# Sound Deductive Compilation

Ran Ji, and Reiner Hähnle

Department of Computer Science
Technische Universität Darmstadt, Germany
{ran,haehnle}@cs.tu-darmstadt.de

**Abstract.** Compiler verification is difficult and expensive. Instead of formally verifying a compiler, we introduce a sound deductive compilation approach, whereby verified bytecode is generated based on symbolic execution of source code embedded in a program logic. The program logic guarantees a weak bisimulation relation between the generated bytecode and the original source code relative to a set of observable locations. The framework is instantiated for Java source and bytecode. The compilation process is fully automatic and first-order solvers are employed for bytecode optimization.

## 1 Introduction

Many formal software verification techniques focus on ensuring the correctness of source programs. This assumes, however, that compilers preserve the behavior of the source program, otherwise, they might introduce errors to the already verified source program. Errors introduced by compilers are notoriously difficult to expose and track down. *Compiler verification* [1] is a possible technique to ensure the correctness of compilers. As it involves reasoning about compiler implementations and the behavior of the compiler for an infinite number of programs and their translations, it is very expensive.

In this paper we present a method to guarantee correct compilation to bytecode for programs written in (a subset of) Java. Instead of formally verifying a Java compiler, we propose a sound deductive compilation approach to generate correct Java bytecode from Java source code. It works in two phases, verification and synthesis. During the verification phase, Java source code is symbolically executed by sequent calculus rules in a program logic. First-order reasoning is involved for a precise analysis of variable dependencies, aliasing, and elimination of infeasible execution paths, which may result in optimized bytecode generation later on. For sound compilation we need to ensure that the generated Java bytecode is *weakly bisimilar* to the original Java source code relative to a set of observable locations (e.g., return variables). In previous work on sound program transformation [2,3], a suitable weak bisimulation relation for two source programs was defined and used. We extend this work by introducing a mapping function that relates Java source code and bytecode. To synthesize Java bytecode in a program logic, we extend the sequent calculus rules to include the *weak bisimulation modalities* relating Java source code and bytecode. Then a leaves-to-root traversal of the symbolic execution tree resulting from the verification

phase synthesizes bytecode by backward application of sequent calculus rules with bisimumation modalities.

The soundness of the compilation process is guaranteed by soundness of the underlying program logic, hence, no compiler verification is necessary. Bytecode generation happens immediately after source code verification on the same proof object. Therefore, it is possible to combine formal verification and sound compilation into a single process, whereas those two aspects are normally considered separately, using different techniques and tools.

The paper is organized as follows: Sect. 2 defines the programming language and program logic; Sect. 3 presents the sequent calculus rules that are used for symbolic execution in the Java source program verification phase; Sect. 4 reviews the bisimulation modality for two Java source programs; Sect. 5 defines the mapping function and presents Java bytecode generation; Sect. 6 discusses related work; Sect. 7 concludes and lists future work. For space reasons, most proofs are omitted. Proofs of results in Sect. 4 can be found in [3].

## 2 Language and Logic

### 2.1 Programming Language

We use a programming language called PL that is a subset of Java. It has classes, objects, fields, and method polymorphism (but not method overloading). Generic types, exceptions, multi-threading, floating points, and garbage collection are not supported. The types of PL are the types derived from class declarations, the type `int` of mathematical integers ($\mathbb{Z}$), and the standard Boolean type `boolean`.

A PL 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 $Cl := (cname, scname_{opt}, fld, mtd)$ consists of (i) a class name $cname$ unique in `p`, (ii) the name of its superclass $scname$ (only omitted for $cname =$ `Object`), and (iii) a list of field $fld$ and method $mtd$ declarations. The syntax coincides with that of Java. The only features lacking from Java are constructors and initialization blocks. We agree on the following conventions: if not stated otherwise, any sequence of statements is viewed as if it were the body of a static, void method declared in a class `Default` with no fields.

In a PL program `p`, a complex statement can be decomposed into a sequence of simpler statements without changing the meaning of `p`. For example, statement `y = z ++;` can be decomposed into `int t = z; z = z + 1; y = t;`, where `t` is a *fresh* variable, not used anywhere else. These *simple statements* have at most one source of side effect each, which can be a non-terminating expression (such as a null pointer access), a method call, or an assignment to a location. They are essential to compute variable dependencies and simplify symbolic states during symbolic execution later on.

