flow security of a SiJa source program into our picture by fixing *obs* as `Low` variables. By doing so, we can generate a program that can be viewed as the dependency flow of `Low` variables. Intuitively, if no `High` variables are present in the generated program, then the non-interference policy is enforced; otherwise, we need to inspect the problem further with other techniques, as the generated program is an approximation of the "real" dependency flow of `Low` variables. Nevertheless, we have shown that using PTr + PE + SNF for program transformation gives a better result than using PTr for analyzing information flow security. Another angle for non-interference policy enforcement is to inspect the used variable set *use* after program transformation. Since *use* contains the locations that may affect the values of *obs*, when the program is fully generated, *use* are indeed the input locations that may affect the output `Low` variables. So we can achieve a strengthened statement: the non-interference policy is enforced if `High` variables do not appear in *use* after program transformation is finished. This result is still approximate, but it provides another angle to tackle information flow security problems, and it is more precise than the results achieved by many existing approaches, e.g., the ones based on type systems. Chapter 5 addressed this security aspect.

In the extended sequent with bisimulation modalities $\Gamma \Longrightarrow \mathcal{U}[\ \mathtt{p}\ \between\ \mathtt{q}\ ]@(obs, use)\phi, \Delta$, the languages of `p` and `q` are not necessarily the same. We can also generate Java bytecode `q` from SiJa source code `p` using the extended sequent calculus rules. A mapping function from SiJa source code to Java bytecode is introduced to integrate the bytecode smoothly into the calculus. This generation works as a SiJa compiler, and, as a consequence of the soundness of the rules, it is a verified optimizing compiler. It is an alternative way to obtain correct bytecode in addition to bytecode verification or compiler verification which are normally very difficult. This idea has been presented in Chapter 6.

To put everything together, we can guarantee safety and security of a SiJa program in one process. Starting with a SiJa program annotated with proper JML specifications, we can verify the correctness of source code by using KeY. After verification, if necessary, we can optimize the program using the program transformation techniques introduced in Chapter 4. To enforce the non-interference policy of SiJa source code, we fix *obs* as `Low` variables and do the analysis described in Chapter 5. Java bytecode can be generated as shown in Chapter 6, which guarantees correct bytecode. The total process contains four phases: one symbolic execution phase, and three generation phases for different purposes. In fact, if we do not need to optimize the program, after source code verification, we can ensure the bytecode correctness and enforce the non-interference policy of the source code within one phase. In addition to normal program generation for bytecode, we can maintain another obs_i initialized with `Low` variables and use_i that is updated like the actual *use* (but without generating any program) along the generation. In the end, we obtain a generated Java bytecode together with two used variable sets *use* and use_i. Now we can also enforce the non-interference policy by inspecting use_i.

The outline of this thesis work can be summarized in Figure 7.1.

$$\Gamma \Longrightarrow \mathcal{U}[\,\mathtt{p} \between \mathtt{q}\,]@(obs, use)\phi, \Delta$$

