PE-KeY: A Partial Evaluator for Java Programs*

Ran Ji and Richard Bubel

Technische Universität Darmstadt, Germany
ran|bubel@cs.tu-darmstadt.de

Abstract. We present a prototypical implementation of a partial evaluator for Java programs based on the verification system KeY. We argue that using a program verifier as technological basis provides potential benefits leading to a higher degree of specialization. We discuss in particular how loop invariants and preconditions can be exploited to specialize programs. In addition, we provide the first results which we achieved with the presented tool.

1 Introduction

In this paper we present a prototypical implementation of a partial evaluator for Java based on the verification system KeY [1] called PE-KeY. The theoretical framework for this approach has been presented in [2].

The KeY verification system formalizes the Java programming language as proof system using a Gentzen-style sequent calculus capturing the sequential Java semantics faithfully. Although declarative, the sequent calculus used in KeY features a strong operational flavor. The calculus rules capturing the semantics of the Java programming language, are designed following closely the symbolic execution paradigm. They realize basically a symbolic interpreter, which differs from a concrete interpreter by using symbolic input values instead of concrete values. For instance, consider the Java program statement `x = y + z;` where the program variables `y` and `z` have the symbolic values y_0 and z_0 respectively. Symbolically executing the statement leads to a (symbolic) state update for program variable `x` whose new symbolic value becomes the expression (and *not* the value of) $y_0 + z_0$. In general, the symbolic execution of control flow statements like an if-statement will branch as the condition might be true or false depending on the *instantiation* of the symbolic values. Subsequently, the symbolic interpreter follows both branches in separation.

Once the Java program has been executed symbolically, the verifier ends up with a set of symbolic states. These symbolic states represent in particular all concrete states the original Java program may encounter in an actual program run. In a verification setting one has now to prove that the property of interest is valid in all these possible final states.

* This work has been supported by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-231620 HATS project *Highly Adaptable and Trustworthy Software using Formal Models*.

The idea of using symbolic execution as a technology for program verification goes back to [3]. While the verification of a program is performed in an analytical manner, i.e., the program is eliminated until only first-order proof obligations remain to be proven, one can also read the proof in a program construction fashion. The tool presented here interweaves both views. While the original source program is analyzed (verified) in a first phase, the specialized program is constructed in a second phase by traversing the obtained proof tree in the opposite direction.

The potential benefit of this approach is the following: The verifier maintains a faithful representation of the symbolic state at all intermediate program states. The degree of precision of the symbolic state is crucial for possible simplifications of program expressions as well as for other specialization techniques like dead-code elimination. We argue that using a program verifier and associated techniques allows the achievement of a high degree of precision. For instance, in case of loops we are able to use unwinding and/or loop invariants. The possibility to provide (strong) loop invariants allows to maintain a highly precise description of the symbolic state even if a loop cannot be completely unwound. A similar argument holds for method calls where method contracts can be used in addition to method inlining.

Another benefit, is that (modular) program verification does not assume a single entry point and can be applied to each method in isolation. In general this limits the effect of program specialization as the parameters cannot be assumed to have static and known values. However, the state and the allowed values of the method parameters are usually restricted by method preconditions and system invariants. In a program verification environment we use these restrictions to simplify expressions and to cut off infeasible program paths. Hence, the specialized programs do not include code for these infeasible code paths. In addition our approach is not restricted to static input values but can also achieve specialization (and/or certain kinds of optimizations) using first-order constraints on the input values.

Further, the program verification calculus can itself be extended by rules performing basic partial evaluation steps like constant propagation. This extension and a logic characterization of the partial evaluation rules has been presented in [4]. Using this technique improves the obtainable degree of specialization considerably.

The paper is structured as follows: In Section 2 we describe the theoretical background of our approach. Section 3 describes the implementation and the results achieved so far. Section 4 describes related work. Finally, Section 5 concludes with an outlook of ongoing and future work.

2 Calculus

2.1 Dynamic Logic

We consider only sequential Java programs (without garbage collection) and can thus make the assumption that all programs are deterministic. In addition,

when using the notion "program" we usually mean an executable sequence of statements. If we want to refer to the context in which these statements are executed, we make this explicit by using the notion *context program*. The *context program* encompasses all interface and class declarations.

