Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 190 (2007) 35–50

www.elsevier.com/locate/entcs

Formal Translation of Bytecode into BoogiePL

Hermann Lehner[1] and Peter Müller[2]

ETH Zurich, Switzerland

Abstract

Many modern program verifiers translate the program to be verified and its specification into a simple intermediate representation and then compute verification conditions on this representation. Using an intermediate language improves the interoperability of tools and facilitates the computation of small verification conditions. Even though the translation into an intermediate representation is critical for the soundness of a verifier, this step has not been formally verified. In this paper, we formalize the translation of a small subset of Java bytecode into an imperative intermediate language similar to BoogiePL. We prove soundness of the translation by showing that each bytecode method whose BoogiePL translation can be verified, can also be verified in a logic that operates directly on bytecode.

Keywords: Program verification, verification conditions, intermediate language, Java bytecode, BoogiePL

1 Introduction

Many modern program verifiers such as ESC/Java [10], Boogie [2], Krakatoa [12], and Caduceus [9] verify programs in two steps. First, they translate the program and the specification into an intermediate representation such as guarded commands, BoogiePL [7], or the Why language [8]. In the second step, they compute verification conditions for the intermediate representation and pass them to a theorem prover. Using an intermediate language improves the interoperability of tools. For instance, Krakatoa and Caduceus translate Java and C code to the Why language, which allows them to share the Why back end. Moreover, simple intermediate representations facilitate the generation of small verification conditions through passification [11].

The translation into an intermediate representation is critical for the soundness of a program verifier. It has to ensure that the verification conditions for the intermediate program are strong enough to guarantee the correctness of the original

[1] hermann.lehner@inf.ethz.ch

[2] peter.mueller@inf.ethz.ch

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2007.02.059

program; otherwise the intermediate program could be verified although the original program is incorrect. Despite its importance for soundness, the translation into an intermediate representation has not been formally verified.

In this paper, we formalize the translation of Java bytecode into an untyped version of the intermediate language BoogiePL and prove soundness of the translation by showing that the verification conditions for the intermediate program are at least as strong as the corresponding verification conditions for the original program. Our formalization and proof [6,14] cover a large subset of Java bytecode. Due to space limitations, we focus on a small, but interesting subset in this paper.

Outline. This paper is organized as follows. Sec. 2 describes the bytecode subset and a weakest precondition calculus that directly operates on bytecode. Sec. 3 introduces BoogiePL including a weakest precondition calculus. The translation from bytecode to BoogiePL is defined in Sec. 4. We illustrate the translation by an example in Sec. 5 and prove soundness in Sec. 6. We review related work in Sec. 7 and offer conclusions in Sec. 8.

2 Java Bytecode

In this section, we describe the bytecode subset used in the rest of the paper. We also explain how specifications are formalized and outline a weakest precondition (wp) calculus for the bytecode language.

2.1 Language Subset

We consider a subset of sequential Java bytecode that contains classes, interfaces, fields, dynamically-bound methods, and exceptions. For brevity, we consider only a very small set of instructions. Nevertheless, this subset is representative as it contains instructions of the major groups: '`iload_`*n*' for manipulations of registers and the operand stack, '`ifgt` *Label*' for control flow instructions, '`invokevirtual` *Method*' and '`ireturn`' for method call and return, as well as '`new` *TypID*' and '`getfield` *FieldID*' for heap operations. The translation of most of the remaining instructions is analogous to these representatives. *Label*, *TypID*, *Method*, and *FieldID* are the sorts for labels, class and interface names, method names, and field names of a program. For simplicity, we assume that type names, method names, and field names are unique in a program. We expect the control flow graph of the bytecode program to be reducible, which means that there is only one single entry point for each loop in the control flow graph. Java compilers always produce reducible control flow graphs. For hand-written bytecode, this property can be achieved by code duplication.

We model the exception table by a function *handlers* that returns the set of labels of all possible handlers for an exception of a given type and all of its subtypes. Including subtypes is necessary because the exception handler that is executed is determined dynamically based on the runtime-type of the exception object, which may be a subtype of the statically-known type.

The special label '$\perp$' is used to indicate that an exception might not be handled by the current method:

$$handlers : Method \times Label \times TypID \rightarrow set\ of\ Label$$

2.2 State Model

The state of an execution of a bytecode program consists of the heap, the operand stack, and the values in the registers. The sort *Value* models object references and values of primitive types such as integers. The instance variables of an object are modelled by sort *InstVar*. The function $iv : Value \times FieldID \rightarrow InstVar$ yields the instance variable for a given object and field identifier.

The heap is modeled as a data type with main sort *Heap* and the following operations:

$$\begin{array}{llll}
update : & Heap \times InstVar \times Value & \rightarrow Heap & \\
get : & Heap \times InstVar & \rightarrow Value & alloc : Value \times Heap \ \rightarrow bool \\
add : & Heap \times TypID & \rightarrow Heap & new : \ Heap \times TypID \rightarrow Value
\end{array}$$

$update(h, i, v)$ updates instance variable i in heap h with value v. $get(h, i)$ yields the value of instance variable i in heap h. Object creation is encoded by two functions: $new(h, C)$ yields a new object of class C in heap h and $add(h, C)$ yields the extended heap. The axiomatization of the heap model relates these two functions and ensures that the two functions are used consistently. Finally, $alloc(v, h)$ yields whether value v is allocated in heap h. We omit the axiomatization of these functions because our translation and soundness proof do not rely on it. The details are presented in a paper by Poetzsch-Heffter and Müller [13].