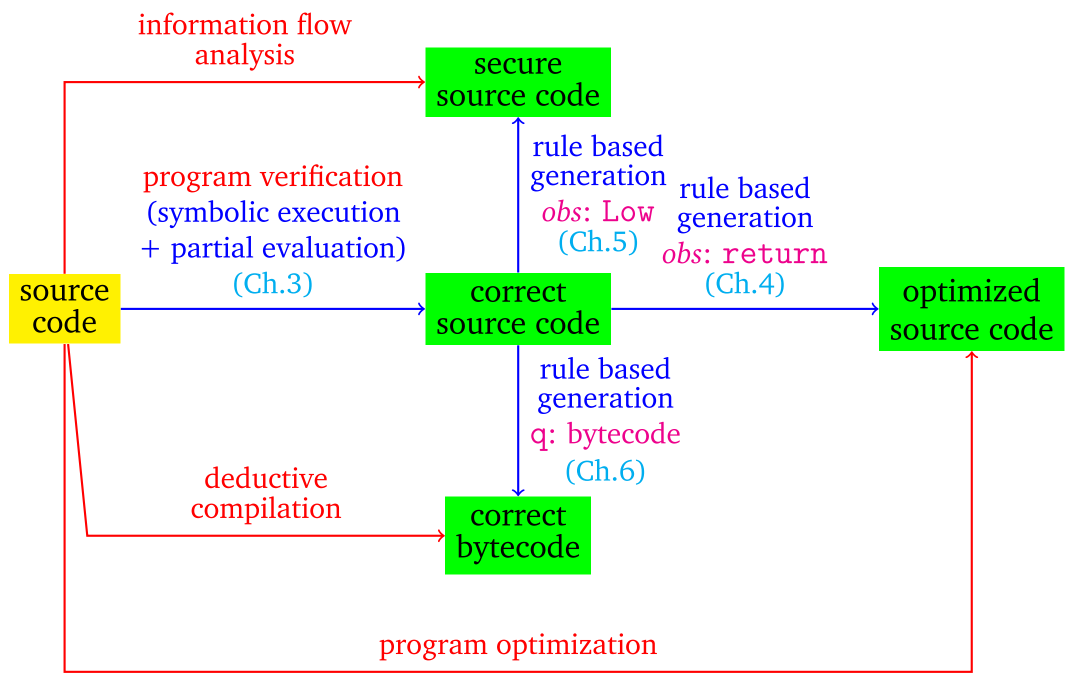

Figure 7.1: Software correctness and security: a uniform framework.

7.2 Future Work

Software verification, compiler verification and information flow security are active research areas, and yet a lot of work needs to be done. In particular, to continue the work of this thesis, we plan to investigate in the following aspects:

(i) Supporting more features of the Java language. So far we have considered SiJa, a subset of Java without floating point and concurrency, in this thesis. Formal verification of Java floating point and concurrency has always been a difficult, yet desirable goal. It is worth to put more effort in this research area.

(ii) Further optimization. In our work, optimization is performed by using interleaving partial evaluation and symbolic execution and involving updates in program generation. There are other optimizations that can be made. For example, considering ranking function may provide us with heuristics of treating a loop. We plan to seek further optimization opportunities.

(iii) Enforcing more security policies. This thesis has presented an approach to enforce non-interference policy for SiJa source code. Other security policies such as information *declassification* [SS05], information integrity [BRS10] and erasure [CM05] may also be interesting. We plan to address other security policies within our framework in the future.

(iv) Consolidate the implementation on Java bytecode. The implementation of the program transformation approaches presented in this thesis is still in a prototypical phase. It can be consolidated further, especially on the implementation of deductive compilation to generate Java bytecode.

(v) Application of our framework to other scenarios. The framework presented in this thesis can be generally applied. For example, we can generate some intermediate languages that can be used for other specific purposes. The soundness of the corresponding extended sequent calculus rules entails the soundness of the program generation process.

Bibliography

[AAG+07] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost analysis of Java bytecode. In *ESOP*, pages 157–172, 2007.

[Abr96] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[AGZP10] Elvira Albert, Miguel Gomez-Zamalloa, and German Puebla. PET: a partial evaluation-based test case generation tool for Java bytecode. In *ACM SIGPLAN WS on Partial Evaluation and Semantics-based Program Manipulation*. ACM Press, 2010.

[AS09] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *CSF*, pages 43–59, 2009.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM.

[Bau07] Marcus Baum. Debugging by visualizing of symbolic execution. Master's thesis, Dept. of Computer Science, Institute for Theoretical Computer Science, June 2007.

[BBC+06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *EuroSys*, pages 73–85, 2006.

[BCD+05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer, 2001.

[BEL75] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SelectÑa formal system for testing and debugging programs by symbolic execution. In *Proceedings of the international conference on Reliable software*, pages 234–245. ACM, 1975.

[BFL$^+$11] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6), 2011.

[BHJ09] Richard Bubel, Reiner Hähnle, and Ran Ji. Interleaving symbolic execution and partial evaluation. In *Post Conf. Proc. FMCO2009*, LNCS. Springer-Verlag, 2009.

[BHJ10] Richard Bubel, Reiner Hähnle, and Ran Ji. Program specialization via a software verification tool. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Post Proc. of FMCO'10*, LNCS. Springer, 2010.

[BHS06] Bernhard Beckert, Reiner Hähnle, and Peter Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2006.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007.

[BHW09] Richard Bubel, Reiner Hähnle, and Benjamin Weiss. Abstract interpretation of symbolic execution with explicit state updates. In Frank de Boer, Marcello M. Bonsangue, and Eric Madelaine, editors, *Post Conf. Proc. 6th Intl. Symposium on Formal Methods for Components and Objects (FMCO)*, volume 5751 of *LNCS*, pages 247–277. Springer, 2009.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[Boa06] The Inquiry Board. Ariane 5 flight 501 failure, report by the inquiry board, Paris, 19 July 2006.

