Chapter 14
Program Transformation and Compilation

Ran Ji and Richard Bubel

The main purpose of the KeY system is to ensure program correctness w.r.t. a formal specification on the level of source code. However, a flawed(?) compiler may invalidate correctness properties that have been formally verified for the program's source code. Hence, we additionally need to guarantee the correctness of the compilation result w.r.t. its source code.

Compiler verification, as a widely used technique to prove the correctness of compilers, has been a research topic for more than 40 years [McCarthy and Painter, 1967, Milner and Weyhrauch, 1972]. Previous works [Leroy, 2006, 2009, Leinenbach, 2008] have shown that compiler verification is an expensive task requiring nontrivial user interactions in proof-assistants like Coq [Leroy, 2009]. Maintaining these proofs for changes to the compiler back-end (e.g., support of new language features or optimization techniques) is not yet counted into that effort.

In this chapter, instead of verifying a compiler, we use the verification engine of KeY to prove a correct bytecode generation through a sound program transformation approach. Thus program correctness on source code level is inherited to the bytecode level. The presented approach guarantees that the behavior of the compiled program coincides with that of the source program in the sense that both programs terminate in states in which the values of a user specified region of the heap are equivalent.

Moreover, we often want the generated program to be more optimized than the original program. If the source and the target programs are in the same language, this program translation process is also known as *program specialization* or *partial evaluation*.

The correctness of the source program (w.r.t. its specification) entails correctness of the generated program. No further verification on the level of bytecode is needed, though verification of Java programs on the bytecode level, even if interaction is needed, is also possible using dynamic logic [Ulbrich, 2011, 2013].

When constructing the symbolic execution tree (see Section 11.2) the program is analyzed by decomposing complex statements into a succession of simpler statements. Information about the heap and local program variables is accumulated and added in the form of formulas and/or updates. This information can be used to deem certain execution paths as unfeasible.

W. Ahrendt et al. (Eds.): Deductive Software Verification, LNCS 10001, pp. 473–492, 2016
DOI: 10.1007/978-3-319-49812-6_14

Technically, we implemented symbolic execution as part of the sequent calculus (see Section 3.5.6), whose rules are applied analytically from bottom-to-up. For the program generation part, the idea is to apply the sequent calculus rules reversely (i.e., top-down) and to generate the target program step-by-step.

This chapter is structured as follows: Section 14.1 introduces partial evaluation and how it can be interleaved with symbolic execution to boost the performance of automatic verification. We discuss how to achieve verified correct compilation in Section 14.2 and discuss a prototypical implementation in Section 14.3.

14.1 Interleaving Symbolic Execution and Partial Evaluation

We first motivate the general idea for interleaving symbolic execution and partial evaluation (Section 14.1.1). Then we show how to integrate a program transformer soundly into a program calculus in Section 14.1.2 and conclude with a short evaluation of the results.

14.1.1 General Idea

To motivate our approach of interleaving partial evaluation and symbolic execution, we first take a look at the program shown in Figure 14.1(a). The program adapts the value of variable `y` to a given `threshold` with an accuracy of `eps` by repeatedly increasing or decreasing `y` as appropriate. The function `abs(·)` computes the absolute value of an integer.

Symbolically executing the program results in the symbolic execution tree (introduced in Section 11.2) shown in Figure 14.2, which is significantly more complex than the program's control flow graph (CFG) in Figure 14.1(b). The reason is that symbolic execution unwinds the program's CFG producing a tree structure. As a consequence, identical code is repeated on many branches, however, under different path conditions and in different symbolic states. Merging back different nodes of the tree is usually not possible without approximation or abstraction [Bubel et al., 2009, Weiß, 2009].

During symbolic execution, there are occasions in which fields or parameters have a value which is fixed a priori, for instance, because certain values are fixed for some call sites, the program is an instantiation of a product family or contracts exclude certain program paths. In our case, the program from Figure 14.1(a) is run with a fixed initial value (`80`) for `y` and the `threshold` is fixed to `100`.

