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

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Formal Translation of Bytecode into BoogiePL

# Hermann Lehner[1](#_bookmark0) and Peter Mu¨ller[2](#_bookmark0)

*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 ver- ification 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

# Introduction

Many modern program verifiers such as ESC/Java [[10](#_bookmark23)], Boogie [[2](#_bookmark14)], Krakatoa [[12](#_bookmark25)], and Caduceus [[9](#_bookmark21)] verify programs in two steps. First, they translate the program and the specification into an intermediate representation such as guarded com- mands, BoogiePL [[7](#_bookmark20)], or the Why language [[8](#_bookmark22)]. In the second step, they compute verification conditions for the intermediate representation and pass them to a theo- rem prover. Using an intermediate language improves the interoperability of tools. For instance, Krakatoa and Caduceus translate Java and C code to the Why lan- guage, which allows them to share the Why back end. Moreover, simple intermedi- ate representations facilitate the generation of small verification conditions through passification [[11](#_bookmark24)].

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](mailto:hermann.lehner@inf.ethz.ch)

2 [peter.mueller@inf.ethz.ch](mailto:peter.mueller@inf.ethz.ch)

1571-0661 © 2007 Elsevier B.V. Open access under [CC BY-NC-ND license](http://creativecommons.org/licenses/by-nc-nd/3.0/).

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 transla- tion 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](#_bookmark19),[14](#_bookmark27)] 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](#_bookmark1) describes the bytecode subset and a weakest precondition calculus that directly operates on bytecode. Sec. [3](#_bookmark4) introduces BoogiePL including a weakest precondition calculus. The translation from bytecode to BoogiePL is defined in Sec. [4](#_bookmark5). We illustrate the translation by an example in Sec. [5](#_bookmark8) and prove soundness in Sec. [6](#_bookmark9). We review related work in Sec. [7](#_bookmark12) and offer conclusions in Sec. [8](#_bookmark13).

# 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.

* 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 remain- ing 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 ‘⊥’ is used to indicate that an exception might not be handled by the current method:

*handlers* : *Method* × *Label* × *TypID* → *set of Label*

* 1. *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* × *FieldID* → *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:

*update* : *Heap* × *InstVar* × *Value* → *Heap*

|  |  |  |  |
| --- | --- | --- | --- |
| *get* : | *Heap* × *InstVar* | → *Value* | *alloc* : *Value* × *Heap* → *bool* |
| *add* : | *Heap* × *TypID* | → *Heap* | *new* : *Heap* × *TypID* → *Value* |

*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 Mu¨ller [[13](#_bookmark26)].

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.

*PreState* : (*Heap* × *Arguments*)

*LocalState* : (*Heap* × *OperandStack* × *Locals* )

*PostState* : (*Heap* × *Value*)

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

* 1. *Speciﬁcations*

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

*pre* : *Method* → (*PreState* → *bool* )

*post* : *Method* → (*PreState* × *PostState* → *bool* ) *post* X : *Method* → (*PreState* × *PostState* → *bool* ) *local* : *Method* × *Label* → (*PreState* × *LocalState* → *bool* )

*pre* (*m*) yields the precondition of a method *m*, which is a predicate over a *PreState* . *post* (*m*) and *post* X(*m*) yield the normal and exceptional postcondition of *m*, respec- tively. 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 *local* (*m, l*). They are predicates over a *PreState* and a *LocalState* . *Local anno- tations* 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. *local* (*m, l*) is undefined if *l* is not the beginning of a loop.

* 1. *Direct Veriﬁcation Condition Generation for Bytecode*

Our weakest precondition calculus for bytecode is a simplified version of the calculus by Gr´egoire, which is proven sound w.r.t. an operational semantics [[6](#_bookmark19)]. 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 *wp*vc : *Method* × *Label* → (*PreState* × *LocalState* → *bool* ). *wp*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:

*wp*l(*m, l*)= *local* (*m, l*) : if *local* (*m, l*) is defined

*wp*vc(*m, l*) : otherwise

The weakest precondition function *wp*vc is defined in Fig. [1](#_bookmark3). For iload n, *wp*vc applies the local precondition of the successor instruction to an adapted local state. For ifgt, *wp*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*

iload *n*

ifgt *l* '

ireturn new *t* getfield *f*

*wp*vc(*m, l*)(*σ*0*,* (*h, top* :: *os , reg* )) =

*wp*l(*m, l* + *1* )(*σ*0*,* (*h, reg* [*n*] :: *top* :: *os, reg* ))