### 2.2 Program Logic

Our program logic is *dynamic logic (DL)* [4]. We consider deterministic programs, hence, a program `p` executed in state $s$ *either* terminates and reaches

exactly *one* final state *or* it does not terminate and no final state reached. The dynamic logic over PL programs is called PL-*DL*. Its signature depends on a *context* PL *program* $\mathcal{C}$.

**Definition 1 (PL-Signature $\Sigma_\mathcal{C}$).** *A* signature $\Sigma_\mathcal{C} = (\mathsf{Srt}, \preceq, \mathsf{Pred}, \mathsf{Func}, \mathsf{LgV})$ *consists of: (i) a set of names* $\mathsf{Srt}$ *called* sorts *containing at least one sort for each primitive type and one for each class* $Cl$ *declared in* $\mathcal{C}$*; (ii) a* partial subtyping order $\preceq\colon \mathsf{Srt} \times \mathsf{Srt}$ *that models the subtype hierarchy of* $\mathcal{C}$ *faithfully; (iii) infinite sets of* predicate symbols $\mathsf{Pred} := \{p : T_1 \times \ldots \times T_n \mid T_i \in \mathsf{Srt}\}$ *and* function symbols $\mathsf{Func} := \{f : T_1 \times \ldots \times T_n \to T \mid T_i, T \in \mathsf{Srt}\}$ *for each* $n \in \mathbb{N}$*. We call* $\alpha(p) = T_1 \times \ldots \times T_n$ *and* $\alpha(f) = T_1 \times \ldots \times T_n \to T$ *the* signature *of the predicate/function symbol; (iv) a set of* logical variables $\mathsf{LgV} := \{x : T \,|\, T \in \mathsf{Srt}\}$*.* $\mathsf{Func}$ *is divided into disjoint subsets* $\mathsf{Func}_r \cup \mathsf{PV} \cup \mathsf{Fld}$*:*

- rigid *function symbols* $\mathsf{Func}_r$*, which do not depend on the current state of program execution;*
- program variables $\mathsf{PV} = \{\mathtt{i}, \mathtt{j}, \ldots\}$*, which are non-rigid constants;*
- field *function symbols* $\mathsf{Fld}$*, where for each field* $\mathtt{a}$ *of type* $T$ *declared in class* $Cl$ *a function* $\mathtt{a}@Cl : Cl \to T \in \mathsf{Fld}$ *exists. We omit* $@C$ *if no ambiguity arises.*

We distinguish between *rigid* and *non-rigid* predicate and function symbols. The semantics of rigid symbols does not depend on the current state of program execution, while non-rigid symbols are state-dependent.

Terms $t$ and formulas $\phi$ are defined as usual and their definitions are omitted for brevity. We use *updates* $u$ to describe symbolic state changes by means of an explicit substitution. An *elementary update* of the form $\mathtt{i} := t$ or $t.\mathtt{a} := t$ is a pair of location and term. Updates have *static single assignment (SSA)* form and the same meaning as (simple) assignments. Elementary updates are composed into *parallel updates* $u_1 \| u_2$ that work like simultaneous assignments. Updates $u$ are defined by the grammar $u ::= \mathtt{i} := t \mid t.\mathtt{a} := t \mid u \parallel u \mid \{u\}u$ (where $\mathtt{a} \in \mathsf{Fld}$), together with the usual well-typedness conditions. Updates applied on terms (formulas), written $\{u\}t$ ($\{u\}\phi$), are again terms (formulas). We use the symbol $\mathcal{U}$ to represent updates of the form $\{u\}$ and $\{u_1\}\ldots\{u_n\}$.

Terms, formulas and updates are evaluated in a PL-DL Kripke structure:

**Definition 2 (Kripke structure).** *A PL-DL Kripke structure* $\mathcal{K}_{\Sigma_{\mathrm{PL}}} = (\mathcal{D}, I, S)$ *consists of (i) a set of elements* $\mathcal{D}$ *called domain, (ii) an interpretation* $I$ *with:*

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

*And (iii) a set of states* $S$ *assigning meaning to non-rigid function symbols: let* $s \in S$ *then* $s(\mathtt{a}@Cl) : \mathcal{D}_{Cl} \to \mathcal{D}_T,\ \mathtt{a}@Cl : Cl \to T \in \mathsf{Fld}$ *and* $s(\mathtt{i}) : \mathcal{D}_T,\ \mathtt{i} \in \mathsf{PV}$*. The pair* $D = (\mathcal{D}, I)$ *is called a first-order structure.*

A *variable assignment* $\beta : \mathsf{LgV} \to \mathcal{D}_T$ maps a first-order variable $x : T$ to its domain $\mathcal{D}_T$. A term, formula or update is evaluated relative to a given first-order structure $D = (\mathcal{D}, I)$, a state $s \in S$ and a variable assignment $\beta$, 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 $val_{D,s,\beta}(t) \in \mathcal{D}_T$; (ii) every formula $\phi$ 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 $val_{D,s}(st) \subseteq 2^S$.