When modeling the state of the JVM, we distinguish the state before, during, and after the execution of a method. A *PreState* contains a heap and values for the method parameters, a *LocalState* contains a heap, an operand stack, and values for the method parameters and local variables, and a *PostState* contains a heap and a value, which is the result value of the method if it terminates normally or an exception if it terminates abruptly.

$$\begin{array}{ll}
PreState : & (Heap \times Arguments) \\
LocalState : & (Heap \times OperandStack \times Locals) \\
PostState : & (Heap \times Value)
\end{array}$$

where *Arguments*, *Locals*, and *OperandStack* are lists of *Value*.

2.3 Specifications

We formalize the specification of a bytecode program as a specification table, which is accessed by the following functions:

$$\begin{array}{lll} \mathit{pre}: & \mathit{Method} & \rightarrow (\mathit{PreState} \rightarrow \mathit{bool}) \\ \mathit{post}: & \mathit{Method} & \rightarrow (\mathit{PreState} \times \mathit{PostState} \rightarrow \mathit{bool}) \\ \mathit{post}_{\mathsf{X}}: & \mathit{Method} & \rightarrow (\mathit{PreState} \times \mathit{PostState} \rightarrow \mathit{bool}) \\ \mathit{local}: & \mathit{Method} \times \mathit{Label} & \rightarrow (\mathit{PreState} \times \mathit{LocalState} \rightarrow \mathit{bool}) \end{array}$$

$\mathit{pre}(m)$ yields the precondition of a method m, which is a predicate over a *PreState*. $\mathit{post}(m)$ and $\mathit{post}_{\mathsf{X}}(m)$ yield the normal and exceptional postcondition of m, respectively. Both are predicates over a *PreState* (to refer to the initial values of the heap and method arguments) and a *PostState*. Local annotations at a label l are denoted by $\mathit{local}(m, l)$. They are predicates over a *PreState* and a *LocalState*. *Local annotations* are used to encode loop invariants. In order to avoid fixpoint computations in the wp calculus, we require that every entry point to a loop in the control flow graph has a local annotation in the specification table. $\mathit{local}(m, l)$ is undefined if l is not the beginning of a loop.

2.4 Direct Verification Condition Generation for Bytecode

Our weakest precondition calculus for bytecode is a simplified version of the calculus by Grégoire, which is proven sound w.r.t. an operational semantics [6]. In this subsection, we present those parts of the calculus that are needed in the rest of the paper. We assume that each bytecode program passes the bytecode verifier before the wp-calculus is applied. Therefore, we do not prove type correctness and the absence of stack over- and underflows in the calculus.

We define a function $\mathit{wp}_{\mathsf{vc}} : \mathit{Method} \times \mathit{Label} \rightarrow (\mathit{PreState} \times \mathit{LocalState} \rightarrow \mathit{bool})$. $\mathit{wp}_{\mathsf{vc}}(m, l)$ yields the *weakest precondition* for the instruction at label l of method m. If this weakest precondition holds, an execution starting from l will: (1) terminate normally in a state that satisfies m's normal postcondition, (2) terminate abruptly in a state that satisfies m's exceptional postcondition, (3) abort due to a runtime error, or (4) run forever.

To handle local annotations, we use a second wp function, which yields the local precondition. The *local precondition* is the local annotation from the specification table if there is any, and otherwise the weakest precondition:

$$\mathit{wp}_{\mathsf{l}}(m, l) = \begin{cases} \mathit{local}(m, l) & : \text{ if } \mathit{local}(m, l) \text{ is defined} \\ \mathit{wp}_{\mathsf{vc}}(m, l) & : \text{ otherwise} \end{cases}$$

The weakest precondition function $\mathit{wp}_{\mathsf{vc}}$ is defined in Fig. 1. For `iload_n`, $\mathit{wp}_{\mathsf{vc}}$ applies the local precondition of the successor instruction to an adapted local state. For `ifgt`, $\mathit{wp}_{\mathsf{vc}}$ yields the conjunction of the local preconditions of the possible jump target and the successor label, weakened by the appropriate conditions. The weakest precondition of `ireturn` is the normal method postcondition. `new` is analogous to `iload_n`. For `getfield`, the weakest precondition requires the top stack element to be non-null in order to prevent `NullPointerExceptions`. For method invocations, we have to prove that the target is non-null and that the precondition is satisfied.