(*top >* 0 ⇒ *wp*l(*m, l* ')(*σ*0*,* (*h, os , reg* ))) ∧

(*top* ≤ 0 ⇒ *wp*l(*m, l* + *1* )(*σ*0*,* (*h, os , reg* )))

*post* (*m*)(*σ*0*,* (*h, top* ))

*wp*l(*m, l* + *1* )(*σ*0*,* (*add* (*h, t*)*, new* (*h, t* ) :: *top* :: *os, reg* ))

*top* /= *null* ∧ *wp*l(*m, l* + *1* )(*σ*0*,* (*h, get* (*h, iv* (*top, f* )) :: *os, reg* ))

invokevirtual *t top* /= *null* ∧ *pre* (*t*)(*h, top* )∧

∀ *h*'*, rv* :(

*post* (*t*)((*h, par* )*,* (*h*'*, rv* )) ⇒

*wp*l(*m, l* + *1* )(*σ*0*,* (*h*'*, rv* :: *os, reg* )

) ∧ (

*post* X(*t*)((*h, par* )*,* (*h*'*, rv* )) ⇒

*wp*l(*m, li*)(*σ*0*,* (*h*'*, rv , reg* ))

*li*∈*handlers* (*m,l,typeof* (*rv*))

)

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 ⊥ if the exception might be propagated. Therefore, we define *wp*vc(*m,* ⊥)= *post* 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*l(*m, l*) implies the weakest precondition *wp*vc(*m, l*). The latter obligation is required to show that loop invariants are actually maintained.

# 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](#_bookmark20)]. To focus on the essentials of the bytecode translation, we use an untyped version of BoogiePL in this paper. However, our full formalization [[6](#_bookmark19)] works with the typed language.

* 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*bpl contains all possible values of a BoogiePL program: *State* bpl : *Var* '→ *Value*bpl

* 1. *Veriﬁcation Condition Generation for BoogiePL*

Our wp-calculus for BoogiePL is similar to the one by Barnett and Leino [[3](#_bookmark15)]. How- ever, 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 inter- nally into an acyclic graph. We make this transformation explicit in our translation, see Sec. [4.3](#_bookmark7).

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*bpl : *Method* × *Position* → (*State* bpl → *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](#_bookmark6).

# Translation from Bytecode to BoogiePL

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

Command at *pos* assume *P* assert *P*

*x* := *e*

havoc *x*

*wp*bpl(*m, pos* ) =

*P* ⇒ *wp*bpl(*m, pos* + 1) *P* ∧ *wp*bpl(*m, pos* + 1) *wp*bpl(*m, pos* + 1)[*e/x*]

∀ *x* : *wp*bpl(*m, pos* + 1)

goto *pos1 ,... , posn*

*i*:=1*..n*

*wp*bpl(*m, posi* )

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

* 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.

*sh* : *Method* × *Label* → *int*

*isEdge* : *Method* × *Label* × *Label* → *bool isEdgeTarget* : *Method* × *Label* → *bool isBackEdge* : *Method* × *Label* × *Label* → *bool isBackEdgeTarget* : *Method* × *Label* → *bool*

*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.

* 1. *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](#_bookmark1) 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, param*i* (*i* = 0*,... ,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 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 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 stack*i* and reg*i* variables we have to declare in the BoogiePL procedure.

We use the following abbreviations: *params* for list of variables param*i*, *stacks* for all stack elements stack*i*, and *regs* for all registers reg*i*.

The state of a bytecode program and its translation are formally related by the mapping function *map* : *Method* × *State*bpl × *Int* → (*PreState* × *LocalState* ):

*map*(*m, ρ, h*)=

*ρ*(old heap)*, ρ*(*params* ) *,* *ρ*(heap)*, ρ*(stack*h*)*,... , ρ*(stack0)*, ρ*(*regs* )

* 1. *Back-Edge Elimination*

As explained in Sec. [2.4](#_bookmark2), 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 in- structions. 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](#_bookmark10) 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 gen- erate 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* : list of *Label* ]] = #foreach *l*' in *ls*

#if *isBackEdgeTarget* (*m, l*')

assert *TrSpec*[[*local* (*m, l*')*,* (old heap*, params* )*,* (heap*, stacks, regs* )]] #end if

#end foreach

#let *l1 ,... , ln* = all *l*' in *ls* where ¬*isBackEdge*(*m, l, l*')

goto block *l1* , *.. .*, block *ln*;

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 correspond- ing 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 block *l* .