[BP06] Bernhard Beckert and André Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In U. Furbach and N. Shankar, editors, *Proc. Intl. Joint Conference on Automated Reasoning, Seattle, USA*, volume 4130 of *LNCS*, pages 266–280. Springer, 2006.

[BRL03] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: a developer-oriented approach. In *Proc. Formal Methods Europe, Pisa, Italy*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.

[BRS10] Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. Unifying facets of information integrity. In *ICISS*, pages 48–65, 2010.

[BS03] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

[Bur74] Rod M. Burstall. Program proving as hand simulation with a little induction. In *IFIP Congress*, pages 308–312, 1974.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Language, Los Angeles*, pages 238–252. ACM Press, New York, January 1977.

[CDH$^+$11] Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Broch Johnsen, Ina Schaefer, Jan Schäfer, Rudi Schlatte, and Peter Y. H. Wong. Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *LNCS*, pages 417–457. PUB-SV, 2011.

[CK05] David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*, pages 108–128. Springer, 2005.

[CKK$^+$12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c - a software analysis perspective. In *SEFM*, pages 233–247, 2012.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, pages 168–176, 2004.

[CM05] Stephen Chong and Andrew C. Myers. Language-based information erasure. In *CSFW*, pages 241–254, 2005.

[Coh77] Ellis S. Cohen. Information transmission in computational systems. In *SOSP*, pages 133–139, 1977.

[CSH10] Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing formal specifications using testing. In *TAP*, pages 6–21, 2010.

[Dav03] Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28:2–2, November 2003.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[Den82] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[Der] Nachum Dershowitz. Software horror stories. `http://www.cs.tau.ac.il/~nachumd/horror.html`.

[dHT08] Jonathan de Halleux and Nikolai Tillmann. Parameterized unit testing with Pex. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *LNCS*, pages 171–181. Springer, 2008.

[Dij70] Edsger W. Dijkstra. Notes on Structured Programming. circulated privately, see `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF`, April 1970.

[Dij86] Edsger W. Dijkstra. Visuals for BP's Venture Research Conference. circulated privately, see `http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD963.PDF`, June 1986.

[DLR06] Xianghua Deng, Jooyong Lee, and Robby. Bogor/Kiasan: a k-bounded symbolic execution for checking strong heap properties of open systems. In *Proc. 21st IEEE/ASM Intl. Conference on Automated Software Engineering, Tokyo, Japan*, pages 157–166. IEEE Computer Society, 2006.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[EH07] Christian Engel and Reiner Hähnle. Generating unit tests from formal proofs. In Bertrand Meyer and Yuri Gurevich, editors, *Proc. Tests and Proofs (TAP), Zürich, Switzerland*, volume 4454 of *LNCS*. Springer, 2007.

[ERSW09] Christian Engel, Andreas Roth, Peter H. Schmitt, and Benjamin Weiß. Verification of modifies clauses in dynamic logic with non-rigid functions. Technical Report 2009-9, Department of Computer Science, University of Karlsruhe, 2009.

[FLL$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.

[FM03] Stephen N. Freund and John C. Mitchell. A type system for the java bytecode language and verifier. *J. Autom. Reasoning*, 30(3-4), 2003.

[FM07] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, Berlin, Germany*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Proc. 29th ACM Symposium on Principles of programming languages*, pages 191–202. ACM Press, 2002.

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[HBBR10] Reiner Hähnle, Marcus Baum, Richard Bubel, and Marcel Rothe. A visual interactive debugger based on symbolic execution. In *ASE*, pages 143–146, 2010.

[HHRS86] Reiner Hähnle, Maritta Heisel, Wolfgang Reif, and Werner Stephan. An interactive verification system based on dynamic logic. In Jörg Siekmann, editor, *Proc. 8th Conference on Automated Deduction CADE, Oxford*, volume 230 of *LNCS*, pages 306–315. Springer, 1986.

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In *SPIN*, pages 235–239, 2003.

[HKT00a] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[HKT00b] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, October 2000.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[Hoa03] Tony Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[HRS87] Maritta Heisel, Wolfgang Reif, and Werner Stephan. Program verification by symbolic execution and induction. In K. Morik, editor, *Proc. 11th German Workshop on Artifical Intelligence*, volume 152 of *Informatik Fachberichte*. Springer, 1987.