Instruction at l	$wp_{\mathsf{vc}}(m,l)(\sigma_0,(h, top :: os, reg)) \;=$
`iload_`n	$wp_{\mathsf{l}}(m, l+1)(\sigma_0,(h, reg[n] :: top :: os, reg))$
`ifgt` l'	$(top > 0 \Rightarrow wp_{\mathsf{l}}(m,l')(\sigma_0,(h, os, reg))) \wedge$ $(top \le 0 \Rightarrow wp_{\mathsf{l}}(m, l+1)(\sigma_0,(h, os, reg)))$
`ireturn`	$post(m)(\sigma_0,(h, top))$
`new` t	$wp_{\mathsf{l}}(m, l+1)(\sigma_0,(add(h,t), new(h,t) :: top :: os, reg))$
`getfield` f	$top \ne null \wedge wp_{\mathsf{l}}(m, l+1)(\sigma_0,(h, get(h, iv(top, f)) :: os, reg))$
`invokevirtual` t	$top \ne null \wedge pre(t)(h, top) \wedge$ $\forall\, h', rv : ($ $\quad post(t)((h, par),(h', rv)) \;\Rightarrow$ $\quad wp_{\mathsf{l}}(m, l+1)(\sigma_0,(h', rv :: os, reg)$ $) \wedge ($ $\quad post_{\mathsf{X}}(t)((h, par),(h', rv)) \;\Rightarrow$ $\quad \bigwedge_{l_i \in handlers(m,l,typeof(rv))} wp_{\mathsf{l}}(m,l_i)(\sigma_0,(h', rv, reg))$ $)$ where par is a list of stack elements that represent the parameters for t

Fig. 1. Weakest precondition calculus for bytecode. The label $l+1$ denotes the textual successor of label l. The function *typeof* yields the runtime type of a value.

The treatment of postconditions has to take into account both normal and abrupt termination. That is, the normal method-postcondition has to imply the local precondition of the successor instruction and the exceptional method-postcondition has to imply the local preconditions of all possible exception handlers. Note that the value of *handlers* includes $\perp$ if the exception might be propagated. Therefore, we define $wp_{\mathsf{vc}}(m,\perp) = post_{\mathsf{X}}(m)$.

Note that this weakest precondition calculus enforces that instructions do not throw runtime exceptions. For instance, `getfield` requires the receiver to be non-null. This requirement is a source of incompleteness: A method that dereferences null may catch the `NullPointerException` and still satisfy its specification, but cannot be verified in our wp-calculus. However, preventing runtime exceptions simplifies verification by avoiding case splits for each instruction that potentially throws a runtime exception.

To verify a method m, one has to prove that (1) the method precondition implies the local precondition of the first instruction and (2) for each label that has a local annotation, the local precondition $wp_{\mathsf{l}}(m,l)$ implies the weakest precondition $wp_{\mathsf{vc}}(m,l)$. The latter obligation is required to show that loop invariants are actually maintained.

3 BoogiePL

In this section, we give a brief overview of BoogiePL and present a wp-calculus. For details, we refer to a report by DeLine and Leino [7]. To focus on the essentials of the bytecode translation, we use an untyped version of BoogiePL in this paper. However, our full formalization [6] works with the typed language.

3.1 Overview and State Model

BoogiePL programs consist of a prelude and a list of procedures. The prelude specifies a background theory in first-order logic using global variables, constants, axioms, and uninterpreted functions. The procedures contain a specification and an implementation. In our translation, we do not use the procedure specifications. The implementation of a procedure starts with the declaration of all local variables, followed by one or more blocks. A block has a unique ID, a body consisting of BoogiePL commands, and ends with a non-deterministic **goto**, which specifies all possible successor blocks in the control flow graph. A **goto** with an empty list of block IDs terminates the execution. BoogiePL provides the following commands: assignment, **call**, **assume**, **assert**, and **havoc**. The **havoc** command assigns an arbitrary value to a given variable. In the following, we will reuse sort *Method* for procedure names.

The state of a BoogiePL program consists of the values of all global and local variables. The sort $Value_{\mathsf{bpl}}$ contains all possible values of a BoogiePL program: $State_{\mathsf{bpl}} : Var \mapsto Value_{\mathsf{bpl}}$

3.2 Verification Condition Generation for BoogiePL

Our wp-calculus for BoogiePL is similar to the one by Barnett and Leino [3]. However, it requires that the control flow graph of the BoogiePL program is acyclic, whereas the Boogie tool accepts a cyclic control flow graph and transforms it internally into an acyclic graph. We make this transformation explicit in our translation, see Sec. 4.3.

We assume that the commands of a BoogiePL method are numbered. We use the term *position* rather than *label* to refer to the number of a BoogiePL command in order to avoid confusion with the labels of a bytecode instruction.

The weakest precondition function $wp_{\mathsf{bpl}} : Method \times Position \rightarrow (State_{\mathsf{bpl}} \rightarrow bool)$ for BoogiePL is analogous to the wp-calculus for bytecode. If the weakest precondition of a position pos in a procedure m holds, an execution starting from pos will not abort due to an assertion violation. That is, the program will terminate or run forever. The weakest precondition function wp_{bpl} is defined in Fig. 2.

4 Translation from Bytecode to BoogiePL

In this section we describe the translation of our bytecode subset to BoogiePL.

Command at *pos*	$wp_{\mathsf{bpl}}(m, pos)$ =
assume P	$P \Rightarrow wp_{\mathsf{bpl}}(m, pos + 1)$
assert P	$P \wedge wp_{\mathsf{bpl}}(m, pos + 1)$
$x := e$	$wp_{\mathsf{bpl}}(m, pos + 1)[e/x]$
havoc x	$\forall\ x : wp_{\mathsf{bpl}}(m, pos + 1)$
goto $pos_1, \ldots, pos_n$	$\bigwedge_{i:=1..n} wp_{\mathsf{bpl}}(m, pos_i)$

Fig. 2. Weakest precondition calculus for BoogiePL. Substitution of a term e for a variable x is denoted by $[e/x]$.