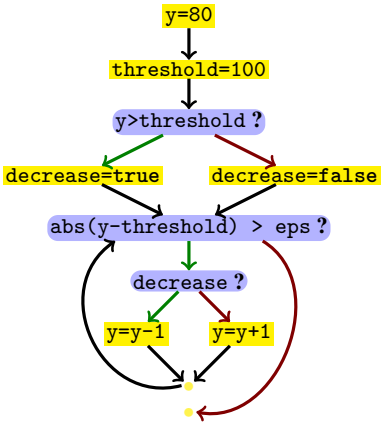
To exploit this knowledge about constant values and to derive more efficient programs, partial evaluation has been used since the mid 1960s, for instance as part of optimizing compilers. The first efforts were targeted towards Lisp. Due to the rise in popularity of functional and logic programming languages, the 1980s saw a large

```
y = 80;
threshold = 100;

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

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

(a) Source code of control circuit



(b) Control flow graph of control circuit

Figure 14.1 A simple control circuit *Java* program and its control flow graph

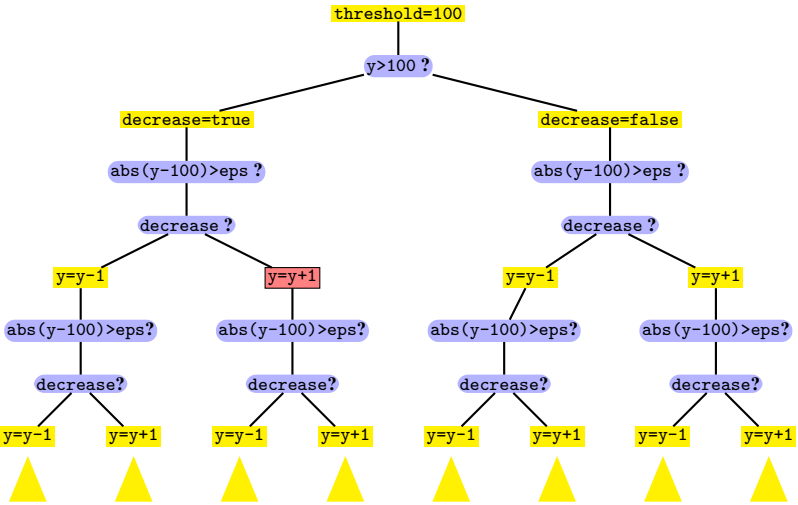


Figure 14.2 Symbolic execution tree of the control circuit program

amount of research in partial evaluation of such languages. A seminal text on partial evaluation is the book by Jones et al. [1993].

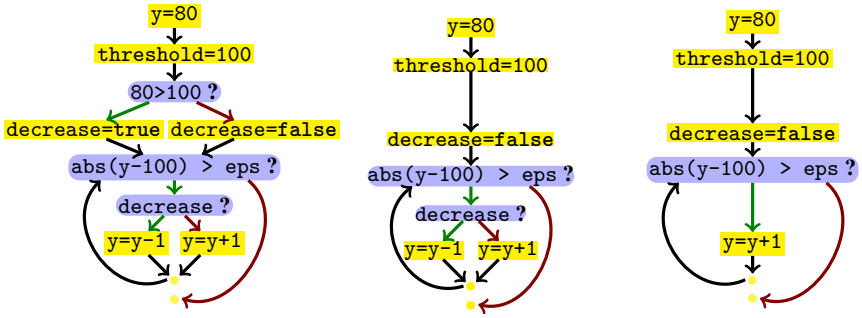
In contrast to symbolic execution, the result of a partial evaluator, also called *program specializer* (or short `mix`), is not the symbolic value of output variables, but another, equivalent program. The known fixed input is also called *static input* while the part of the input that is not known at compile time is called *dynamic input*.

Partial evaluation traverses the CFG (e.g., the one of Figure 14.1(b)) with a partial evaluator, while maintaining a table of concrete (i.e., constant) values for the program

locations. In our example, that table is empty at first. After processing the two initial assignments, it contains $\mathscr{U} = \{\mathtt{y} := 80 \,||\, \mathtt{threshold} := 100\}$.

Whenever a new constant value becomes known, the partial evaluator attempts to propagate it throughout the current CFG. This *constant propagation* transforms the CFG from Figure 14.1(b) into the one depicted in Figure 14.3(a). We can observe that occurrences of y within the loop (incl. the loop guard) have *not* been replaced. The reason for this is that the value of y at these occurrences is not static, because it might be updated in the loop. 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 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 the *constant expression* `80>100` 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 14.3(b). 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 shown in Figure 14.3(c).



(a) CFG after constant propagation for `threshold` and y

(b) CFG after evaluation of constant expressions (and dead code elimination)

(c) Final CFG after additional constant propagation and dead code elimination

Figure 14.3 Partial evaluation of a simple 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 (i.e., fixed values at compile time).

Typical usage scenarios for symbolic execution like program verification are not of this kind. For example, in the program shown in Figure 14.1, 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.

To be specific, we reconsider the example shown in Figure 14.1(a), but we remove the first statement, which assign y the value 80. As observed above, no noteworthy simplification of the program's CFG can be any longer achieved by partial evaluation. The CFG's structure after partial evaluation remains exactly the same and only the occurrences of variable `threshold` are replaced by the constant value `100`. If we symbolically execute this program, then the resulting execution tree spanned by unrolling the loop twice is shown in Figure 14.2. 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.

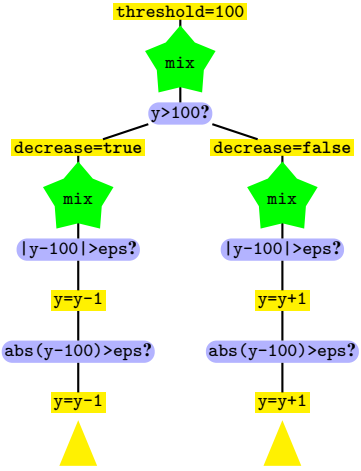


Figure 14.4 Symbolic execution with interleaved partial evaluation

All subsequent branches result from either the loop condition (omitted in Figure 14.2) 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 `y=y+1` (filled in red) is infeasible, because the value of `decrease` is `true` in that branch. Symbolic execution will not continue on these infeasible branches, but abandon them 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 unwound first 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 14.4 (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.

14.1.2 The Program Specialization Operator

We define a program specialization operator suitable for interleaving partial evaluation with symbolic execution in JavaDL. The operator implements a program transformer which issues correctness conditions as side-proofs that are 'easy' to proof directly and can thus be safely integrated into the sequent calculus. This approach avoids formalizing the partial evaluator in the program logic itself which would be tedious and inefficient.

Definition 14.1 (Program Specialization Operator). Let Σ be a sufficiently large signature containing countably infinitely many program variables and function symbols for any type and arity. A *program specialization operator*

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

takes as arguments a (i) program statement or expression, (ii) an update and (iii) a formula; and maps these to a program statement or expression.

The intention behind the above definition is that $\mathtt{p} \downarrow_\Sigma (\mathcal{U}, \phi)$ denotes a "simpler" but semantically equivalent version of $\mathtt{p}$ under the assumption that both are executed in a state which satisfies the constrained imposed by $\mathcal{U}$ and ϕ. The program specialization operator may introduce new temporary variables or function symbols.

Interleaving partial evaluation and symbolic execution is achieved by introduction rules for the specialization operator. Application of the program transformer is triggered by application of the rule

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

where $(\mathtt{p}) \downarrow (\mathcal{U}, \mathit{true})$ returns a semantically equivalent program w.r.t. initial state $\mathcal{U}$ and condition ϕ. The program transformer is usually defined recursively over the program structure. We discuss a selection of program transformation rules that can be used to define the specialization operator in the next section.