Java Dynamic Logic (JavaDL) is basically a standard first-order logic plus two modalities $\langle\cdot\rangle\cdot$ (diamond) and $[\cdot]\cdot$ (box). In this paper we use only the box modality. Given an executable sequence of Java statements `p` and an arbitrary JavaDL formula ϕ then the formula

- $\langle \texttt{p} \rangle \phi$ means intuitively that program `p` terminates and in its final state ϕ holds (total correctness).
- $[\texttt{p}]\phi$ means that if program `p` terminates then ϕ holds in its final state (partial correctness).

As mentioned in the introduction, symbolic states play a central role for our approach. The representation of symbolic states, and in particular of state changes are crucial and often a bottleneck for symbolic interpreters. JavaDL models locations (program variables, attributes, etc.) as flexible constants (or unary functions on objects), i.e., constants and functions whose value can be changed by a program. Relations between locations and restrictions on their values can be expressed using standard first-order (dynamic logic) formulas. To keep track and to represent state changes efficiently, JavaDL has one additional important feature named *updates*.

An elementary update u is an update of the form $loc := val$ where loc is a program variable and val is a term representing the value assigned to loc. Updates can be parallelized $u_1 \| \ldots \| u_n$ meaning that all locations are assigned their new values simultaneously. An update u can be applied to terms t and formulas ϕ resulting again in a term $\{u\}t$ or a formula $\{u\}\phi$. Some examples:

- $\{\texttt{i} := \texttt{j}\}\phi$: formula ϕ is evaluated in a state where the program variable `i` is assigned `j`.
- $\{\texttt{i} := \texttt{j} \| \texttt{j} := \texttt{i}\}\phi$ where ϕ is evaluated in a state where the values of `i` and `j` have been swapped.

Update simplification is performed eagerly to achieve a compact representation of the symbolic state.

2.2 Sequent Calculus

A formula in JavaDL is proven correct in KeY using a Gentzen-style sequent calculus. A sequent is a data structure of the following shape $\Gamma \Longrightarrow \Delta$, where Γ and Δ are sets of formulas. The meaning of a sequent is the same as the meaning of the implication $\bigwedge_{\gamma \in \Gamma} \gamma \rightarrow \bigvee_{\delta \in \Delta} \delta$. The sequent calculus performs syntactic transformations of the sequents by applying rules of the form

$$\mathsf{name}\ \frac{seq_1 | \ldots | seq_n}{seq}$$

where $seq, seq_i, i \in \{1, \ldots n\}, n \geq 0$ are sequents. The sequent seq is called conclusion of the rule, while the sequents $seq_1, \ldots, seq_n$ are called premises. An example of such a rule is the andRight rule:

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

The formula $\phi \wedge \psi$ is called main formula of the sequent, ϕ and ψ are schema variables that stand for any possible JavaDL formula, Γ, Δ are schema variables that match on sets of formulas.

Application of a rule happens in an analytic manner, i.e., the rule is applied from bottom to top. A rule can be applied on a sequent s if there is a match for its conclusion. In that case the premises are instantiated accordingly and become the children of sequent s.

A sequent proof is a tree with a root r where each node $n \neq r$ is the result of a rule application on its parent node. A branch of the tree is closed if a closure rule like close/closeTrue/closeFalse

$$\frac{*}{\Gamma, \phi \Longrightarrow \phi, \Delta} \qquad \frac{*}{\Gamma \Longrightarrow \mathit{true}, \Delta} \qquad \frac{*}{\Gamma, \mathit{false} \Longrightarrow \Delta}$$

has been applied, marking the sequent as valid. A proof is called closed if all its branches are closed.

For this paper we are most interested in the rules working on program formulas. The calculus is designed to work like a symbolic interpreter. The rules match and work on the first (active) statement. One class of rules performs equivalent program transformations which stepwise decompose a complex statement into a series of atomic statements. While another class of rules translates atomic statements into a first-order logic representation.