4.1 Information about the Bytecode Program

Our translation uses information that is computed by the bytecode verifier, namely (1) the height of the operand stack at each label and (2) the control flow graph of each method. We encode this information by the following functions.

$$\begin{array}{lll}
sh : & Method \times Label & \rightarrow int \\
isEdge : & Method \times Label \times Label & \rightarrow bool \\
isEdgeTarget : & Method \times Label & \rightarrow bool \\
isBackEdge : & Method \times Label \times Label & \rightarrow bool \\
isBackEdgeTarget : & Method \times Label & \rightarrow bool
\end{array}$$

$sh(m, l)$ yields the index of the top element of the operand stack before execution of the instruction at label l of method m. $isEdge(m, l_1, l_2)$ yields true iff there is an edge from label l_1 to label l_2 in the control flow graph of method m. If this is the case, $isEdgeTarget(m, l_2)$ yields true. The functions *isBackEdge* and *isBackEdgeTarget* are analogous to *isEdge* and *isEdgeTarget*, but consider only backward edges in the control flow graph.

4.2 Encoding of the Bytecode State in BoogiePL

We encode the state of a bytecode program by a number of BoogiePL variables.

Prestate: The heap model described in Sec. 2 is encoded in the prelude of the BoogiePL program. We do not show this formalization here because it is not interesting. We use the variable `old_heap` to refer to the heap of the prestate. The n parameters of the bytecode method are modeled by the parameters of the BoogiePL procedure, $\texttt{param}_i$ $(i = 0, \ldots, n - 1)$.

Local State: We use the global variable `heap` to model to heap of the current execution state. The operand stack is modeled by the variables $\texttt{stack}_i$, where i denotes the height of the stack (starting with 0). The registers (representing the local variables and parameters) are modeled by variables $\texttt{reg}_i$. Since the maximum height of the operand stack and the number of used registers is given in the class file, we know statically how many of the $\texttt{stack}_i$ and $\texttt{reg}_i$ variables we have to declare in the BoogiePL procedure.

We use the following abbreviations: *params* for list of variables $\texttt{param}_i$, *stacks* for all stack elements $\texttt{stack}_i$, and *regs* for all registers $\texttt{reg}_i$.

The state of a bytecode program and its translation are formally related by the mapping function $map : Method \times State_{\mathsf{bpl}} \times Int \rightarrow (PreState \times LocalState)$:

$$map(m, \rho, h) =$$
$$\Big(\big(\rho(\texttt{old\_heap}), \rho(params)\big), \big(\rho(\texttt{heap}), \rho(\texttt{stack}_h), \ldots, \rho(\texttt{stack}_0), \rho(regs)\big)\Big)$$

4.3 Back-Edge Elimination

As explained in Sec. 2.4, our translation eliminates backward edges in the control flow graph in order to avoid a fixpoint calculation in the wp calculus.

It is important to understand that back-edges do not only occur at jump instructions. Any transition from an instruction to its textual successor instruction could be a back-edge. For instance, the transition from label 8 to label 11 in the bytecode program in Fig. 3 is a back-edge, because label 11 has been visited before due to processing the jump at label 2.

A transition along a back-edge always closes a loop in the control flow graph. Therefore, we can assume that the target of the back-edge has a local annotation for the loop invariant. In the translation, we eliminate the back-edge and instead generate an assertion that the loop invariant holds. A forward-edge is simply translated into a **goto**. This is done by the translation function *TrEdge*. Which is defined for a list of possible successor labels ls.

$TrEdge[\![m : Method, l : Label, ls : \text{list of } Label]\!] =$
  #foreach l' in ls
    #if $isBackEdgeTarget(m, l')$
      assert $TrSpec[\![local(m, l'), (\texttt{old\_heap}, params), (\texttt{heap}, stacks, regs)]\!]$
    #end if
  #end foreach
  #let $l_1, \ldots, l_n$ = all l' in ls where $\neg isBackEdge(m, l, l')$
  goto $\texttt{block\_}l_1$, ..., $\texttt{block\_}l_n$;

In our notation, text in `typewriter` font is directly printed, whereas text in italics is interpreted by the translator. Lines beginning with the '#' character describe how the code is generated. *TrSpec* translates a bytecode specification to the corresponding BoogiePL expression, thereby replacing occurrences of the heap, operand stack, and registers by the given variables. We use the convention that the BoogiePL block for a bytecode basic block starting at label l has the ID $\texttt{block\_}l$.

4.4 Translation of Bytecode Instructions

The function *TrInstr* translates a single bytecode instruction. $instructionAt(m, l)$ denotes the instruction at label l in method m.

$TrInstr[\![m : Method, l : Label]\!] =$
  #let $cntr = sh(m, l)$

#switch $instructionAt(m, l)$

We present the individual cases of the switch in the following:
iload: The loading of a value from a register is translated in an assignment to the stack variable for the top stack element.