As PL is deterministic, all sets of states or state-value pairs have at most one element. The semantics of terms, formulas, expressions and statements is the same as in Java. Details are in [3].

## 3 Source Code Verification

We verify PL source code following the KeY [5] approach, where a PL program is symbolically executed. In this paper, we do not emphasize on proving correctness (for which we refer to [5]), but introduce only those concepts of symbolic execution and the sequent calculus that are needed for bytecode generation.

Symbolic execution of a PL program is performed by application of sequent calculus rules. Soundness of the rules ensures validity of provable PL-DL formulas in a program verification setting [5].

A *sequent* is a pair of sets of formulas $\Gamma = \{\phi_1, \ldots, \phi_n\}$ (antecedent) and $\Delta = \{\psi_1, \ldots, \psi_m\}$ (succedent) of the form $\Gamma \Longrightarrow \Delta$. Its semantics is defined by the formula $\bigwedge_{\phi \in \Gamma} \phi \to \bigvee_{\psi \in \Delta} \psi$. A *sequent calculus rule* has one conclusion and zero or more premises. It is applied to a sequent $s$ by matching its conclusion against $s$. The instantiated premises are then added as children of $s$.

Our PL-DL sequent calculus behaves as a symbolic interpreter for PL. A *sequent* for PL-DL is always of the form $\Gamma \Longrightarrow \mathcal{U}[\mathtt{p}]\phi, \Delta$. The box modality $[\cdot]$ indicates partial correctness: *if* p is executed and terminates *then* in all reached final states $\phi$ holds. During symbolic execution performed by the sequent rules (see Fig. 1), the antecedents $\Gamma$ accumulate path conditions and may contain preconditions. Updates $\mathcal{U}$ record the symbolic value at a point of execution and $\phi$ represents postconditions. The update $\mathcal{V}_{mod}$ in the loopInvariant rule skolemizes the program locations that might be changed in the loop body. This is necessary, because the invariant must hold in any state. When the program is fully executed, we obtain a set of first-order formulas (one for each symbolic execution path), to be proven or disproven by first-order reasoning.

During symbolic execution, complex statements are decomposed into simple statements. First-order reasoning and interleaved partial evaluation [6] help to simplify the target program on-the-fly. Symbolic execution 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.

$$\textsf{emptyBox}\ \frac{\Gamma \Longrightarrow \mathcal{U}\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[]\phi, \Delta} \qquad \textsf{assignment}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{l} := \mathtt{r}\}[\omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{l} = \mathtt{r}; \omega]\phi, \Delta}$$

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

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

**Fig. 1.** Selected sequent calculus rules (further rules are in [3,5]).

3. Apply the sequent calculus rule corresponding to the active statement.
4. Simplify the resulting updates 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. Goto 1.

*Example 1.* 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 continues with `p` $\omega$.
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*. More techniques for *partial evaluation* can be integrated into symbolic execution [6].
3. $\Gamma \Rightarrow \{\mathtt{o1.a} := \mathtt{v1} \| \ldots\}[\mathtt{o2.a = v2;}\ \omega]\phi$: After executing $\mathtt{o2.a} = \mathtt{v2}$, the *alias* is analyzed: (i) if $\mathtt{o2} = \mathtt{null}$ is true the program does not terminate; (ii) else, if $\mathtt{o2} = \mathtt{o1}$ holds, the value of `o1.a` in the update is overridden and the new update is $\{\mathtt{o1.a} := \mathtt{v2} \| \ldots \| \mathtt{o2.a} := \mathtt{v2}\}$; (iii) else the new update is $\{\mathtt{o1.a} := \mathtt{v1} \| \ldots \| \mathtt{o2.a} := \mathtt{v2}\}$. Neither of (i)–(iii) might be provable and then symbolic execution splits into these three cases when encountering a potentially aliased object access.

