A Logic for Bytecode

Fabian Yves Bannwart and Peter Müller

August 2, 2004

Contents

Abstract

Firstly, this technical report presents a Hoare-style programming logic ("axiomatic semantics") for a sequential, stack-based bytecode language with unstructured control flow and OO-features similar to the JVM or the CLI languages. We prove soundness and completeness with respect to the operational semantics and derive a weakest precondition calculus that does not sacrifice modular reasoning. We then extend the bytecode language and its logic to include structured exception handling and class initialization and we show how the weakest precondition calculus can be used to trivially derive provably correct rules for most JVM and many CLI instructions.

1 Introduction

This technical report defines a Hoare programming logic for a simple, sequential, stack based bytecode language with objects and dynamic dispatch. For the purpose of our explanations, we shall call the virtual machine for this bytecode language $\mathrm{VM_K}$. This machine is similar to the JVM or the CLI. We also present an operational semantics for the $\mathrm{VM_K}$ bytecode.

The instruction set is small to keep the logic simple, but large enough to show all the problems that occur when specifying a bytecode language for an existing virtual machine. In fact, most JVM and many CLI data structures and instructions can be easily translated to the restricted set we are reasoning about in this paper. We will see how the translation of these "compound instructions" together with the special shape of the programming logic can be used to trivially extend the logic to include such instructions. (section 5 on page 46)

The formalisms for the bytecode logic loosely follow [Ben04] for individual instructions. Objects and dynamic dispatch are treated similarly as in [PHM99].

Unlike the logic in [Ben04], we do not merge specification and typing information in our bytecode logic. But we naturally require certain well-typedness conditions.

This simple well-formedness can be checked by a "bytecode verifier". This separation of the verification process is necessary to keep the programming logic manageable: Properties that can be easily checked by a verifier[1] are complicated enough that they should be reasoned about separately such that its result can form a *basis* for the more complicated behavioral correctness proofs of a program.

[1] e.g. are all variables definitely assigned when they are used, are there enough values on the stack for the instructions and do they contain values of the right type for the operations applied to them, are all statements reachable?

1.1 Why is a Bytecode Logic Needed?

Is it necessary to have a logic for bytecode programs? The ideas of proof-carrying code (PCC, [Nec97]) are already old. "Untrusted code" is augmented with information (the proof) that can render its (type-) safety checkable. The obligation to provide a proof for the safety of a program is deferred to the code producer. The only thing the user of the code has to do is checking the proof that comes with the program. Checking a proof is comparably simple. This procedure allows mobile code to be executed directly and without expensive runtime checks.

Unfortunately, properties described by PCC are typically limited to simple well-typedness of the binary code. PCC is used for hardware platforms, where well-typedness is not guaranteed. Because the proofs are simple, a *certifying compiler* can add a proof when compiling a program. For virtual machines like the JVM or the CLI, the bytecode verifier *automatically* ensures the type-safety of a program.

What is more, in order to show that programs and especially program *components* do the right things, it is just not enough to show that they do the things right, which is what PCC can guarantee. For complicated properties like adherence to an interface specification, we need a formal program proof. But program proofs are at most available on the level of the source code. They cannot be used for component- or class libraries that are shipped or sent over a network as bytecode.

Our arguments illustrate the necessity of two elements for a "proven components industry":

1. A logic for bytecode verification, to be able to show the correctness of bytecode programs.
2. An automatic translation from source code proofs to this bytecode logic by proof transforming compilers. No one will be willing to prove compiler output and manual intervention is still necessary for formal program verification. We have already implemented a very simple version of a proof transforming compiler for a subset of Java to a logic similar to the one presented here in this report. [Ban04]

With introduction of the CLI, the idea of bytecode being a device for language interoperability has gained ground. By the definition of how source code is translated to bytecode, we define at the same time how source programs written in different languages can inter-operate. This fact defines another and important application of any bytecode logic and its corresponding translation procedures of source level proofs into that logic: To define a common semantics for specifications written in different languages. A bytecode logic can guarantee that correctness properties survive and are even formally accessible across language boundaries.

It is sometimes be necessary to program directly in a bytecode language – most often when writing embedded applications. That's why it is important to develop an intuitive bytecode logic that can also be used directly and for which tool support is feasible.