The rule postInc

$$\frac{\Gamma \Longrightarrow \{u\}[\texttt{int t = i; i = i + 1; j = t; r}]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\texttt{j = i++; r}]\phi, \Delta}$$

is a representative of a pure program transformation rule transforming a complex statement in two simpler statements[1]. The assignment rule

$$\text{assignment}\ \frac{\Gamma \Longrightarrow \{u\}\{\texttt{j} := \texttt{i}\}[\texttt{r}]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\texttt{j = i; r}]\phi, \Delta}$$

belongs to the second class of rules. It performs the single side-effects of an atomic assignment by moving the assignment into an update. Another representative of this class is the rule conditional

$$\frac{\Gamma, \{u\}b \Longrightarrow \{u\}[\texttt{p;r}]\phi, \Delta \quad \Gamma, \{u\}\neg b \Longrightarrow \{u\}[\texttt{q;r}]\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\texttt{if (b) \{ p \} else \{ q \} r}]\phi, \Delta}$$

[1] The temporary variable `t` is necessary because the schema variables `i` and `j` may match on the same local variable.

which causes the proof to split. The first branch assumes that `b` holds and we have to show that ϕ holds if the `then` branch is executed or that the assumption `b` is contradictory in the current state. Analogous for the `else` branch.

2.3 Compilation Rules

While the sequent calculus is applied analytically when used for verification, it is also possible to interpret a proof in a program construction manner. We use the second reading as motivation for our approach to implement a partial evaluator on top of a proof attempt.

The main idea is to extend the box modality

$$[\ \mathtt{p} \sim \mathtt{sp_p}\ ]@(obs, use)$$

to carry additional information. The extension consists of a second compartment for the specialized version $sp_{\mathtt{p}}$ of the source program `p` and two additional annotations *obs* and *use* containing sets of locations (program variables, fields, etc.). The location set *obs* keeps track of observable locations by an "outside" entity, while the location set *use* contains those locations that are read from in the continuation of the program. Without going into detail, these sets of locations are used to detect unused variable assignments and to eliminate them as soon as possible. We call these modalities, *bisimulation* modalities as `p` and $\mathtt{sp_p}$ have to be in a bisimulation relation with respect to the postcondition and the set of observable variables.

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

$$\text{ruleName}\quad \frac{\begin{array}{c}\Gamma_1 \Longrightarrow \{u_1\}[\ \mathtt{p_1} \sim \mathtt{sp_1}\ ]@(obs_1, use_1)\phi_1, \Delta_1 \\ \cdots \\ \Gamma_n \Longrightarrow \{u_n\}[\ \mathtt{p_n} \sim \mathtt{sp_n}\ ]@(obs_n, use_n)\phi_n, \Delta_n\end{array}}{\Gamma \Longrightarrow \{u\}[\ \mathtt{p} \sim \mathtt{sp}\ ]@(obs, use)\phi, \Delta}$$

The application of sequent calculus rules for the bisimulation modality consists of two phases.

1. Symbolic execution of source program `p`. It is performed bottom-up as usual in sequent calculus rules. In addition, the observable location sets obs_i are also propagated since they contain the locations observable by $\mathtt{p_i}$ and ϕ_i that will be used in the second phase to synthesize the specialized program. Normally *obs* could contain the return variables of a method and the locations used in the continuation of the program.
2. We synthesize the target program $\mathtt{sp_i}$ and use_i by applying the rules in a top-down manner.

To obtain a specialization of a method `m(args)` we start the proof with a sequent of the form

$$\Longrightarrow pre \rightarrow [\ \mathtt{r{=}m(args)} \sim \mathtt{sp}\ ]@(\{\mathtt{r}\}, use)\mathtt{POST}$$

where *pre* is the precondition of method `m` and `POST` is an unspecified predicate which can neither be proven nor disproved. This allows the easy identification of closed proof branches with infeasible paths and thus sound elimination of dead-code. The variables *sp* and *use* are placeholders for the specialized program and the used variable set which is computed in the second phase.