#case ***iload*** n

$\texttt{stack}_{cntr+1}$:= $\texttt{reg}_n$;

ifgt: A branch instruction is translated into a non-deterministic `goto` to two successor blocks that assume the branch condition to be *true* or *false*, respectively. The true-block then jumps to the target label l'. To eliminate back-edges, we apply $TrEdge$ at this point. The false-block is continued with the translation of the next instruction.

#case ***ifgt*** l'

goto block_l_True, block_l_False;

<u>block_l_True:</u>

assume $\texttt{stack}_{cntr} > 0$;

$TrEdge(m, l, [l'])$

<u>block_l_False:</u>

assume $\neg(\texttt{stack}_{cntr} > 0)$;

ireturn: The return instruction copies the top stack element to the bottom of the stack and then jumps to the special block `post`, which asserts the current method's normal postcondition.

#case ***ireturn***

$\texttt{stack}_0$:= $\texttt{stack}_{cntr}$;

goto post;

new: Object creation is translated into applications of the BoogiePL versions of the heap functions `add` and `new`.

#case ***new*** t

heap := add(heap, t);

$\texttt{stack}_{cntr+1}$:= new(heap, t);

getfield: Before reading a field, we assert that the top stack element (the receiver) is not null. Next, the top stack element is replaced by the value of the field in the current heap.

#case ***getfield*** f

assert $\texttt{stack}_{cntr} \neq$ null;

$\texttt{stack}_{cntr}$:= get(heap, instvar($\texttt{stack}_{cntr}$, f));

invokevirtual: We use the variables `pre_heap` and `arg0` to save the old values of the heap and the receiver of the call since they may be used in the postcondition of the callee method. We save the receiver, but not the other arguments because the stack location of the receiver will be overwritten by the method result, whereas the other arguments are preserved. P denotes the number of (implicit and explicit) parameters of the callee method.

The actual method call is translated as follows: We assert that the receiver is not null and that the precondition of the callee holds. Possible side effects of the callee are accounted for by havocing the heap. Since the callee may terminate normally or abruptly, we continue with a non-deterministic **goto**.

For abrupt termination, we erase all information about $\texttt{stack}_0$, which now contains the exception object. For simplicity, we do not consider the throws clause of the callee here, but simply assume that the exception is an allocated object of type `Throwable` (`typeof` yields the runtime type of a value and `<:` denotes the subtype relation). We then assume the exceptional postcondition of the callee and jump to all possible handlers of the exception. A more fine-grained handling of exceptions using throws clauses is straightforward.

For normal termination, we erase all information about the stack location that contains the receiver, because this location will now contain the method result. We then assume the normal postcondition of the callee.

In the translation, we use *args* to refer to the arguments of the call, that is, $args = [\texttt{arg}_0, \texttt{stack}_{cntr-P+2}, \ldots, \texttt{stack}_{cntr}]$.

#case ***invokevirtual*** *callee*

$\quad\quad \texttt{arg}_0 \texttt{ := } \texttt{stack}_{cntr-P+1}\texttt{;}$
$\quad\quad \texttt{pre\_heap := heap;}$
$\quad\quad \textbf{assert}\ \ \texttt{arg}_0 \neq \texttt{null;}$
$\quad\quad \textbf{assert}\ \ \mathit{TrSpec}[\![\mathit{pre}(\mathit{callee}), (\texttt{pre\_heap}, \mathit{args})]\!]$
$\quad\quad \textbf{havoc}\ \texttt{heap;}$
$\quad\quad \textbf{goto}\ \ \texttt{block\_}l\texttt{\_Normal, block\_}l\texttt{\_Exception;}$

$\quad \underline{\texttt{block\_}l\texttt{\_Exception:}}$
$\quad\quad \textbf{havoc}\ \texttt{stack}_0\texttt{;}$
$\quad\quad \textbf{assume}\ \texttt{alloc(stack}_0\texttt{, heap)} \wedge \texttt{typeof(stack}_0\texttt{) <: Throwable;}$
$\quad\quad \textbf{assume}\ \ \mathit{TrSpec}[\![\mathit{post}_{\mathsf{X}}(\mathit{callee}), (\texttt{pre\_heap}, \mathit{args}), (\texttt{heap, stack}_0)]\!]$
$\quad\quad \mathit{TrEdge}(m, l, \mathit{handlers}(m, l, \texttt{Throwable}))$

$\quad \underline{\texttt{block\_}l\texttt{\_Normal:}}$
$\quad\quad \#\text{let}\ \mathit{res} = \texttt{stack}_{cntr-P+1}$
$\quad\quad \textbf{havoc}\ \ \mathit{res}\ \texttt{;}$
$\quad\quad \textbf{assume}\ \ \mathit{TrSpec}[\![\mathit{post}(\mathit{callee}), (\texttt{pre\_heap}, \mathit{args}), (\texttt{heap}, \mathit{res})]\!]$

4.5 Translation of Instruction Sequences

$\mathit{TrInstrSeq}[\![m, l]\!]$ translates the instruction sequence starting at label l until the end of method m. It uses the control flow graph of m to determine whether l is the beginning of a basic block. In case l is the target of a back-edge, we know that l is the entry to a loop. We apply the strategy described by Barnett and Leino [3] to make the loop body represent a general loop iteration. This is done by havocing all variables that are potentially modified by the loop(the heap, the operand stack, and the registers, but not the prestate values of the heap and the parameters) and by assuming the loop invariant.