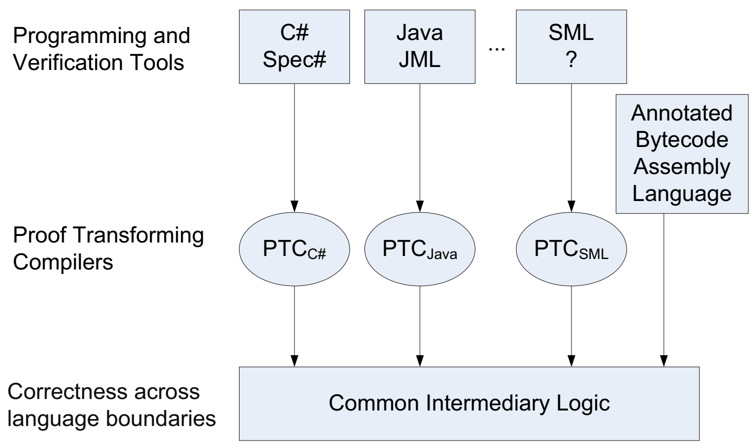

Figure 1: The vision of the trusted components market. Components are proven on the source level and then translated – together with their proof – to an intermediary bytecode language that now guarantees correctness across language boundaries *beyond* operational interoperability.

1.1.1 Why Another Operational Semantics for Bytecode?

In spite of the amazing number of operational semantics for bytecode ([HM01], [SBS01]) that have been developed, we are introducing another, new semantics for the sake of crafting a bytecode logic. Why? Most semantics pretend to be close to the actual virtual machines while ignoring important aspects of the concrete machine like class initialization and garbage collection (finalizers). This is not truly prohibitive however. The fundamental reason not to use any of the existing semantics are the following:

- Current operational semantics model the stack of procedure activation frames explicitly precluding easy comparison with modular source level programming logics. This is helpful when constructing a bytecode logic that should be easy to translate to from a source logic.
- We want a *layered architecture* that is easy to understand and easy to reason about. We use a small set of simple instructions. Most of them are not instructions of a real virtual machine. But most real JVM/CLI instructions can be trivially translated to $\mathrm{VM_K}$ instructions. Giving an operational semantics for complex JVM and especially CLI instructions

directly is awkward and less intuitive than giving a translation to primitive instructions that are easy to understand.

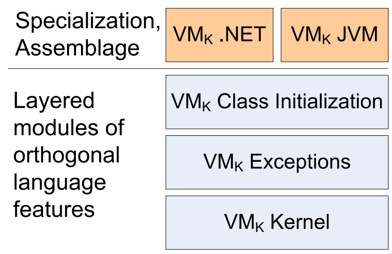

Figure 2: The overall architecture of the bytecode logic. The $\mathrm{VM_K}$ kernel contains a small set of instructions. Most of them are either generalized instructions like $\mathtt{op}_{\mathrm{op}}$ that can stand for any operation op of any arity or more primitive than the instructions that can be found on a real machine. "$\mathrm{VM_K}$ Kernel" is discussed in section 3 on page 15. Tailoring the logic for formally verifying CLI or JVM programs can be done by specialization and assemblage of $\mathrm{VM_K}$ instructions. Specialization is trivial. Assembling instructions to more complex ones is discussed in section 5 on page 46. The $\mathrm{VM_K}$ kernel logic itself is split into modular layers of primitive orthogonal language features. Adding one layer changes little in lower layers.

1.2 Omissions

Many JVM/CLI instructions fit into our framework. Exceptions are unmanaged CLI instructions and instructions that can only be used in conjunction with delegates and `out`/`ref` parameters in managed code. Delegates can be translated to interfaces and classes that implement them.[2] `out` and `ref` parameters can be treated just like heap objects (adding another level of indirection). A better idea for effective reasoning may be to treat locals whose address is taken differently. *The concrete treatment of these features depends on the exact nature of the intended application* and is therefore beyond the scope of this report.[3]

1.3 Overview

The sections of this paper are organized as follows: We define an operational semantics for the $\mathrm{VM_K}$bytecode. The operational semantics leads to the Hoare-style program logic for bytecode. Soundness and completeness proofs are given together with a weakest precondition calculus. The logic is then extended to

[2]For reasoning about them, it may be better to introduce another interface type for every single delegate variable declaration because variables of the same delegate type are often used for very different purposes.

[3]e.g, it would be nice to use the same abstractions as a source logic that supports these features in order to make proof translation easier.

include exceptions and class initializers. We show how complex instructions are assembled from simpler ones and how rules for them can be derived.

Readers familiar with bytecode languages may want to skip the overview and the operational semantics in section 2 and start directly with section 3 on page 15 and go back only when needed.