14.1.3 Specific Specialization Actions

We instantiate the generic program specialization operator of Definition 14.1 with some possible actions. In each case we derive soundness conditions.

Specialization Operator Propagation

The specialization operator needs to be propagated through 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 $(\mathscr{U},\phi)$, 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 program transformation (read $p \rightsquigarrow p'$ as program p is transformed into program p')

$$(\mathtt{p};\mathtt{q})\downarrow(\mathscr{U},\phi) \quad \rightsquigarrow \quad \mathtt{p}\downarrow(\mathscr{U},\phi);\ \mathtt{q}\downarrow(\mathscr{U}',\phi') \ .$$

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

1. Let *mod* be a collection that contains all program locations possibly changed by p including local variables. This can be proven similar to framing in case of loop invariants (see Section 8.2.5).
2. Let $\mathscr{V}_{mod}$ be the anonymizing update for *mod*, which assigns each local program variable contained in *mod* a new constant and performs the heap anonymization using the *anon* function. By fixing $\mathscr{U}' := \mathscr{U}\,\mathscr{V}_{mod}$, 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 $\models \mathscr{U}(\phi \rightarrow \langle p\rangle\phi')$ holds. This ensures that the postcondition of p is correctly represented by ϕ'. Computation of such a ϕ' can be arbitrarily complex. The actual complexity depends on the concrete realization of the program specialization operator. Usually, ϕ' is a relatively cheaply computed safe approximation (abstraction) of p's postcondition.

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

$$(v)\downarrow(\mathscr{U},\phi) \rightsquigarrow c$$

models the replacement operation. To ensure soundness the rather obvious condition $\models \mathcal{U}(\phi \rightarrow v \doteq c)$ has to be proved where c is an interpreted constant (e.g., a compile-time constant or literal).

Dead-Code Elimination

Constant propagation and constant expression evaluation often result 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

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

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: $\models \mathcal{U}(\phi \rightarrow \texttt{b} \doteq \mathit{TRUE})$.

Another case is

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

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

Safe Field Access

Partial evaluation can be used to mark expressions as safe that contain field accesses or casts that may otherwise cause abrupt 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 $\models \mathcal{U}(\phi \rightarrow \neg(\texttt{o} \doteq \texttt{null}))$.

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 3.7.1). 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:

$$\texttt{res} = \texttt{o}.m(\texttt{a}_1,\ldots,\texttt{a}_n);\downarrow(\mathscr{U},\phi) \quad \rightsquigarrow$$
$$\texttt{res} = @((Cl)\texttt{o}\downarrow(\mathscr{U},\phi)).m(\texttt{a}_1\downarrow(\mathscr{U},\phi),\ldots,\texttt{a}_n\downarrow(\mathscr{U},\phi));$$

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

A note to the side conditions: The side conditions are in general full-blown first-order proofs and the needed effort to discharge them could eliminate any positive effects of the specialization. But in practice, these side conditions can be (i) proven very easily as the accumulated information is already directly contained in the formula ϕ without the need of full first-order reasoning; and (ii) the conditions can be proven in separate side-proofs and hence, do not pollute the actual proof tree. This results in a shorter and more human-readable proof object.

14.1.4 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:

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

The element `Button` is implemented in one way for Max OS X while it is implemented differently for the X Window System. The class `Button`, which is extended by the classes `CheckBox`, `Component`, and `Dialog`, defines the method `paint()`. Altogether, `paint()` is implemented in 16 different classes including `ButtonX11`, `ButtonMPC`, `RadioButtonX11`, `MenuItemX11`, etc. The type hierarchy is outlined in Figure 14.5. In the code fragment above, `button` is assigned an object of type `RadioButtonX11` which implements `paint()`. We want to prove that it always terminates, and hence, the formula $\langle \texttt{gui} \rangle true$ should be provable where `gui` abbreviates the code above.

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 (see Section 3.6.5.5), each corresponding to a possible implementation of `button`

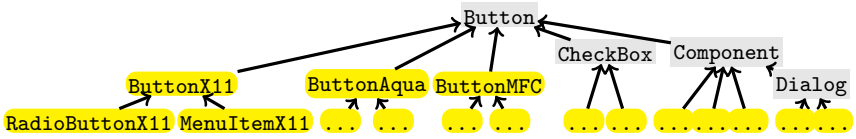


Figure 14.5 Type hierarchy for the GUI example

in one of the subclasses of `Button`. The proof is constructed automatically using an experimental version of KeY; the proof consists of 161 nodes on ten branches.

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 presented in the previous section the information that the runtime type of `button` is `RadioButtonX11` (known from the declared type of variable `radiobuttonX11` and the type hierarchy) and the only possible implementation of `button.paint()` is `RadioButtonX11.paint()`. All other possible implementations are pruned. Only 24 nodes and two branches occur in the proof tree when running KeY integrated with a partial evaluator.

The reduction in the size of the proof tree is in particular important for human readability and increases the efficiency of the interactive proving process. A thorough evaluation and more details can be found in Ji [2014].

14.2 Verified Correct Compilation

The previous section was concerned with the interleaving of partial evaluation and symbolic execution. In this section, we go one step further and discuss how to employ JavaDL and symbolic execution calculus to support more semantics-preserving program transformations. One interesting use case is the compilation of a program into a target language that the compiled program behaves verifiably equivalent w.r.t. to its source code version. For ease of presentation, we describe the approach here for a source-to-source transformation of a while language. But the presented approach can be extended to all sorts of source and target languages. A detailed description including bytecode compilation can be found in [Ji, 2014].