We then translate the instruction at label l. If the transition from l to its textual successor $l + 1$ is an edge in the control flow graph, we have to end the current block and possibly apply the back-edge elimination using *TrEdge*. Finally, we proceed by translating the rest of the instruction sequence.

$TrInstrSeq[\![m : Method, l : Label]\!] =$
  #if $isEdgeTarget(m, l)$
    <u>block_ l :</u>
    #if $isBackEdgeTarget(m, l)$
      havoc `heap` , $stacks, regs$;
      assume $TrSpec[\![local(m, l), (\texttt{old\_heap}, params), (\texttt{heap}, stacks, regs)]\!]$
    #end if
  #end if
  $TrInstr[\![m, l]\!]$
  #if $isEdge(m, l, l + 1)$
    $TrEdge(m, l, [l + 1])$
  #end if
  $TrInstrSeq[\![m, l + 1]\!]$

4.6 Translation of Methods

The translation of a method is done by a function $Tr[\![m : Method]\!]$. This function generates the signature of the BoogiePL procedure, declares the local variables, and applies the translation function *TrBody* for the method body, which we describe next.

The translation of method body starts with creation of a block `init`, which saves the heap of the prestate for later use in local annotations and postconditions. In addition, it copies the values of the parameters to the register variables. P denotes the number of (implicit and explicit) parameters of the method. Next, we assume the precondition. In the case that the first instruction of the method body is a jump target, we assert the local annotation (if there is any) and finish this block by jumping to the block starting at the first instruction. The instructions of the method body are translated using *TrInstrSeq*. *TrBody* also generates blocks for the method's normal and exceptional postconditions.

$TrBody[\![m : Method]\!] =$
  <u>init:</u>
    `old_heap := heap;`
    #for $i := 0 \dots (P - 1)$:
      $\texttt{reg}_i$:= $\texttt{param}_i$;
    #end for
    assume $TrSpec[\![pre(m), (\texttt{old\_heap}, params)]\!]$;

$\#$if $isEdgeTarget(m, 0)$
$\#$if $isBackEdgeTarget(m, 0)$
assert $TrSpec[\![local(m, 0), (\texttt{old\_heap}, params), (\texttt{heap}, \emptyset, regs)]\!]$
$\#$end if
goto `block_0;`
$\#$end if
$TrInstrSeq[\![m, 0]\!]$

<u>`post:`</u>
assert $TrSpec[\![post(m), (\texttt{old\_heap}, params), (\texttt{heap,stack}_0)]\!]$;
goto;

<u>`post_X:`</u>
assert $TrSpec[\![post_{\mathsf{X}}(m), (\texttt{old\_heap}, params), (\texttt{heap,stack}_0)]\!]$;
goto;

5 Example

We illustrate our translation using method `twice`. Fig. 3 shows the source code, bytecode, and specification table of this method. Fig. 4 shows the result of the translation. The specifications are included as **assume** and **assert** commands. Note that the back-edge from label 8 to 11 is eliminated during translation. Instead, `block_11` starts with by havocing the state and assuming the loop invariant. We verified a typed version of this example in Boogie.

6 Soundness

In this section, we present a soundness theorem and sketch the proof of the main lemma. The full proof is presented in our report [6, Appendix E].

Theorem 6.1 *Let m be a method of a specified bytecode program. If the translation of m into BoogiePL can be verified in the wp-calculus for BoogiePL then m can also be verified in the calculus that operates directly on bytecode.*

This theorem is a consequence of the following lemma.

Lemma 6.2 *Let m be a method of a specified bytecode program and l a label in m. The weakest precondition of the BoogiePL translation of m at the position corresponding to label l implies the weakest precondition of m at label l:*

$$\forall m, l, \rho : (wp_{\mathsf{bpl}}(Tr(m), instrpos(Tr(m), l))(\rho) \Longrightarrow wp_{\mathsf{vc}}(m, l)(map(m, \rho, sh(m, l))))$$

where $instrpos(Tr(m), l)$ yields the position of the first BoogiePL command of $TrInstr(Tr(m), l)$.

Proof: The proof runs by induction on the control flow graph of the bytecode method m, more precisely, on the distance of label l from the end of the method

```
//@ requires x > 0;
//@ ensures \result == \old(x)+\old(x);
int twice(int x){
  int i = x;
  //@ loop_invariant
  //@  x + i == \old(x)+\old(x) ∧ i ≥ 0;
  while(i > 0){
    x++; i−−;
  }
  return x;
}
```

```
0:  iload_0
1:  istore_1
2:  goto 11
5:  iinc  0, 1
8:  iinc  1, 1
11: iload_1
12: ifgt  5
15: iload_0
16: ireturn
```

Specification table:

$$pre(\mathrm{twice})(h_0,[p_0]) \equiv p_0 > 0$$

$$post(\mathrm{twice})(h_0,[p_0])(h,rv) \equiv rv = p_0 + p_0$$

$$local(\mathrm{twice},11)(h_0,[p_0])(h,[s_0],[r_1 :: r_0]) \equiv r_0 + r_1 = p_0 + p_0 \wedge r_1 \geq 0$$