2 The $\mathrm{VM_K}$ Bytecode

In this section, we're describing the design of the $\mathrm{VM_K}$ virtual machine bytecode language and give an operational semantics for it. As indicated in the introduction, $\mathrm{VM_K}$ is a machine

- with unrestricted control flow expressed using conditional and unconditional jumps to *labeled instructions* within a method body.
- $\mathrm{VM_K}$ is stack based. All arithmetic operations operate on this *evaluation stack*. The machine does not impose any limit on the elements that may be pushed onto the stack.
- In addition to the evaluation stack, there are *locals*. They include local variables as well as method parameters.

Definition 1. A $\mathrm{VM_K}$ program consists of a number of classes and interfaces *just like in Java*: The classes are templates for the instantiation of objects with fields and method implementations. Classes are in the usual subtype relation with the interfaces they *implement* and in the subtype and subclass relation with the single class they inherit from. The exception is the class `object` that does not inherit from any other class. Classes can *override* individual methods of super-types that are marked as `virtual`.

Method implementations are sequences of labeled bytecode instructions. The labels are consecutive non-negative integers starting with 0. The operational semantics is normative, but the following list gives an *informal* overview of the instructions available in the $\mathrm{VM_K}$ kernel.

- `pushc` v pushes a constant v onto the stack
- `pushv` x pushes the value of a local variable (or method parameter) onto the stack
- `pop` x pops the top element off the stack and assigns it to the local variable x
- $\mathtt{op}_{\mathrm{op}}$ Assuming that op is a function that takes n input values to m output values, it removes the n top elements from the stack by applying op to them and puts the m output values onto the stack. We write $\mathtt{binop}_{\mathrm{op}}$ if op is a binary function.

Example 1. – `dup` is a abbreviation for $\texttt{op}_{x \mapsto (x,x)}$.

– The JVM instruction `swap` is a abbreviation for $\texttt{op}_{(x,y) \mapsto (y,x)}$.

– The CLI instruction `isinst` T is a abbreviation for $\texttt{op}_{x \mapsto (\tau(x) \preceq T)}$. The τ function maps a value to its type. $\preceq$ is the subtype relation.

- `goto` l transfers control the program point l
- `brtrue` l transfers control the program point l if the top element of the stack is true and unconditionally pops it.
- `checkcast` T checks whether the top element is of type T or a subtype thereof.
- `newobj` T allocates a new object of type T and pushes it onto the stack
- `invokevirtual` M and `call` M invokes the method M on an optional object reference and parameters on the stack and replaces these values by the return value of the invoked method (if M returns a value). `call` invokes non-virtual and static methods, `invokevirtual` invokes virtual methods. The invoked code depends on the actual type of the object reference (dynamic dispatch).
- `getfield` F replaces the top element by its field F
- `putfield` F sets the field F of the object denoted by the second-topmost element to the top element of the stack and pops both values.
- `nop` has no effect

Example 2. The following program is an example of bytecode method implementation fragment that calculates $S = \sum_{i=1}^{n} i$ in a naive manner.

```
 0: pushc 0     // the top of the stack now contains 0
 1: pop S       // store the top of the stack into the local S
 2: goto 11     // unconditional jump to the conditional beginning
                // of the loop that calculates the sum

// beginning of the loop body where n is decremented
// by 1 and S is incremented by n
 3: pushv S     // push the local S onto the stack
 4: pushv n     // stack is now (S, n)
 5: dup         // duplicate the topmost element:  (S, n, n)
 6: pushc 1     // push the constant 1:  (S, n, n, 1)
 7: binop "-"   // in order to decrement n
 8: pop n       // store it back
 9: binop "+"   // add the old n to S...
10: pop S       // ...and store it
                // the evaluation stack is empty again

// we decide here whether there is anything to do
11: pushv n     // n...
12: pushc 0     // ...and 0...
13: binop ">"   // ...are compared
14: brtrue 3    // if it is true that n > 0 then do it again
                // if it is not true, fall through
```

The bytecode program is the translation of the following C♯ fragment:

```
S = 0;
while(n > 0)
  S += n--;
```

Note that although this example bytecode program *does* have a reducible control flow graph, *no structure is required by the bytecode language, the operational semantics or the programming logic we are going to present.* Unstructured programs are treated exactly as programs where high level structures could be rediscovered.

Definition 2. Method bodies can terminate and return a value back to the invoking method. The return value is stored in the special local variable `result`. There is no `return`-like instruction in $\mathrm{VM_K}$.[4]

A method terminates (returns to the caller) when it reaches the instruction beyond the end of its body. We require that it is the special instruction `end_method`. `end_method` halts the execution. The intuition is that it transfers control back to the invoking method. It serves as an *end marker for a method implementation.* There must not be a `end_method` instruction before the end of the method.