In the second phase, when the program is fully symbolically executed, the specialized program is synthesized by "applying" the rules in the opposite direction. This phase starts effectively with nodes where the emptyBox rule has been applied:

$$\text{emptyBox}\ \frac{\Gamma \Longrightarrow \{\sqcap\}@(obs, use)\phi, \Delta}{\Gamma \Longrightarrow \{u\}[\ \texttt{nop}\ \sim\ \texttt{nop}\ ]@(obs, obs)\phi, \Delta}$$

with `nop` denoting the empty program. The rule initializes the variable set *use* to the set of observable variables *obs*. The idea is that *use* keeps track of all variables whose value has (potentially) been read.

We show only some of the bisimulation rules introduced in [2]. We use $\overline{p}$ to denote the specialized version of p and omit for space reasons the context variables Γ, Δ. The rule

$$\text{assignLocalVariable}\ \frac{\Longrightarrow \{u\}\{\texttt{l} := \texttt{r}\}[\ \omega\ \sim\ \overline{\omega}\ ]@(obs, use)\phi}{\begin{array}{l}\Longrightarrow \{u\}[\ \texttt{l} = \texttt{r}; \omega\ \sim\ \texttt{l} = \texttt{r}; \overline{\omega}\ ]@(obs, use - \{l\} \cup \{r\})\phi \\ \qquad\qquad \text{if } l \in use\ \&\ \texttt{r} := \texttt{r}' \notin u \\ \Longrightarrow \{u\}[\ \texttt{l} = \texttt{r}; \omega\ \sim\ \texttt{l} = \texttt{r}'; \overline{\omega}\ ]@(obs, use - \{l\} \cup \{r'\})\phi \\ \qquad\qquad \text{if } l \in use\ \&\ \texttt{r} := \texttt{r}' \in u \\ \Longrightarrow \{u\}[\ \texttt{l} = \texttt{r}; \omega\ \sim\ \overline{\omega}\ ]@(obs, use)\phi \qquad\qquad otherwise\end{array}}$$

describes the specialization of an assignment statement where one local variable is assigned to another one. The rule has three conclusions of which only one is taken. They differ only in the synthesized program compartments, i.e., in the analyzing (first) phase no ambiguity arises.

If the left side l of the assignment is a location with a (potential) read access before its next re-definition, i.e., $l \in use$, an assignment statement is generated. The *use* set is updated by removing the now re-defined program variable l and adding the program variable r which is read by the assignment. This explains the first of the three conclusions. But we can do better: in case the preceding update contains an elementary update with r as left-hand side and r' as right-hand side, we inline the actual value directly and generate as assignment $l = r'$. The *use* set is updated accordingly[2].

We give a brief example motivating the existence of the second conclusion. Assume we encounter the following sequent:

$$\Longrightarrow \{\ldots \| \texttt{y} := \texttt{z} + \texttt{1}\}\{\texttt{x} := \texttt{y}\}[\ \omega\ \sim\ \overline{\omega}\ ]@(obs, \{x, \ldots\})\phi$$

[2] if r' is an expression all variables occurring in r' have to be added to *use*

Since x is in the *use* set, an assignment statement has to be generated. Notice that $\mathtt{y} := \mathtt{z} + \mathtt{1}$ occurs in the update u, therefore the assignment is synthesized as `x = z + 1` according to the second case of the assignment rule. The variable `x` is removed and the variable `z` is added to the *use* set. We get as result:

$$\Longrightarrow \{\ldots \| \mathtt{y} := \mathtt{z} + \mathtt{1}\}[\mathtt{x} = \mathtt{y}; \omega \sim \mathtt{x} = \mathtt{z} + \mathtt{1}; \overline{\omega}\,]@(obs, \{z, \ldots\})\phi$$

The third conclusion of the assignment rule is used if the value of l has not been accessed by the remaining program ω and does not generate an assignment at all.

Synthesizing loops is achieved using one (or a combination) of two approaches: (i) loop unwinding to execute a fixed number of loop iterations and (ii) using the loop invariant rule for loops with no fixed bound:

whileInv

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \{u\}inv, \Delta \\ \Gamma, \{u\}\{\mathcal{V}_{mod}\}(\mathtt{b} = \mathtt{TRUE} \wedge inv) \Longrightarrow \{u\}\{\mathcal{V}_{mod}\} \\ \qquad\qquad [\,\mathtt{p} \sim \overline{\mathtt{p}}\,]@(obs \cup use_1 \cup \{b\}, use_2)inv, \Delta \\ \Gamma, \{u\}\{\mathcal{V}_{mod}\}(\mathtt{b} = \mathtt{FALSE} \wedge inv) \Longrightarrow \{u\}\{\mathcal{V}_{mod}\}[\,\omega \sim \overline{\omega}\,]@(obs, use_1)\phi, \Delta\end{array}}{\Gamma \Longrightarrow \{u\}[\,\mathtt{while(b)\{p\}}\,\omega \sim \mathtt{while(b)\{\overline{p}\}}\,\overline{\omega}\,]@(obs, use_1 \cup use_2 \cup \{b\})\phi, \Delta}$$

to achieve program specialization in finite time.

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

Two things are of importance: the third premise 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, we must add to the variable set *obs* of the second premise the used variables of the third 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 using the resulting use_1 set in the second premise.

3 Implementation and Experiments

The implementation of PE-KeY is a non-trivial extension based on KeY, which includes the following efforts:

– An information collector along with the symbolic execution of the source Java program. It keeps track of the observable variables and constructs the working stack that is used in the synthesize phase.
– An integrated partial evaluator which performs some simple partial evaluation operations such as constant propagation and dead code elimination. It is used in the symbolic execution phase.
– The compilation rules that are used to generate specialized program in the second phase. KeY's sequent calculus has around 1200 rules of which around 100-150 rules are used for symbolic execution of programs. Around half of them has been implemented in the current version of PE-KeY, but a considerable effort is required to get a complete coverage.
– An update analyzer used to extract symbolic values of program variables from preceding updates to achieve a higher degree of specialization.

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

PE-KeY is available at `www.key-project.org/ifm12` and runnable via Java Web Start (no installation needed). We have tried PE-KeY with a set of example programs that are available from the website given above. Although in an early stage, the examples indicate the potential of PE-KeY once full Java is supported. For instance, the (simplified) formula

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

leads to the following specialization of the conditional statement:

```
max = i;
```

because of the precondition `i>j` and thanks to the integrated first-order reasoning mechanism in PE-KeY. For the same reason,

$$\texttt{i = 5} \rightarrow [\texttt{i++;}]\texttt{POST}$$

results in the specialized statement `i = 6`.

In fact, the program can be specialized according to the given specification from a general implementation. Fig. 1 shows a fragment of a bank account implementation. A bank account includes the current available balance and the credit line (normally fixed) that can be used when the balance is negative. Cash withdraw can be done by calling the `withdraw` method. If the withdraw amount does not exceed the available balance, the customer will get the cash without any extra service fee; if the available balance is less than the amount to be withdrawn, the customer will use the credit line to cover the difference with 5 extra cost; if the withdrawn amount could not be covered by both the available balance and the credit line, the withdraw does not succeed. In every case, the information of the new available balance will be printed (returned). This is a general

implementation of the cash withdrawal process, but some banks (or ATMs) only allow cash withdraw when the balance is above 0. In this case, the precondition of the `withdraw` method is restricted to `withdrawAmt <= availableBal`. Then with help of PE-KeY, the implementation of method `withdraw` is specialized to:

```
  return availableBal - withdrawAmt;
```

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

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

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

Fig. 1. Code fragment of bank account

We applied our prototype partial evaluator also on some examples stemming from the JSpec test suite [5]. One of them is concerned with the computation of the power of an arithmetic expression, as shown in Fig. 2.

The interesting part is that the arithmetic expression is represented as an abstract syntax tree (AST) like 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

```
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;
 }
}
```

Fig. 2. Source code of the Power example as found in the JSpec suite [5]

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

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

As input formula for PE-KeY we get:

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

PE-KeY then executes the program symbolically and extracts the specialized program $\texttt{sp}_{\texttt{res}}$ as `power = ((x*x)*x)*...)*x;` (see Fig. 3). The achieved result is a simple `int`-typed expression without the intermediate creation of the abstract syntax tree and should provide a significant better performance than executing the original program.

Our current implementation is still in its infancies and there are other examples of the JSpec test suite. But the already achieved results indicate that the presented approach can be scaled up to real world examples. More examples and results can be found at `www.key-project.org/ifm12`.