Fig. 3. The annotated Java program and the corresponding bytecode program with specification table

```
implementation twice(param0) returns (result) {
    var stack0, reg0, reg1, old_heap;

init:
    old_heap := heap;
    reg0 := param0;
    assume param0 > 0;                                  // requires
    stack0 := reg0;                                     // 0:   iload_0
    reg1 := stack0;                                     // 1:   istore_1
    assert reg0 + reg1 == param0 + param0 ∧ reg1 ≥ 0;   // loop_invariant
    goto block_11;                                      // 2:   goto    11

block_5:
    reg0 := reg0 + 1;                                   // 5:   iinc    0, 1
    reg1 := reg1 − 1;                                   // 8:   iinc    1, 1
    assert reg0 + reg1 == param0 + param0 ∧ reg1 ≥ 0;   // loop_invariant
    return;

block_11:
    havoc stack0, reg0, reg1;
    assume reg0 + reg1 == param0 + param0 ∧ reg1 ≥ 0;   // loop_invariant
    stack0 := reg1;                                     // 11:  iload_1
    goto block_12_true, block_12_false ;                // 12:  ifgt    5

block_12_true:
    assume stack0 > 0;
    goto block_5;

block_12_false:
    assume !(stack0 > 0);
    goto block_15;

block_15:
    stack0 := reg0;                                     // 15:  iload_0
    result := stack0;                                   // 16:  ireturn
    goto post;

post:
    assert result == param0 + param0;                   // ensures
    return;
}
```

Fig. 4. BoogiePL translation of the example with highlighted annotation parts.

not considering back-edges. The base case covers all instructions that terminate the method, in particular, `ireturn`. All other instructions are covered by the induction step. In the following, we show one case of the induction step, namely the case where the instruction at label l is $\texttt{iload\_}n$.

According to the definition of $\mathit{TrInstrSeq}(m, l)$, we have to consider three parts of the translation of the instruction: (1) If there is at least one back-edge pointing to l, the local state is havoced and the loop invariant assumed. (2) The instruction $\texttt{iload\_}n$ itself is translated. (3) Depending on the control flow graph, the BoogiePL code for the instruction is followed by a suffix.

By definition, *instrpos* yields the first position of part (2). Part (1) is needed to show the proof obligations stemming from loop invariants (see last paragraph of Sec. 2.4), but not relevant for this lemma. Therefore, we consider the output of $\mathit{TrInstr}(Tr(m), l)$ in the following. We start by expanding the left-hand side of the implication:

$$
\begin{aligned}
&\mathit{wp}_{\mathsf{bpl}}(\mathit{Tr}(m), \mathit{instrpos}(\mathit{Tr}(m), l))(\rho) \equiv \\
&\mathit{wp}_{\mathsf{bpl}}(\mathit{Tr}(m), \mathit{instrpos}(\mathit{Tr}(m), l) + 1)[\texttt{reg}_n/\texttt{stack}_{cntr+1}](\rho) \equiv \\
&\mathit{wp}_{\mathsf{bpl}}(\mathit{Tr}(m), \mathit{instrpos}(\mathit{Tr}(m), l) + 1)(\rho[\texttt{stack}_{cntr+1} \mapsto \rho(\texttt{reg}_n)])
\end{aligned}
$$

For part (3), we have to consider four cases: (a) there is a back-edge from l to $l+1$; (b) there is a forward-edge from l to $l+1$, and there is at least one back-edge pointing to $l+1$; (c) there is a forward-edge from l to $l+1$, but $l+1$ is not the target of a back-edge; (d) the transition from l to $l+1$ is not an edge in the control flow graph, that is, both labels belong to the same basic block. We continue by case distinction.

Case (a): The translation asserts the loop invariant and returns. In the following, we abbreviate $\rho[\texttt{stack}_{cntr+1} \mapsto \rho(\texttt{reg}_n)]$ by ρ'. By the definition of *TrEdge*, we get:

$$
\begin{aligned}
&\mathit{wp}_{\mathsf{bpl}}(\mathit{Tr}(m), \mathit{instrpos}(\mathit{Tr}(m), l) + 1)(\rho') \equiv \\
&\mathit{local}(m, l+1)(\mathit{map}(m, \rho', \mathit{sh}(m, l+1))) \equiv \\
&\mathit{wp}_{\mathsf{l}}(m, l+1)(\mathit{map}(m, \rho', \mathit{sh}(m, l+1))) \equiv \\
&\mathit{wp}_{\mathsf{vc}}(m, l)(\mathit{map}(m, \rho, \mathit{sh}(m, l)))
\end{aligned}
$$

Case (b): The translation asserts the loop invariant and jumps to the next instruction. So we get:

$$
\begin{aligned}
&\mathit{wp}_{\mathsf{bpl}}(\mathit{Tr}(m), \mathit{instrpos}(\mathit{Tr}(m), l) + 1)(\rho') \equiv \\
&\mathit{local}(m, l+1)(\mathit{map}(m, \rho', \mathit{sh}(m, l+1))) \wedge \mathit{wp}_{\mathsf{bpl}}(\mathit{Tr}(m), \mathit{pos}(\mathit{Tr}(m), l+1))(\rho')
\end{aligned}
$$

where $\mathit{pos}(\mathit{Tr}(m), l+1)$ is the position of the first command of $\mathit{TrInstrSeq}(m, l+1)$. Since $\mathit{wp}_{\mathsf{vc}}$ handles case (b) exactly like case (a), the above conjunction implies $\mathit{wp}_{\mathsf{vc}}(m, l)(\mathit{map}(m, \rho, \mathit{sh}(m, l)))$.