Example 3. The following method implementation returns directly to the caller.

```
0: end_method
```

We say that the length of the method is zero. The `end_method` instruction is not considered actual part of the method body but a syntactic trick to allow comfortable formulation of method call semantics.

Definition 3. A method can also take parameters. They are accessed like local variables as $\mathrm{p}_0, \ldots, \mathrm{p}_n$. While we allow an arbitrary number of parameters per method, we will – without loss of generality – only reason about non-static methods with exactly one parameter called p to keep the proofs simple. Likewise, we will only reason about $\mathtt{binop}_{\mathrm{op}}$ and not the more general $\mathtt{op}_{\mathrm{op}}$. The operational semantics therefore covers only these cases.

Example 4. The following method returns its argument, i.e., it is the identity function.

```
0: pushv p
1: pop result
2: end_method
```

Example 5. An example of a function that divides its argument by 5. The state of the evaluation stack after the execution of each instruction is shown. The initial value of the parameter p is denoted by p_0.

[4] We will see in section 6 on page 49 how return-like instructions can be easily formulated as "compound instructions".

```
0: pushv p      // (p0)
1: pushc 5      // (p0,5)
2: binop "/"    // (p0/5)
3: pop result   // ()
4: end_method
```

Definition 4. Instance variables names are written as $Type@fieldname$, methods that are known at compile-time as *Type*@*method* and virtual method identifiers as *Type* : *method*. The body of the method declaration $T@m$ is denoted by $body_{\mathrm{VM_K}}(T@m)$, the implementing method declaration for a virtual method $T : m$ in S (for $S \preceq T$) is $impl(S, T : m)$ or simply $impl(S, m)$.

Example 6.
- The class T has the fields a, b, c. They are referred to as $T@a$, $T@b$, $T@c$, resp.
- The following C♯ class

  ```
  class Turtle{
    ...
    public Turtle(){ ... }
    public virtual void go(){ ... }
    public void home(){ ... }
  }
  ```

  Introduces the method identifiers

  - $Turtle@.ctor$, the instance constructor
  - $Turtle@go$ and $Turtle : go$, the virtual method identifier for $Turtle@go$
  - $Turtle@home$.

Example 7. This dynamically bound method $Factorial@fact$ calculates the factorial $n!$ recursively, assuming the fields N and R are initialized by n and 1 resp. t is a local variable.

```
 0: pushv this               // (this)
 1: getfield Factorial@N     // (this.N)
 2: pushc 0                  // (this.N, 0)
 3: binop "<="               // (this.N ≤ 0)
 4: brtrue 21                // ()

 5: pushv this               // (this)
 6: dup                      // (this, this)
 7: getfield Factorial@R     // (this, this.R)
 8: pushv this               // (this, this.R, this)
 9: dup                      // (this, this.R, this, this)
10: getfield Factorial@N     // (this, this.R, this, this.N)
11: dup                      // (this, this.R, this, this.N, this.N)
12: pop t                    // (this, this.R, this, this.N)
13: pushc 1                  // (this, this.R, this, this.N, 1)
14: binop "-"                // (this, this.R, this, this.N − 1)
15: putfield Factorial@N     // (this, this.R)
16: pushv t                  // (this, this.R, t)
17: binop "*"                // (this, this.R·t)
```

```
18: putfield Factorial@R   // ()
19: pushv this             // (this)
20: invokevirtual Factorial:fact
21: end_method
```

2.1 Operational Semantics

The abstract execution requires an *abstract state* and some *code* we want to execute. First let's repeat and formalize the notion of a method body we have introduced above (section 2 on page 6):

Definition 5. A $\mathrm{VM_K}$ method implementation $\mathfrak{p}$ consists of a sequence of labeled $\mathrm{VM_K}$ instructions.

1. $|\mathfrak{p}|$ is the number of instructions in the method (without the obligatory `end_method` instruction beyond the method body)
2. The labels of the instructions are in $\Lambda_{\mathfrak{p}} = \{0..|\mathfrak{p}|\}$, i.e., the `end_method` instruction is labeled as well.
3. For every label in $\mathfrak{p}$, there is exactly one corresponding instruction.

$$\mathfrak{p}(l) = I_l$$

   Putting it differently, labels are *unique* within an instruction sequence.