4 Related Work

JSpec [5] is the state-of-the-art program specializer for Java. In fact, JSpec does not support full Java but a subset without concurrency, exception, reflection.

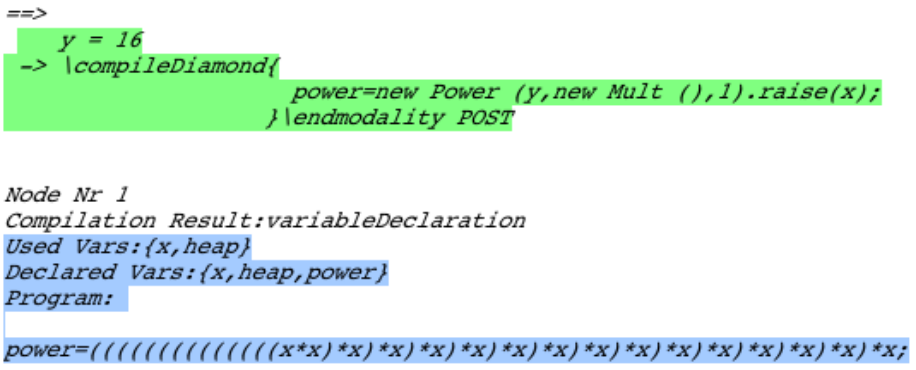

Fig. 3. Specialized program computed by PE-KeY as result of the JSpec example

Our tool has similar limitations due to the restriction of KeY itself. JSpec uses an *offline* partial evaluation technique that depends on *binding time analysis*, which in general is possibly not as precise as *online* partial evaluation.

Civet [6] is a recent partial evaluator for Java based on *hybrid* partial evaluation, which performs offline-style specialization using an online approach without static binding time analysis. The programmer needs to explicitly identify which parts of the programs partial evaluation should be applied. In our approach, which is based on the program verification, the specification (or precondition) is mainly used to prove the correctness of the programs, and it also contributes to the program specialization. Our approach is similar to Civet in the sense that the specification is also user provided, but no extra annotations specially for the partial evaluation purpose is needed.

Our approach is based on symbolic execution to derive information on-the-fly, similar to *online* partial evaluation [7], the main difference being that we do not generate the specialized program during the symbolic execution phase, but synthesize it in the second phase. In principle, our first phase can obtain as much information as online partial evaluation, and the second phase can generate a more precise specialized program.

A big advantage of our approach is that the specialized program is guaranteed to be correct with respect to the original program. It is related to the work of proving the correctness of a partial evaluator by Hatcliff et al. [8,9]. The difference is that they need to encode the correctness properties into a logic programming language to perform the proof; while our approach ensures the correctness naturally by the deductive compilation rules and thus no further proof is needed.

Verifying Compiler [10] project aims at the development of a compiler that verifies the program during compilation. In contrast to this, our approach might be called the *Compiling Verifier*, since the specialized Java program is generated

after verifying the source program. The correctness of the generated program is ensured by the compilation rules. Related to our work, compiler verification [11] aims to guarantee the correctness of the target program. However, compiler verification attempts to verify the compiling program which is very expensive and hardly scales to realistic target languages and sophisticated optimizations.

Our work is closely related to rule-based compilation [12,13], but to the best of our knowledge their inference machine is by far not as powerful as the reasoning engine of KeY. Also closely related are recent approaches to translation validation of optimizing compilers (e.g., [14]) which also use a theorem prover to discharge proof obligations. They work usually on an abstraction of the target program. Both mentioned approaches encode the compilation strategy within the rules, while our approach separates the actual strategy from the translation rules. What distinguishes our work from most approaches that we know is that the starting point is a system for functional verification of Java which is used for program specialization in such a way that it becomes fully automatic.

5 Conclusions and Future Work

In this work, we presented PE-KeY, a partial evaluator for Java programs based on KeY. It works in two phases. In the first phase, symbolic execution interleaved with simple partial evaluation is performed, which is similar to online partial evaluation process. In the second phase, the specialized Java program is synthesized. A use-definition chain set is maintained to eliminate unused assignments and to avoid unnecessary statements occurring in the specialized program. The advantage of PE-KeY, among other Java partial evaluators, is that the correctness of the specialization is naturally guaranteed by the bisimulation relationship of the source and specialized programs, together with the soundness of the program logic. This has been proved in our previous work[2].