The result of symbolic execution for a PL program `p` is a *symbolic execution tree (SET)*, as illustrated in Fig. 2. Note that, here we did not show the part that does not contain any PL program, e,g, the (init) branch obtained after applying the loopInvariant rule. Complete SETs are finite 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 remaining program to be executed. Every child node in an SET is generated by rule application from its parent. A *branching node* represents a statement whose execution causes branching, e.g., conditional, loop, etc. We call a *sequential block (SB)* a maximal program fragment in an SET that is symbolically executed without branching. The SET in Fig. 2 has 7 SBs $bl_0, \ldots, bl_6$.

## 4 Weak Bisimulation and Program Transformation

Here we review the concepts of weak bisimulation relations of PL programs, and the extended sequent calculus rules for program transformation.

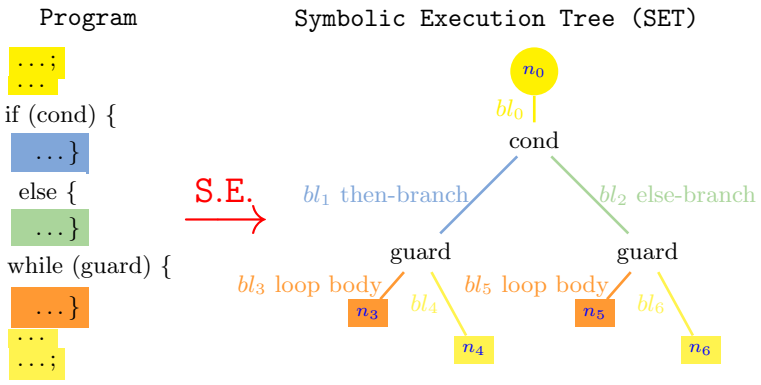


**Fig. 2.** Symbolic execution tree with loop invariant applied.

The structure of an SET makes it possible to generate a program through its leaves-to-root traversal, by applying the sequent calculus rules reversely step-by-step. This requires to extend the sequent calculus rules with means for program synthesis. Obviously, the generated program should behave exactly as the original one, at least for the *observable locations* (e.g., return variables).

### 4.1 Weak Bisimulation Relation of Programs

**Definition 3 (Location sets, observation equivalence).** *A* location set *is a set containing program variables* x *and attribute expressions* $o.\mathtt{a}$ *(*$\mathtt{a} \in \mathsf{Fld}$ *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* $obs \subseteq loc$*, 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*.*

A *transition relation* $\longrightarrow: \Pi \times S \times S$ relates two states $s$, $s'$ by a program p iff 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 simple statements during symbolic execution, we can assume that a program consists of simple statements.

**Definition 4 (Observable and internal statement/transition).** *Consider states* $s, s'$*, a simple statement* 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* $\xrightarrow{\mathtt{sSt}}_{obs}$*. Otherwise,* sSt *is called an* internal statement *and* $\longrightarrow$ *an* internal transition*, written* $\longrightarrow_{int}$*.*

Assume an observable transition $s \xrightarrow{\mathtt{sSt}}_{obs} s'$ changes the evaluation of some location $ol \in obs$ in state $s'$. The observable locations $obs_1$ in state $s$ should also contain the locations $ol_1$ that are *read* by $ol$, since changes to $ol_1$ can lead to a change of $ol$ in the final state $s'$.

*Example 2.* 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 5 (Weak transition).** *Given observable locations obs, the transition relation* $\Longrightarrow_{int}$ *is the reflexive, transitive closure of* $\longrightarrow_{int}$*. The transition relation* $\overset{\mathtt{sSt}}{\Longrightarrow}_{obs}$ *is the composition of the relations* $\Longrightarrow_{int}$, $\overset{\mathtt{sSt}}{\longrightarrow}_{obs}$ *and* $\Longrightarrow_{int}$*. The* weak transition $\overset{\widehat{\mathtt{sSt}}}{\Longrightarrow}_{obs}$ *represents either* $\overset{\mathtt{sSt}}{\Longrightarrow}_{obs}$*, if* sSt *observable, or* $\Longrightarrow_{int}$ *otherwise.*

**Definition 6 (Weak bisimulation for states).** *Given two programs* $\mathtt{p}_1, \mathtt{p}_2$ *and observable locations obs, obs′, let* $\mathtt{sSt}_1$ *be a simple statement and* $s_1, s_1'$ *two program states of* $\mathtt{p}_1$*, and* $\mathtt{sSt}_2$ *be a simple statement and* $s_2, s_2'$ *are two program states of* $\mathtt{p}_2$*. A relation* $\approx$ *is a* weak bisimulation *for states if and only if* $s_1 \approx_{obs} s_2$ *implies:*

- *if* $s_1 \overset{\widehat{\mathtt{sSt}_1}}{\Longrightarrow}_{obs'} s_1'$*, then* $s_2 \overset{\widehat{\mathtt{sSt}_2}}{\Longrightarrow}_{obs'} s_2'$ *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 7 (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* $0 \leq i \leq n$ *and* $0 \leq j \leq m$*.*

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*. It directly implies the lemma below that relates the weak bisimulation relation of programs to a *big-step semantics*.