[HS06] Sebastian Hunt and David Sands. On flow-sensitive security types. In *POPL*, pages 79–90, 2006.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[JB12] Ran Ji and Richard Bubel. Pe-key: A partial evaluator for java programs. In *IFM*, pages 283–295, 2012.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, 1993.

[JHB13] Ran Ji, Reiner Hähnle, and Richard Bubel. Program transformation based on symbolic execution and deduction. In *SEFM*, pages 289–304, 2013.

[JP08] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, August 2008.

[Kah87] Gilles Kahn. Natural semantics. In *Proc. Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 247 of *LNCS*, pages 22–39. Springer, 1987.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[LBR03] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06y, Iowa State University, Department of Computer Science, 2003. Revised June 2004.

[Lei08] Dirk Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Saarbr§cken, 2008.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54, 2006.

[Ler09a] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[Ler09b] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.

[Loc10] Andreas Lochbihler. Verifying a compiler for java threads. In *ESOP*, pages 427–447, 2010.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[MCDE02] Madanlal Musuvathi, Andy Chou, David L. Dill, and Dawson R. Engler. Model checking system software with cmc. In *ACM SIGOPS European Workshop*, pages 219–222, 2002.

[Mey86] Bertrand Meyer. Design by contract. *Technical Report TR-EI-12/CO*, pages 1–50, 1986.

[Mey92] Bertrand Meyer. Applying “design by contract”. *IEEE Computer*, 25(10):40–51, October 1992.

[Mey00] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2000.

[Mos05] Wojciech Mostowski. Formalisation and verification of Java Card security properties in dynamic logic. In Maura Cerioli, editor, *Proc. Fundamental Approaches to Software Engineering (FASE), Edinburgh*, volume 3442 of *LNCS*, pages 357–371. Springer, April 2005.

[MP67] John Mccarthy and James Painter. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science, volume 19 of Proc. of Symposia in Applied Mathematics*, pages 33–41. American Mathematical Society, 1967.

[MPH00] Jörg Meyer and Arnd Poetzsch-Heffter. An architecture for interactive program provers. In *TACAS*, pages 63–77, 2000.

[MW72] Robin Milner and Richard Weyhrauch. Proving compiler correctness in a mechanized logic. In *Proc. 7th Annual Machine Intelligence Workshop, volume 7 of Machine Intelligence*, pages 51–72. Edinburgh University Press, 1972.

[Mye04] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, second edition, 2004.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[Pra76] Vaughan R. Pratt. Semantical considerations on floyd-hoare logic. In *FOCS*, pages 109–121, 1976.

[PV04] Corina S. Pasareanu and Willem Visser. Verification of Java programs using symbolic execution and invariant generation. In Susanne Graf and Laurent Mounier, editors, *Proc. Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain*, volume 2989 of *LNCS*, pages 164–181. Springer, 2004.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[Rüm06] Philipp Rümmer. Sequential, parallel, and quantified updates of first-order structures. In Miki Hermann and Andrei Voronkov, editors, *Proc. Logic for Programming,* *Artificial Intelligence and Reasoning, Phnom Penh, Cambodia*, volume 4246 of *LNCS*, pages 422–436. Springer, 2006.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM.

[SC11] Amin Shali and William R. Cook. Hybrid partial evaluation. In *OOPSLA*, pages 375–390, 2011.

[SLC03] Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. Automatic program specialization for Java. *ACM Transactions on Programming Languages and Systems*, 25, 2003.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[SS05] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *CSFW*, pages 255–269, 2005.

[Ste04] Kurt Stenzel. A formally verified calculus for full Java Card. In Charles Rattray, Savi Maharaj, and Carron Shankland, editors, *Proc. Algebraic Methodology and Software Technology, AMAST 2004, Stirling, Scotland, UK*, volume 3116 of *Lecture Notes in Computer Science*, pages 491–505. Springer, 2004.

[VIS96] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

[Vol99] Dennis M. Volpano. Safety versus secrecy. In *SAS*, pages 303–311, 1999.

[Wei09] Benjamin Weiß. Predicate abstraction in a program logic calculus. In Michael Leuschel and Heike Wehrheim, editors, *Proc. 7th International Conference on integrated Formal Methods (iFM 2009)*, volume 5423 of *LNCS*, pages 136–150. Springer, 2009.

[Wei11] Benjamin Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, 2011.