* 1. *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* : *M ethod, 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*

stack*cntr*+1 := reg*n*;

*ifgt*: A branch instruction is translated into a non-deterministic goto to two suc- cessor 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;

block *l* True:

assume stack*cntr >* 0;

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

block *l* False:

assume ч(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*

stack0 := 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*); 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 stack*cntr* /= null;

stack*cntr* := get(heap, instvar(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 stack0, which now con- tains 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* = [arg0, stack*cntr*−*P* +2*,... ,*stack*cntr* ].

#case *invokevirtual callee*

arg0 := stack*cntr*−*P* +1;

pre heap := heap;

assert arg0 /= null;

assert *TrSpec*[[*pre* (*callee* )*,* (pre heap*, args* )]]

havoc heap;

goto block *l* Normal, block *l* Exception;

block *l* Exception:

havoc stack0;

assume alloc(stack0, heap) Λ typeof(stack0) <: Throwable; assume *TrSpec*[[*post* X(*callee* )*,* (pre heap*, args* )*,* (heap, stack0)]] *TrEdge*(*m, l, handlers* (*m, l,* Throwable))

block *l* Normal:

#let *res* =stack*cntr*−*P* +1

havoc *res* ;

assume *TrSpec*[[*post* (*callee* )*,* (pre heap*, args* )*,* (heap*, res* )]]

* 1. *Translation of Instruction Sequences*

*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](#_bookmark15)]

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* : *M ethod, l* : *Label*]] = #if *isEdgeTarget* (*m, l*)

block *l* :

#if *isBackEdgeTarget* (*m, l*)

havoc heap *, stacks, regs* ;

assume *TrSpec*[[*local* (*m, l*)*,* (old heap*, params* )*,* (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* ]]

* 1. *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* : *M ethod*]] =

init:

old heap := heap;

#for *i* := 0 *...* (*P* − 1) :

reg*i* := param*i*;

#end for

assume *TrSpec*[[*pre* (*m*)*,* (old heap*, params* )]];

#if *isEdgeTarget* (*m,* 0)

#if *isBackEdgeTarget* (*m,* 0)

assert *TrSpec*[[*local* (*m,* 0)*,* (old heap*, params* )*,* (heap*,* ∅*, regs* )]] #end if

goto block 0;

#end if

*TrInstrSeq* [[*m,* 0]]

post:

assert *TrSpec*[[*post* (*m*)*,* (old heap*, params* )*,* (heap,stack0)]] ;

goto;

post X:

assert *TrSpec*[[*post* X(*m*)*,* (old heap*, params* )*,* (heap,stack0)]] ;

goto;

# Example

We illustrate our translation using method twice. Fig. [3](#_bookmark10) shows the source code, bytecode, and specification table of this method. Fig. [4](#_bookmark11) 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.

# 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](#_bookmark19), 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 corre- sponding to label l implies the weakest precondition of m at label l:*

∀*m, l, ρ* : (*wp*bpl(*T r*(*m*)*, instrpos* (*Tr*(*m*)*, l*))(*ρ*) =⇒ *wp*vc(*m, l*)(*map*(*m, ρ, 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* (twice)(*h*0*,* [*p*0]) ≡ *p*0 *>* 0

*post* (twice)(*h*0*,* [*p*0])(*h, rv*) ≡ *rv* = *p*0 + *p*0

*local* (twice*,* 11)(*h*0*,* [*p*0])(*h,* [*s*0]*,* [*r*1 :: *r*0]) ≡

*r*0 + *r*1 = *p*0 + *p*0 Λ *r*1 ≥ 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;

reg1 := stack0;

*// 0: iload 0*

*// 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*

return; block 11:

havoc stack0, reg0, reg1;

assert reg0 + reg1 == param0 + param0 ∧ reg1 ≥ 0; *// loop invariant*

assume reg0 + reg1 == param0 + param0 ∧ reg1 ≥ 0; *// loop invariant*

stack0 := reg1;

goto block 12 true, block 12 false ;

*// 11: iload 1*

*// 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; result := stack0; goto post;

post:

*// 15:*

*// 16:*

*iload 0 ireturn*

return;

}

assert result == param0 + param0; *// ensures*

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 iload *n*.