Equivalence checking between code and compilation result is important in compiler correctness checking. General equivalence checking of programs of the same abstraction level is also an active field of research.

The simplified language for this presentation is a while-language (with a Java-like syntax) that operates on integer variables and comes without intricacies like abrupt termination. The verifiably correct output is a simplified (and possibly specialized) variant of the original one.

Semantic equivalence is a relational property of the two compared programs. In order to accommodate such relational problems on the syntactical level in JavaDL, a new modality, called the *weak bisimulation modality*, is introduced that contains not

one but two programs. The two programs in the modality are meant to be equivalent, but need not reach fully equivalent poststates. A criterion can be given which decides about the equivalence of states. This criterion is the set of observable variables *obs* on which the termination states have to coincide.

Definition 14.2 (Weak bisimulation modality—syntax). Let p, q be two while-language programs, $obs, use \subseteq$ ProgVSym sets of program variables and $\phi \in$ DLFml a first-order formula.

We extend the definition of JavaDL formulas: Under the above conditions $[\,\mathtt{p} \between \mathtt{q}\,]@(obs,use)\phi$ is also a JavaDL formula.

This modality is closely related to the relational Hoare calculus by Benton [2004], the notion of product programs by Barthe et al. [2011] and similar to the two-program weakest-precondition calculus in [Felsing et al., 2014, Kiefer et al., 2016]. The principle idea behind the modality is that $[\,\mathtt{p} \between \mathtt{q}\,]@(obs,use)\phi$ holds if the programs $\mathtt{p},\mathtt{q}$ behave equivalently w.r.t. the program variables in *obs*. The formula ϕ is used as postcondition for program p such that the weak bisimulation modality implies the 'ordinary' modality $[\mathtt{p}]\phi$. Initially, formula ϕ is chosen as *true*. Only when handling loops, to increase precision by means of a loop invariant, other formulas can appear for ϕ.

In the verification-based compilation process outlined in the following, the bisimulation modality serves two purposes:

1. It guides the generation of compiled code after symbolic execution.
2. It allows the formal equivalence verification between source code and compilation result afterwards.

Before looking at the formal semantics of the modality and stating the calculus rules performing these tasks, we will give a brief overview over the compilation process. The initial input is the source program p and the equivalence criterion *obs*; this can, for example, be the set that only contains the variable holding the returned value if the result-equivalence is the target property.

The process follows a two-step protocol. In the first step, the source program is symbolically executed. This can be done using rules corresponding to ones of the calculus presented in Chapter 3, in a fashion similar to the symbolic execution debugger outlined in Section 11.2. It starts from the modality $[\,\mathtt{p} \between Q_1\,]@(obs,U_1)true$ with Q_1 and U_1 placeholder meta variable symbols which have no impact in the first phase. In the second phase, a compilation algorithm will fill these gaps starting from the leaves of the symbolic execution tree such that every step is one for which the calculus for the bisimulation modality has a rule.

The result is a closed proof tree with root $[\,\mathtt{p} \between \mathtt{q}\,]@(obs,use)true$ for some program q synthesized during the second phase. The proof guarantees us that q is equivalent to the input program p as far as the observations in *obs* are concerned.

To explain the meaning of the likewise synthesized *use*, we first introduce the set $usedVar(s,\mathtt{p},obs)$ capturing precisely those program variables whose value influences the final value of an observable location $l \in obs$ after executing p in a state s.

Definition 14.3 (Used program variable). Let s be a (Kripke) state (see Section 3.3.1).

A variable $v \in \text{ProgVSym}$ is *used by program $\mathtt{p}$ from s* with respect to variable set *obs* if there is a program variable $l \in obs$ such that

$$s \models \forall v_l; \big((\langle \mathtt{p} \rangle l \doteq v_l) \rightarrow \exists v_0; \{v := v_0\}\langle \mathtt{p} \rangle l \not\doteq v_l\big) \ .$$

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

A program variable v is used if and only if there is an interference with a location contained in *obs*, i.e., the value of v influences at least the value of one observed variable. Conversely, this means that if two states coincide on the variables in *use*, then the result states after the execution of p coincide on the variables in *obs*.

If two states s, s' coincide on the variables in a set $set \subseteq \text{ProgVSym}$, we write $s \approx_{set} s'$.

Definition 14.4 (Weak bisimulation modality—semantics). Let $\mathtt{p}, \mathtt{q}$ be while-programs, $obs, use \in \text{ProgVSym}$, s a Kripke state. Then $s \models [\, \mathtt{p} \between \mathtt{q} \,]@(obs, use)\phi$ if and only if

1. $s \models [\mathtt{p}]\phi$
2. $use \supseteq usedVar(s, \mathtt{q}, obs)$
3. for all $s \approx_{use} s'$ and $(s,t) \in \rho(\mathtt{p})$, $(s',t') \in \rho(\mathtt{q})$, we have $t \approx_{obs} t'$.

The formula $[\, \mathtt{p} \between \mathtt{q} \,]@(obs, use)\phi$ holds if the behaviors of $\mathtt{p}$ and $\mathtt{q}$ are equivalent w.r.t. the program variables contained in the set *obs*, and the set *use* contains all program locations and variables that may influence the value of any program variable or location contained in *obs* or the truth value of ϕ.