**Lemma 1.** *Let* $\mathtt{p}, \mathtt{q}$ *be programs, obs a set of observable locations. Then* $\mathtt{p} \approx_{obs} \mathtt{q}$ *if and only if* $val_{D,s}(\mathtt{p}) \approx_{obs} val_{D,s}(\mathtt{q})$ *for any first-order structure D, state s.*

### 4.2 The Weak Bisimulation Modality

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

**Definition 8 (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* $\phi$ *be a* PL*-DL formula,* $\mathtt{p}, \mathtt{q}$ PL *programs, and* $obs, use$ *location sets such that* $pv(\phi) \subseteq obs$*, where* $pv(\phi)$ *is the set of all program variables occurring in* $\phi$*, then* $[\ \mathtt{p}\ \between\ \mathtt{q}\ ]@(obs, use)\phi$ *is also a* PL*-DL formula. An* extended sequent *based on the bisimulation modality has the form* $\Gamma \Longrightarrow \mathcal{U}[\ p\ \between\ q\ ]@(obs, use)\phi, \Delta$.

Next we define the location set $usedVar(s, \mathtt{p}, obs)$. Its purpose is to capture precisely those locations whose value influences the final value of an observable location $l \in obs$ after executing a program $\mathtt{p}$. Later we approximate this set by the set of all program variables in a program that are used before being redefined (i.e., assigned a new value).

**Definition 9 (Used program variable).** *A variable* $v \in \mathsf{PV}$ *is called* used by a program $\mathtt{p}$ *in state* $s$ *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 2, $\mathtt{z}$ is a used variable. We formalize the semantics of the weak bisimulation modality:

**Definition 10 (Weak bisimulation modality: semantics).** *With* $\mathtt{p}, \mathtt{q}$ PL*-programs,* $D, s, \beta$*, and obs, use as above, 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 2.** *Let obs be the set of all locations observable by* $\phi$ *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* $D$, $s$, $\beta$.

The following lemma shows the intended meaning of the used variable set *use*.

**Lemma 3.** *Let the extended sequent* $\Gamma \Longrightarrow \mathcal{U}[\ p\ \between\ q\ ]@(obs, use)\phi, \Delta$ *occur in a sequential block bl in a state* $s_1$*, where* $P$ *is the program corresponding to bl,* $p$ *is the program to be executed in bl in state* $s_1$*, and* $p'$ *is the program already executed in bl (see Fig. 3). Let* $Q$ *be the program to be generated in bl,* $q$ *the already generated program in bl, and* $q'$ *the remaining program to be generated in bl. The location set use are the* dynamically observable locations *for which the following relations hold: (i)* $p \approx_{obs} q$*; (ii)* $P \approx_{obs} Q$*; (iii)* $p' \approx_{use} q'$.

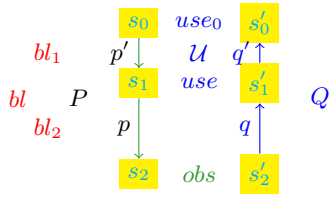


**Fig. 3.** Program in a sequential block.

### 4.3 Sequent Calculus Rules for Weak Bisimulation Modality

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

$$\frac{\begin{array}{c}\Gamma_1 \Longrightarrow \mathcal{U}_1[\ p_1\ \langle\!\rangle\ q_1\ ]@(obs_1, use_1)\phi_1, \Delta_1 \\ \cdots \\ \Gamma_n \Longrightarrow \mathcal{U}_n[\ p_n\ \langle\!\rangle\ q_n\ ]@(obs_n, use_n)\phi_n, \Delta_n\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\ p\ \langle\!\rangle\ q\ ]@(obs, use)\phi, \Delta}$$

An example is the following assignment rule:

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

Here, $\overline{\omega}$ represents the already generated program before the rule application. The *use* set contains all variables that may affect the values of observable locations in the final state. In the first case, when $\mathtt{l}$ is among those variables, we have to update the *use* set by removing $\mathtt{l}$ and adding $\mathtt{r}$ which is read by the assignment. Otherwise, we generate no code. The emptyBox rule, ifElse rule and loopInvariant rule are as follows ($\_$ denotes the place holder of *empty*):

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

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

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

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

The emptyBox rule starts the program generation, where *use* is instantiated as *obs*. We can begin generating a program when the original program is symbolically executed completely, but not when it is fully verified. So even if we cannot

verify the original program, we are still able to perform a sound program transformation. In the loopInvariant rule, *obs* in the (preserves) branch depends on the $use_1$ from the (use case) branch, so we need to synthesize the (use case) branch first when synthesizing a loop. The (init) branch does not contribute to program generation (that is why this branch is not shown in the SET in Fig. 2, Sect. 3). More extended sequent calculus rules are in [3].

To synthesize a program, one starts with a leaf node and generates the program within its SB first, e.g., $bl_3$, $bl_4$ in Fig. 2. These are combined by rules corresponding to statements that contain a SB, such as loopInvariant (containing $bl_3$ and $bl_4$). One continues with these steps recursively until the root is reached.

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

## 5 Bytecode Generation

The weak bisimulation modality [ p ≬ q ]@(*obs*, *use*) of Sect. 4.2 requires q to be a program in the same language as p, but this is not really necessary. We can just as well generate Java bytecode from Java source code (or PL to be precise), which realizes a Java (PL) compiler. The soundness of compilation is entailed by the sound bisimulation modality and extended sequent calculus rules.

We target the version of Java bytecode that can be executed by a *Java Virtual Machine (JVM)* [7]. 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`.

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 (see [7]) and focus on the relation between the semantics of PL source code and Java bytecode. This allows us to use most results from Sect. 4.

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 $\xi$ to relate PL source code to Java bytecode.

**Definition 11 (Mapping function).** *For a* PL *program, St is the set of statements, S is a set of states,* PV *is a set of program variables. And for Java byte-*

*code,* $\mathtt{Inst}$ *is the set of instructions,* $S_B$ *is a set of states,* $\mathsf{PV}_B$ *is a set of program variables. A mapping function* $\xi$ *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* $\mathtt{s} \in S$ *to an* $\mathtt{s_B} \in S_\mathtt{B}$. $\xi(\mathtt{s}) = \mathtt{s_B}$.
*(iii)* *every* $\mathtt{st} \in \mathtt{St}$ *to a sequence of instructions:* $\mathtt{inst_1} \cdots \mathtt{inst_n}$, *where for* $0 \leq i \leq n$ $\mathtt{inst_i} \in \mathtt{Inst}$ *and* $\xi(\mathtt{st}) = \mathtt{inst_1} \cdots \mathtt{inst_n}$.

$\xi^{-1}$ *is the* inverse *of* $\xi$.

Fig. 4 shows some PL statements and Java bytecode related by the mapping function $\xi$. We also maintain a *program counter* $pc$ (initially 0) to indicate the label of bytecode instructions, and $pc_i$ has the value of $pc + i$.

| **PL statement** | **Java bytecode** |
|---|---|
| `l=r` | `iload` $\xi$(`r`)<br>`istore` $\xi$(`l`) |
| $\mathtt{p_1}$;$\mathtt{p_2}$ | $\xi(\mathtt{p_1})$<br>$\xi(\mathtt{p_2})$ |
| `if`(b) {p} `else` {q} | `iload` $\xi$(`b`)<br>`ifeq` $pc_1$<br>$\xi$(`p`)<br>`goto` $pc_2$<br>$pc_1$: $\xi$(`q`)<br>$pc_2$: - |
| `while`(b) {p} | $pc_1$: `iload` $\xi$(`b`)<br>`ifeq` $pc_2$<br>$\xi$(`p`)<br>`goto` $pc_1$<br>$pc_2$: - |

**Fig. 4.** Mapping of PL programs to Java bytecode.

As a property of the mapping function $\xi$, the following lemma gives the relation of the semantics of PL programs to the semantics of Java bytecode. We assume Java bytecode is evaluated by an evaluation function *valb*, logic structure $D_B$ and in a state $s$. The actual representation of $D_B$ is not of importance.