As an extension of KeY, PE-KeY does not support multi-threading and floating point in Java due to the restriction of KeY itself. However, these features are being investigated in KeY, once they are supported, PE-KeY could easily extend these features as well.

There are still some optimization opportunities for the current version of PE-KeY. For instance, on encountering a loop, the heuristics that decide whether to unwind it or not have a strong influence on the resulting specialized programs. Importing information, e.g., loop invariants or ranking functions, from other tools could also be useful. And for the method call, the method body is always inlined for the moment, however, to use the method contract instead might sometimes generate a better specialized program and is worthy to investigate in the future.

The idea of this paper is to generate specialized Java programs, however, the bisimulation modality is not restricted to source and target program being from the same language, but it can be generalized to other languages provided with corresponding observable locations. Consequentially, the approach is still sound for generating bytecode or other intermediate languages. An important future

work will be to generate Java bytecode from Java source by using our approach. It will be a deductive Java compiler which guarantees a correct compilation without further verification of the bytecode or JVM as introduced in [15]. We also plan to apply our approach to the modeling language ABS developed in the context of the HATS project [16,17].

References

1. Beckert, B., Hähnle, R., Schmitt, P., eds.: Verification of Object-Oriented Software: The KeY Approach. Volume 4334 of LNCS. Springer (2007)
2. Bubel, R., Hähnle, R., Ji, R.: Program specialization via a software verification tool. In Aichernig, B., de Boer, F.S., Bonsangue, M.M., eds.: Post Proc. of FMCO'10. LNCS, Springer (2011)
3. King, J.C.: A program verifier. PhD thesis, CMU (1969)
4. Bubel, R., Hähnle, R., Ji, R.: Interleaving symbolic execution and partial evaluation. In: Post Proc. of FMCO'09. LNCS, Springer (2010)
5. Schultz, U.P., Lawall, J.L., Consel, C.: Automatic program specialization for Java. ACM-TPLS **25**(4) (2003) 452–499
6. Shali, A., Cook, W.R.: Hybrid partial evaluation. In: OOPSLA. (2011) 375–390
7. Ruf, E.S.: Topics in online partial evaluation. PhD thesis, Stanford University, Stanford, CA, USA (1993) UMI Order No. GAX93-26550.
8. Hatcliff, J.: Mechanically verifying the correctness of an offline partial evaluator. In: PLILP. (1995) 279–298
9. Hatcliff, J., Danvy, O.: A computational formalization for partial evaluation. Mathematical Structures in Computer Science **7**(5) (1997) 507–541
10. Hoare, T.: The verifying compiler: A grand challenge for computing research. J. ACM **50** (January 2003) 63–69
11. Dave, M.A.: Compiler verification: a bibliography. SIGSOFT Softw. Eng. Notes **28** (November 2003) 2–2
12. Augustsson, L.: A compiler for lazy ML. In: Proc. of the ACM Symposium LFP'84, New York, USA, ACM (1984) 218–227
13. Breebaart, L.: Rule-based compilation of data parallel programs. PhD thesis, Delft University of Technology (2003)
14. Barrett, C.W., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., Zuck, L.D.: TVOC: A translation validator for optimizing compilers. In Etessami, K., Rajamani, S.K., eds.: CAV 2005. LNCS, Spinger (2005) 291–295
15. Stärk, R.F., Schmid, J., Börger, E.: Java and the Java Virtual Machine. Springer-Verlag (2001)
16. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E.B., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.Y.H.: Modeling Spatial and Temporal Variability with the HATS ABS Language. In Bernardo, M., Issarny, V., eds.: Formal Methods for Eternal Networked Software Systems. Volume 6659 of LNCS. Springer (2011) 417–457
17. Clarke, D., Muschevici, R., Proença, J., Schaefer, I., Schlatte, R.: Variability modelling in the ABS language. In Aichernig, B., de Boer, F.S., Bonsangue, M.M., eds.: Post Proc. of FMCO'10. LNCS, Springer (2011)