In the compilation scenario, $\mathtt{p}$ is the source program and $\mathtt{q}$ the created target program, hence validity of the formula ensures that the compilation is correct w.r.t. the equivalence criterion *obs*.

Bisimulation modalities can be embedded into sequents like $\Gamma \Longrightarrow \mathscr{U}[\, \mathtt{p} \between \mathtt{q} \,]@(obs, use)\phi, \Delta$, and the sequent calculus rules for the bisimulation modality are of the following form:

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

As mentioned earlier, application of the bisimulation rules is a two step process:

Step 1: Symbolic execution of source program $\mathtt{p}$ as usual using rules obtained from the ones in Chapter 3. The equivalence criterion *obs* is propagated from one modality to its children in the proof tree. In every arising modality, the second program parameter and the use set are filled with distinct meta-level placeholder symbols. The observable location sets obs_i are propagated and contain those

variables on which the two programs have to coincide. Intuitively, the variables mentioned here are protected in the sense that information about the value of these variables must not be thrown away during the symbolic execution step as the synthesized program will have to maintain their value.

Step 2: Synthesis of the target program q and used variable set *use* from $\mathtt{q}_i$ and use_i by applying the rules in a leave-to-root manner. Thus the placeholder symbols are instantiated. Starting with a leaf node, the program is generated until branching node is reached where the generation stops. The synthesis continues in the same fashion with the remaining leaves until programs for all subtrees of a branching node have been generated. Then these programs are combined according to the rule applied on the branching node.

For instance, in case of an if-then-else statement, first the then-branch and then else-branch are generated before synthesizing the corresponding conditional statement in the target program (see rule ifElse). Note that, in general, the order of processing the different branches of a node matters, for instance, in case of the loopInvariant the program for the branch that deals with program after the loop has to be synthesized before the loop body (as the latter's set of observable variable depends on those used on the other branch).

We explain some of the rules in details.

$$\textsf{emptyBox}\ \frac{\Gamma \Longrightarrow \mathscr{U}\,\phi, \Delta}{\Gamma \Longrightarrow \mathscr{U}\,[\ \{\}\ \langle\!\rangle\ \{\}\ ]@(obs, obs)\phi, \Delta}$$

The emptyBox rule is the starting point of program transformation in each sequential block. The location set *use* is set to *obs*.

assignment

$$\frac{\Gamma \Longrightarrow \mathscr{U}\,\{\mathtt{l} := \mathtt{r}\}[\ \omega\ \langle\!\rangle\ \overline{\omega}\ ]@(obs, use)\phi, \Delta}{\begin{pmatrix} \Gamma \Longrightarrow \mathscr{U}\,[\ \mathtt{l{=}r;}\omega\ \langle\!\rangle\ \mathtt{l{=}r;}\overline{\omega}\ ]@(obs, use \setminus \{\mathtt{l}\} \cup \{\mathtt{r}\})\phi, \Delta & \text{if } \mathtt{l} \in use \\ \Gamma \Longrightarrow \mathscr{U}\,[\ \mathtt{l{=}r;}\omega\ \langle\!\rangle\ \overline{\omega}\ ]@(obs, use)\phi, \Delta & \text{otherwise} \end{pmatrix}}$$

The assignment rule above comes in two variants. In the symbolic execution phase (first step) both are identical. The difference between both comes to play in the program synthesis phase (second step), i.e., when we instantiate the meta variables for the program and the used variable set.

In the second step, we check if the program variable l is contained in the *use* set of the premiss, i.e., the variable has been potentially read by the original program after the assignment. If l is read later-on, then the assignment of the original program (left compartment of the bisimulation modality) is generated for the specialized program. Otherwise the assignment is not generated for the specialized program.

In addition, the used variable set has to be updated, if the assignment was generated. The update used variable set removes first the variable on the left-hand side (l) as it is assigned a new value, and hence, the old value of l is unimportant from that time on. Thereafter, the variable r on the right-hand side is added to the used variable set

(as we read from it) which ensures that the following program syntheses steps will ensure that the correct value of `r` is computed. The order of the removal and addition is of importance as can be seen for the assignment `l=l;` where the computed used variable set must contain variable `l`.

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

(with `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 Section 14.1, 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.

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

(with `b` a Boolean program variable and *inv* a first-order formula)

The loop invariant rule has, as expected, three premises like in other appearances in this book. Here we are interested in compilation of the analyzed program rather than in proving its correctness. Therefore, it would be sufficient to use *true* as a trivial loop invariant. In this case, the first premise ensuring that the loop invariant

is initially valid contributes nothing to the program compilation process and can be ignored (if *true* is used as invariant then it holds trivially).

Using a stronger loop variant allows the synthesis algorithm to be more precise since the context on the sequent then contains more information which can be exploited during program synthesis.

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 14.6.

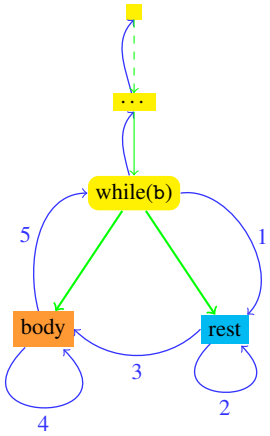


Figure 14.6 Work flow of synthesizing loop

Now we show the program transformation in action.

Example 14.5. Given observable locations *obs*={x}, we perform program transformation for the following program.

Java

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