**Lemma 4.** *Given the evaluation function val, the first-order structure* $D$ *and a state* $s \in S$ *of* PL *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 4 shows that instead of evaluating Java bytecode, we can evaluate its $\xi^{-1}$-mapped PL program. This allows us to define the weak bisimulation modality for a PL program and Java bytecode by adding a mapping function $\xi$ to Defs. 3-7 in Sect. 4.2. For example, the definition of weak bisimulation for PL program and Java bytecode is given below. The other definitions are analogous.

**Definition 12 (Weak bisimulation for PL program and Java bytecode).** *Let* $\mathtt{p}_1, \mathtt{p}_2$ *be two* PL *programs,* $\mathtt{q}$ *is Java bytecode, and* $\xi$ *is a mapping function such that* $\xi(\mathtt{p}_2) = \mathtt{q}$. *Assume obs, obs′ are observable locations, and* $\approx$ *is a weak bisimulation relation for states. Then* $\approx$ *is a* weak bisimulation *for a* PL *program* $\mathtt{p}_1$ *and Java bytecode* $\mathtt{q}$*, written* $\mathtt{p}_1 \approx_{obs} \mathtt{q}$*, if for the sequence of state transitions:*

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

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

*(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* $0 \leq i \leq n$ *and* $0 \leq j \leq m$.

**Definition 13 (Weak bisimulation modality for PL program and Java bytecode: syntax).** *The* bisimulation modality $[\,\mathtt{p} \between \mathtt{q}\,]@(obs, use)$ *is a modal operator providing compartments for a* PL *program* $\mathtt{p}$*, Java bytecode* $\mathtt{q}$ *and location sets obs and use. We extend our definition of formulas: Let* $\phi$ *be a* PL*-DL formula,* $\mathtt{p}$ *a* PL *program,* $\mathtt{q}$ *Java bytecode, and obs, use location sets such that* $pv(\phi) \subseteq obs$*, then* $[\,\mathtt{p} \between \mathtt{q}\,]@(obs, use)\phi$ *is also a* PL*-DL formula.*

**Definition 14 (Weak bisimulation modality for PL program and Java bytecode: semantics).** *For* PL*-programs* $\mathtt{p}$*,* $\mathtt{p}_1$ *and Java bytecode* $\mathtt{q}$*;* $D, s, \beta$*, and obs, use are as before,* $\xi$ *is a mapping function such that* $\xi(\mathtt{p}_1) = \mathtt{q}$*. 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* $val_{D,s}(\mathtt{p}) \approx_{obs} val_{D,s'}(\mathtt{p}_1) = valb_{D_B,\xi(s')}(q)$

The *used program variable* set $usedVar(s, \mathtt{p}, obs)$ is defined in the same way as in Def. 9.

The extended sequent calculus rules for Java bytecode generation can be defined based on the weak bisimulation modality for PL program and Java bytecode. In most cases, by changing the generated PL program part to its $\xi$-mapped Java bytecode in the rules presented in Sect. 4.3, we can obtain the rules for bytecode generation, as shown in Fig. 5. The symbol $\overline{\omega}$ represents the generated Java bytecode for PL program $\omega$, $\xi$ 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.

**Theorem 2.** *The extended sequent calculus rules given in Fig. 5 are sound.*

*Proof.* Follows from Theorem 1, Def. 11, and Lemma 4. □

By introducing a mapping function $\xi$ we avoid to use the semantics of Java bytecode directly but relate it to the semantics of PL programs, which results in a better integration of the new weak bisimulation modality with the ones introduced before. In fact, $\xi$ can also be viewed as the *compilation* function since it maps the source code to the bytecode. However, instead of operating on

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

assignment

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

ifElse

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

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

loopInvariant

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

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

**Fig. 5.** A collection of sequent calculus rules for generating Java bytecode.

the original source program like a standard compiler would do, $\xi$ is applied to the generated source code and the bytecode is generated based on that already specialized code. So it works as an *optimizing compiler*.

*Example 3.* Given suitable pre- and post-conditions $Pre$ and $Post$, one can formally verify the PL program shown in Fig. 6 on the left and, at the same time, compile it to Java bytecode. Assume that `cpn` = `FALSE` is known and contained in $Pre$, and the observable locations $obs$ is the return variable set {`tot`}.

This program might be part of a cashier system to calculate the total amount a customer has to pay (if buying `i` items at a price of 20 units). The total sum is stored in both `tot` and `atot`. If the customer can provide a coupon (`cpn`), then a reduction of 50 units is applied. Finally, the total cost is returned as `tot`.