According to the definition of *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 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](#_bookmark2)), but not relevant for this lemma. Therefore, we consider the output of *TrInstr* (*Tr*(*m*)*, l*) in the following. We start by expanding the left-hand side of the implication:

*wp*bpl(*Tr* (*m*)*, instrpos* (*Tr* (*m*)*, l*))(*ρ*) ≡

*wp*bpl(*Tr* (*m*)*, instrpos* (*Tr* (*m*)*, l*)+ 1)[reg*n/*stack*cntr*+1](*ρ*) ≡

*wp*bpl(*Tr* (*m*)*, instrpos* (*Tr* (*m*)*, l*)+ 1)(*ρ*[stack*cntr*+1 '→ *ρ*(reg*n*)])

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 follow- ing, we abbreviate *ρ*[stack*cntr*+1 '→ *ρ*(reg*n*)] by *ρ*'. By the definition of *TrEdge*, we get:

*wp*bpl(*Tr* (*m*)*, instrpos* (*Tr* (*m*)*, l*)+ 1)(*ρ*') ≡

*local* (*m, l* + *1* )(*map*(*m, ρ*'*, sh*(*m, l* + *1* ))) ≡

*wp*l(*m, l* + *1* )(*map*(*m, ρ*'*, sh*(*m, l* + *1* ))) ≡

*wp*vc(*m, l*)(*map* (*m, ρ, sh*(*m, l*)))

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

*wp*bpl(*Tr* (*m*)*, instrpos* (*Tr* (*m*)*, l*)+ 1)(*ρ*') ≡

*local* (*m, l* + *1* )(*map*(*m, ρ*'*, sh*(*m, l* + *1* ))) Λ *wp*bpl(*Tr* (*m*)*, pos* (*Tr* (*m*)*, l* + *1* ))(*ρ*')

where *pos* (*Tr* (*m*)*, l* + *1* ) is the position of the first command of *TrInstrSeq* (*m, l* + *1* ). Since *wp*vc handles case (b) exactly like case (a), the above conjunction implies *wp*vc(*m, l*)(*map*(*m, ρ, sh*(*m, l*))).

*Case (c)*: The translation generates the command goto *pos* (*Tr* (*m*)*, l* + *1* ) at

position *instrpos* (*Tr* (*m*)*, l*) + 1. Therefore, we have:

*wp*bpl(*Tr* (*m*)*, instrpos* (*Tr* (*m*)*, l*)+ 1)(*ρ*') ≡

*wp*bpl(*Tr* (*m*)*, pos* (*Tr* (*m*)*, l* + *1* ))(*ρ*')

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:

*wp*bpl(*Tr* (*m*)*, pos* (*Tr* (*m*)*, l* + *1* ))(*ρ*') =⇒

*wp*vc(*m, l* + *1* )(*map*(*m, ρ*'*, sh*(*m, l* + *1* ))) ≡

*wp*l(*m, l* + *1* )(*map*(*m, ρ*'*, sh*(*m, l* + *1* ))) ≡

*wp*vc(*m, l*)(*map*(*m, ρ, sh*(*m, l*)))

*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).

# Related Work

ESC/Java [[10](#_bookmark23)] uses guarded commands as intermediate representation. Both Kraka- toa [[12](#_bookmark25)] and Caduceus [[9](#_bookmark21)] translate programs into the Why language. To our knowl- edge, none of these translations have been formalized and verified. We expect that our work can adapted to these translations, although the treatment of source pro- grams instead of bytecode will require changes of the technical details.

The Boogie verifier [[2](#_bookmark14)] 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](#_bookmark15)] 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 Gr´egoire [[6](#_bookmark19)]. We expect that our formalization can be adapted easily to other bytecode logics [[1](#_bookmark16),[4](#_bookmark17),[5](#_bookmark18)]

# 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 Tech- nologies under the IST-2005-015905 MOBIUS project.

# References

1. F. Y. Bannwart and P. Mu¨ller. 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.

1. Mobius Consortium. Deliverable 3.1: Bytecode specification language and program logic. Available online from [http://mobius.inria.fr](http://mobius.inria.fr/) , 2006.
2. 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.
3. J.-C. Filliˆatre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Universit´e Paris Sud, 2003.
4. J.-C. Filliˆatre and C. March´e. 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.
5. 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.
6. 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.
7. C. March´e, 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.
8. A. Poetzsch-Heffter and P. Mu¨ller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods*, 1998.
9. A. Suzuki. Translating Java bytecode to BoogiePL. Master’s thesis, ETH Zurich, 2006.

[www.sct.inf.ethz.ch/projects/student docs/Alex Suzuki](http://www.sct.inf.ethz.ch/projects/student_docs/Alex_Suzuki).