```
  x = y + z;
  y = x;
  x = y + 2;
}
```

Java

In the first phase, we do symbolic execution using the extended sequent calculus from above We use placeholders sp_i to denote the program to be generated, and placeholders use_i to denote the used variable set. To ease the presentation, we omit postcondition ϕ, as well as the context 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 14.7.

$$\frac{\mathcal{U}_1\texttt{b} \Longrightarrow \mathcal{U}_1[\,\texttt{y=z++;}\ldots \Diamond\, sp_2\,]@(\{\texttt{x}\},use_2) \quad \mathcal{U}_1\neg\texttt{b} \Longrightarrow \mathcal{U}_1[\,\texttt{z=1;}\ldots \Diamond\, sp_3\,]@(\{\texttt{x}\},use_3)}{\frac{\Longrightarrow \{\texttt{y} := \texttt{y+z}\}[\,\texttt{if(b)\{...\}else\{...\}} \Diamond\, sp_1\,]@(\{\texttt{x}\},use_1)}{\Longrightarrow [\,\texttt{y = y + z;...} \Diamond\, sp_0\,]@(\{\texttt{x}\},use_0)}}$$

Figure 14.7 Symbolic execution tree until conditional

Now the symbolic execution tree splits into two branches. $\mathcal{U}_1$ denotes the update computed in the previous steps: $\{\texttt{y} := \texttt{y+z}\}$. We first concentrate on the `then` branch, where the condition `b` is *True*. The first active statement `y=z++;` is a complex statement. We decompose it into three simple statements using the postInc rule. 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 14.8.

$$\frac{\mathcal{U}_1\texttt{b} \Longrightarrow \mathcal{U}_1\{\texttt{t} := \texttt{z}\}\{\texttt{z} := \texttt{z+1}\}\{\texttt{y} := \texttt{t}\}\{\texttt{x} := \texttt{z}\}\phi}{\frac{\mathcal{U}_1\texttt{b} \Longrightarrow \mathcal{U}_1\{\texttt{t} := \texttt{z}\}\{\texttt{z} := \texttt{z+1}\}\{\texttt{y} := \texttt{t}\}\{\texttt{x} := \texttt{z}\}[\,\texttt{\{\}} \Diamond\, sp_8\,]@(\{\texttt{x}\},use_8)}{\frac{\mathcal{U}_1\texttt{b} \Longrightarrow \mathcal{U}_1\{\texttt{t} := \texttt{z}\}\{\texttt{z} := \texttt{z+1}\}\{\texttt{y} := \texttt{t}\}[\,\texttt{x=z;} \Diamond\, sp_7\,]@(\{\texttt{x}\},use_7)}{\frac{\mathcal{U}_1\texttt{b} \Longrightarrow \mathcal{U}_1\{\texttt{t} := \texttt{z}\}\{\texttt{z} := \texttt{z+1}\}[\,\texttt{y=t;}\ldots \Diamond\, sp_6\,]@(\{\texttt{x}\},use_6)}{\frac{\mathcal{U}_1\texttt{b} \Longrightarrow \mathcal{U}_1\{\texttt{t} := \texttt{z}\}[\,\texttt{z=z+1; y=t;}\ldots \Diamond\, sp_5\,]@(\{\texttt{x}\},use_5)}{\frac{\mathcal{U}_1\texttt{b} \Longrightarrow \mathcal{U}_1[\,\texttt{int t=z; z=z+1; y=t;}\ldots \Diamond\, sp_4\,]@(\{\texttt{x}\},use_4)}{\mathcal{U}_1\texttt{b} \Longrightarrow \mathcal{U}_1[\,\texttt{y=z++;}\ldots \Diamond\, sp_2\,]@(\{\texttt{x}\},use_2)}}}}}}$$

Figure 14.8 Symbolic execution tree of then branch

Now the source program is empty, so we can start generating a program at this node. By applying the emptyBox rule in the other direction, we get sp_8 as `{}`

(empty program) and use_8={x}. The next rule application is assignment. Because x $\in$ use_8, the assignment `x = z;` is generated and the used variable set is updated by removing x but adding z. So we have sp_7: `x = z;` and use_7={z}. In the next step, despite another assignment rule application, no statement is generated because 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: `z = z + 1; x = z;` and use_2={z}. So `z = z + 1; x = z;` is the program synthesized 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 14.9. In the second phase, the program is synthesized after applying a sequence of assignment rules. The else branch is sp_3:

```
z = 1; x = y + z; y = x; x = y + 2; ,
```

with use_3={y}.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathscr{U}_1 \neg \mathtt{b} \Longrightarrow \mathscr{U}_1\{\mathtt{z:=1}\}\{\mathtt{x:=y+z}\}\{\mathtt{y:=x}\}\{\mathtt{x:=y+2}\}\phi}{\mathscr{U}_1 \neg \mathtt{b} \Longrightarrow \mathscr{U}_1\{\mathtt{z:=1}\}\{\mathtt{x:=y+z}\}\{\mathtt{y:=x}\}\{\mathtt{x:=y+2}\}[\,\{\}\,\langle\!\rangle\, sp_{12}\,]@(\{\mathtt{x}\},use_{12})}}{\mathscr{U}_1 \neg \mathtt{b} \Longrightarrow \mathscr{U}_1\{\mathtt{z:=1}\}\{\mathtt{x:=y+z}\}\{\mathtt{y:=x}\}[\,\mathtt{x=y+2;}\,\langle\!\rangle\, sp_{11}\,]@(\{\mathtt{x}\},use_{11})}}{\mathscr{U}_1 \neg \mathtt{b} \Longrightarrow \mathscr{U}_1\{\mathtt{z:=1}\}\{\mathtt{x:=y+z}\}[\,\mathtt{y=x;\ldots}\,\langle\!\rangle\, sp_{10}\,]@(\{\mathtt{x}\},use_{10})}}{\mathscr{U}_1 \neg \mathtt{b} \Longrightarrow \mathscr{U}_1\{\mathtt{z:=1}\}[\,\mathtt{x=y+z;\ldots}\,\langle\!\rangle\, sp_9\,]@(\{\mathtt{x}\},use_9)}}{\mathscr{U}_1 \neg \mathtt{b} \Longrightarrow \mathscr{U}_1[\,\mathtt{z=1;\ldots}\,\langle\!\rangle\, sp_3\,]@(\{\mathtt{x}\},use_3)}$$

Figure 14.9 Symbolic execution tree of else branch

Now we have synthesized the program for both branches of the if-then-else statement. Back to the symbolic execution tree shown in Figure 14.7, we can build a conditional by applying the ifElse rule. The result is sp_1:

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

and use_1={b,z,y}. After a final assignment rule application, the program generated is shown in Listing 14.1.

Remark 14.6. 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, dead-code elimination, safe field access and type inference (Section 14.1.2). It will result in a more optimized program.

Example 14.7. We specialize the program shown in Example 14.5. In the first phase, symbolic execution is interleaved with simple partial evaluation actions.

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

Listing 14.1 The generated program for Example 14.5

In the first two steps of symbolic execution until conditional, no partial evaluation is involved. The resulting symbolic execution tree is identical to that shown in Figure 14.7.

There are 2 branches in the symbolic execution tree. Symbolical execution of the `then` branch is the same as in Example 14.5. It builds the same symbolic execution tree (Figure 14.8).

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

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

$$\mathscr{U}_1 \neg \mathtt{b} \Longrightarrow \mathscr{U}_1 \{\mathtt{z} := 1\}\{\mathtt{x} := \mathtt{y+1}\}\{\mathtt{y} := \mathtt{x}\}\{\mathtt{x} := \mathtt{x+2}\}@(\{\mathtt{x}\}, \_)$$

$$\mathscr{U}_1 \neg \mathtt{b} \Longrightarrow \mathscr{U}_1 \{\mathtt{z} := 1\}\{\mathtt{x} := \mathtt{y+1}\}\{\mathtt{y} := \mathtt{x}\}\{\mathtt{x} := \mathtt{x+2}\}[\ \langle\rangle\ sp_{12}\ ]@(\{\mathtt{x}\}, use_{12})$$

$$\mathscr{U}_1 \neg \mathtt{b} \Longrightarrow \mathscr{U}_1 \{\mathtt{z} := 1\}\{\mathtt{x} := \mathtt{y+1}\}\{\mathtt{y} := \mathtt{x}\}[\ \mathtt{x=x+2;}\ \langle\rangle\ sp_{11}\ ]@(\{\mathtt{x}\}, use_{11})$$

$$\mathscr{U}_1 \neg \mathtt{b} \Longrightarrow \mathscr{U}_1 \{\mathtt{z} := 1\}\{\mathtt{x} := \mathtt{y+1}\}[\ \mathtt{y=x;} \ldots\ \langle\rangle\ sp_{10}\ ]@(\{\mathtt{x}\}, use_{10})$$

$$\mathscr{U}_1 \neg \mathtt{b} \Longrightarrow \mathscr{U}_1 \{\mathtt{z} := 1\}[\ \mathtt{x=y+1;} \ldots\ \langle\rangle\ sp_9\ ]@(\{\mathtt{x}\}, use_9)$$

$$\mathscr{U}_1 \neg \mathtt{b} \Longrightarrow \mathscr{U}_1 [\ \mathtt{z=1;} \ldots\ \langle\rangle\ sp_3\ ]@(\{\mathtt{x}\}, use_3)$$

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

Combining both branches, we finally get the specialized version of the original, shown in Listing 14.2.

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

Listing 14.2 The generated program for Example 14.7

Compared to the result shown in Listing 14.1, we generated a more optimized program by interleaving partial evaluation actions during symbolic execution phase. Further optimizations can be achieved by involving updates during program generation, which are discussed in [Ji, 2014].

14.3 Implementation and Evaluation

We have a prototype implementation of the program transformation framework named PE-KeY introduced in this chapter.

We applied our prototype partial evaluator also on some examples stemming from the JSpec test suite [Schultz et al., 2003]. One of them is concerned with the computation of the power of an arithmetic expression, as shown in Figure 14.10.

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:

```
power = new Power(y, new op(), neutral).raise(x)
```

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

```
power = new Power(y, new Mult(), 1).raise(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:

$$\texttt{y} \doteq 16 \rightarrow$$
$$[\texttt{power=new Power(y,new Mult(),1).raise(x);} \Diamond\, \texttt{sp}_{res}\,]@(obs, use)post$$

with *post* denoting an unspecified predicate which can neither be proven nor disproved. PE-KeY then executes the program symbolically and extracts the specialized program $\texttt{sp}_{res}$ as `power = (...((x*x)*x)*...)*x;` (or `power = x`^{16}). The

```
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 14.10 Source code of the `Power` example as found in the JSpec suite

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.

14.4 Conclusion

In this chapter we described how symbolic execution and thus verification can benefit from interleaving partial evaluation and symbolic execution steps. This interleaving results in smaller, less redundant proof and symbolic execution trees making these easier to comprehend. This is advantageous for both manual interaction with the prover itself and for code reviews/debugging using the symbolic execution debugger (see Chapter 11).

We also presented how to integrate verifiably correct compilation of programs within our verification framework based on JavaDL. We showcased our application along the implementation of a partial evaluator that produces verifiably correct specialized programs. Our approach can make use of the full power of our verifier and thus produce optimized [Ji, 2014] programs. [Ji and Hähnle, 2014] adapt the approach to implement an information flow analysis.