To verify this program we perform symbolic execution, using the sequent calculus rules shown in Fig. 1. After a few applications of the assignment rule, we use the loopInvariant rule and continue. In the (use case) branch, we encounter the conditional. Knowing `cpn = FALSE`, by partial evaluation, we continue with the (else) branch. After fully executing the code in each branch, we do first-order reasoning steps to complete the verification. Now we focus on compilation, using the extended sequent calculus rules given in Fig 5. The resulting program is shown in Fig. 6 on the right.

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 `atot = 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.

```
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;
```

```
   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
```

**Fig. 6.** Program to be compiled into bytecode and generated bytecode.

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.

## 6 Related Work

Compiler verification has been a research topic for more than 40 years [8,9]. Since then, many proofs have been conducted, ranging from single-pass compilers for toy languages to sophisticated code optimizations [1]. Recently, the *CompCert*

project [10,11,12] 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 non-optimizing compiler from C0, a subset of C, directly to DLX assembly has been verified in Isabelle/HOL [13]. Like CompCert, it focuses on low-level details and proves a weak simulation theorem for sequential executions. The paper [14] 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. All of this work consists of monolithic, highly complex specifications and proofs that consumed person years of work. In contrast, our approach aims at fully automated, correct compilation on-the-fly of *one* given program at a time.

A "grand challenge" for computer science proposed by Hoare [15] is to achieve a "verifying compiler" that checks the correctness of a program along with compilation, just like a compiler performing type checking nowadays. Our approach has the same goal of verifying source code and compiled code within one process, but takes the opposite view: it performs sound compilation by a source program verification tool. Our approach may be viewed as a "compiling verifier".

## 7 Conclusion and Future Work

In this work, we presented a sound deductive compilation approach for programs written in a subset of Java. It generates Java bytecode following Java source code verification, so it is possible to ensure the correctness of both source code and bytecode within one process. The generated Java bytecode is more optimized than that obtained from line-by-line compilation. The soundness of compilation is guaranteed.

We plan to consolidate the implementation of the approach presented here, and to support more features of Java in the future. The bisimulation modality defined in Sect. 4.2 gives the opportunity to analyze information flow security problems. It can be integrated into the current work, so we can achieve a unified framework for program verification, sound compilation and information flow analysis.

## References


1. Dave, M.A.: Compiler verification: a bibliography. SIGSOFT Softw. Eng. Notes **28** (November 2003) 2–2
2. Ji, R., Hähnle, R., Bubel, R.: Program transformation based on symbolic execution and deduction. In: SEFM. LNCS, Springer (2013) 289–304
3. Ji, R., Hähnle, R., Bubel, R.: Program transformation based on symbolic execution and deduction. Technical Report CS-2013-0348, Technische Universität Darmstadt, Fachbereich Informatik (2013) `https://www.se.tu-darmstadt.de/fileadmin/user_upload/Group_SE/Page_Content/Group_Members/ran_ji/TUD-CS-2013-0348.pdf`.

4. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
5. Beckert, B., Hähnle, R., Schmitt, P., eds.: Verification of Object-Oriented Software: The KeY Approach. Volume 4334 of LNCS. Springer (2006)
6. Bubel, R., Hähnle, R., Ji, R.: Interleaving symbolic execution and partial evaluation. In: Post Conf. Proc. FMCO2009. LNCS, Springer-Verlag (2009)
7. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley (1997)
8. McCarthy, J., Painter, J.: Correctness of a compiler for arithmetic expressions. In: Mathematical Aspects of Computer Science, volume 19 of Proc. of Symposia in Applied Mathematics, American Mathematical Society (1967) 33–41
9. Milner, R., Weyhrauch, R.: Proving compiler correctness in a mechanized logic. In: Proc. 7th Annual Machine Intelligence Workshop, volume 7 of Machine Intelligence, Edinburgh University Press (1972) 51–72
10. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL. (2006) 42–54
11. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7) (2009) 107–115
12. Leroy, X.: A formally verified compiler back-end. J. Autom. Reasoning **43**(4) (2009) 363–446
13. Leinenbach, D.: Compiler Verification in the Context of Pervasive System Verification. PhD thesis, Saarland University, Saarbrücken (2008)
14. Lochbihler, A.: Verifying a compiler for Java threads. In: ESOP. (2010) 427–447
15. Hoare, T.: The verifying compiler: A grand challenge for computing research. Journal of the ACM **50**(1) (2003) 63–69