Case (c): The translation generates the command `goto` $\mathit{pos}(\mathit{Tr}(m), l+1)$ at position $\mathit{instrpos}(\mathit{Tr}(m), l) + 1$. Therefore, we have:

$$
\begin{aligned}
&\mathit{wp}_{\mathsf{bpl}}(\mathit{Tr}(m), \mathit{instrpos}(\mathit{Tr}(m), l) + 1)(\rho') \equiv \\
&\mathit{wp}_{\mathsf{bpl}}(\mathit{Tr}(m), \mathit{pos}(\mathit{Tr}(m), l+1))(\rho')
\end{aligned}
$$

Since $l + 1$ is not the target of a back-edge, $pos(Tr(m), l + 1)$ is the same position as $instrpos(Tr(m), l + 1)$, and we can apply the induction hypothesis:

$$\begin{aligned}
&wp_{\mathsf{bpl}}(Tr(m), pos(Tr(m), l + 1))(\rho') \Longrightarrow \\
&wp_{\mathsf{vc}}(m, l + 1)(map(m, \rho', sh(m, l + 1))) \equiv \\
&wp_{\mathsf{l}}(m, l + 1)(map(m, \rho', sh(m, l + 1))) \equiv \\
&wp_{\mathsf{vc}}(m, l)(map(m, \rho, sh(m, l)))
\end{aligned}$$

Case (d): This case does not generate any suffix after the translation of the instruction. Thus, we have $pos(Tr(m), l + 1) = instrpos(Tr(m), l) + 1$. The rest of this case is analogous to case (c). □

7 Related Work

ESC/Java [10] uses guarded commands as intermediate representation. Both Krakatoa [12] and Caduceus [9] translate programs into the Why language. To our knowledge, none of these translations have been formalized and verified. We expect that our work can adapted to these translations, although the treatment of source programs instead of bytecode will require changes of the technical details.

The Boogie verifier [2] translates annotated CIL code to BoogiePL. Some aspects of this translation have been proven sound, for instance, the back-edge elimination and the translation of the statements that manipulate the heap. Our formalization and proof cover a much larger language subset, in particular, exceptions, which are not yet handled by Boogie.

Barnett and Leino [3] present a passification for BoogiePL and prove soundness. In combination with our results, this shows that the translation of bytecode to passive BoogiePL is sound.

Our wp-calculus for bytecode is inspired by Grégoire [6]. We expect that our formalization can be adapted easily to other bytecode logics [1,4,5]

8 Conclusions

We have formalized a translation of a small subset of Java bytecode to BoogiePL and proved soundness of this translation. This work closes a gap in the soundness argument of several program verifiers. We managed to keep the complexity of the translation and proof reasonable by using the identical heap model and a very similar state model for bytecode and the BoogiePL translation. Moreover, our translation relies on the guarantees and information given by the bytecode verifier.

As future work, we plan to extend the translation to cover the whole set of Java bytecode instructions. A more long-term goal is to use the translation from bytecode to BoogiePL as part of a Proof-Carrying Code architecture, which will make a formal soundness proof even more important.

Acknowledgments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

[1] F. Y. Bannwart and P. Müller. A program logic for bytecode. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, volume 141 of *ENTCS*, pages 255–273. Elsevier Science, Inc., 2005.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Object (FMCO)*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer-Verlag, 2006.

[3] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Program Analysis For Software Tools and Engineering (PASTE)*, pages 82–87. ACM Press, 2005.

[4] N. Benton. A Typed, Compositional Logic for a Stack-Based Abstract Machine. In K. Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS'05, 2005*, volume 3780 of *Lecture Notes in Computer Science*, pages 364–380. Springer–Verlag, 2005.

[5] L. Beringer, K. MacKenzie, and I. Stark. Grail: a functional form for imperative mobile code. In *Foundations of Global Computing*, number 85.1 in ENTCS, pages 1–21. Elsevier Science, Inc., 2003.

[6] Mobius Consortium. Deliverable 3.1: Bytecode specification language and program logic. Available online from http://mobius.inria.fr, 2006.

[7] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[8] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, 2003.

[9] J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer-Verlag, 2004.

[10] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI)*, volume 37(5) of *SIGPLAN Notices*, pages 234–245. ACM, 2002.

[11] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Principles of programming languages (POPL)*, pages 193–205. ACM Press, 2001.

[12] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.

[13] A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods*, 1998.

[14] A. Suzuki. Translating Java bytecode to BoogiePL. Master's thesis, ETH Zurich, 2006. www.sct.inf.ethz.ch/projects/student_docs/Alex_Suzuki.