The state of a method invocation consists of the locals, the evaluation stack and the object store. To support dynamic allocation and object features, we have to model the object store:

2.1.1 Modeling the Heap

The *object store* $\$: ObjectStore$ is introduced to model the dynamic heap. The object store and some auxiliary functions together support the usual operations that are normally associated with the heap. These operations follow certain intuitive axioms (given in section 3.1 of [PH97]). We do not treat them here, because they are only necessary to actually prove properties about your programs, but not to understand the concept of how the operations work and how they can be used.

- instance variable lookup:

$$\mathrm{iv} : Value \times FieldDeclId \to InstVar$$

  Remember that $FieldDeclId$s are written as $Type@FieldName$ $InstVar$ is the set of instance variables. This is comparable to addresses of heap variables in other models.

- instance variable update:

$$\$\langle f := v\rangle : ObjectStore \times InstVar \times Value \rightarrow ObjectStore$$

updates an $ObjectStore$ \$ and returns a new $ObjectStore$ where the $InstVar$ f has the new value v.

- instance variable load:

$$\$(f) : ObjectStore \times InstVar \rightarrow Value$$

returns the value of an $InstVar$ in \$.

- new object allocation: this function yields the object-store obtained by allocating a new object of type T in \$

$$\$\langle T\rangle : ObjectStore \times ClassTypeId \rightarrow ObjectStore$$

- return a new object of type T

$$new(\$, T) : ObjectStore \times ClassTypeId \rightarrow Value$$

2.1.2 The Abstract State

Definition 6. The configuration (abstract state)

$$K \equiv \langle S, \sigma, l\rangle$$

of a method invocation during execution consists of *the program environment* S, *the evaluation stack* σ and *the program counter* l (the label of the next instruction to be executed).

- The environment S maps variables and the parameters this and p to values and \$ to the current object store

$$\begin{aligned} S &\in State \\ State &\equiv (LocalVariable \cup \{\text{this}, p\} \hookrightarrow Value) \cup (\{\$\} \rightarrow ObjectStore) \end{aligned}$$

As mentioned before, we allow only one parameter in order to simplify the formalism.

- The stack is a list of values

$$\begin{aligned} \sigma &\in Stack \\ Stack &\equiv Value* \end{aligned}$$

- The program counter is always at a valid position

$$l \in \Lambda_{\mathtt{p}}$$

Definition 7. The small step transition relation

$$\mathfrak{p}; \langle S, \sigma, l\rangle \to \langle S', \sigma', l'\rangle$$

means that for the program $\mathfrak{p}$, the machine can go in one step from the state $\langle S, \sigma, l\rangle$ to the $\mathrm{VM_K}$ state $\langle S', \sigma', l'\rangle$.

Note 1. Other common symbols for small step transition relations are

- $\langle \mathfrak{p}, K\rangle \to \langle \mathfrak{p}', K'\rangle$ or
- $\langle \mathfrak{p}, K\rangle \triangleright \langle \mathfrak{p}', K'\rangle$

These relations do not only transform the abstract state K to K', they normally transform the code as well ($\mathfrak{p}$ to $\mathfrak{p}'$). This is not the case with our relation "$\_;\_ \to \_$". $\mathfrak{p}; \langle S, \sigma, l\rangle \to \langle S', \sigma', l'\rangle$ leaves the code $\mathfrak{p}$ as is and operates only on the abstract state. For a given instructions sequence $\mathfrak{p}$, we can then talk about the transition relation "$\to$" from state to state.

For a given $\mathfrak{p}$, the multistep relation $\to^*$ is the reflexive transitive closure of $\to$. The multistep relation frees us from having to model procedure activation frames explicitly.

We can now define the individual instructions of our virtual machine. See partition III of [ECM02] and [LY99] to compare them with the actual instructions of our example VMs. The primary goal of the operational semantics is to allow the soundness and completeness of the axiomatic semantics to be verified. That's why we omit the transition rules for `call` s to static methods, methods with more than one explicit parameter, etc. which do not feature in our soundness/completeness proofs.

2.1.3 Instructions for the Compilation of Expressions

Pushing a Constant onto the Stack: `pushc` v

$$[\ldots\ l : \texttt{pushc}\ v\ \ldots]; \langle S, \sigma, l\rangle \to \langle S, (\sigma, v), l+1\rangle$$

Pushing the Value of a Local Variable onto the Stack: `pushv` x

$$[\ldots\ l : \texttt{pushv}\ x\ \ldots]; \langle S, \sigma, l\rangle \to \langle S, (\sigma, S(x)), l+1\rangle$$

Popping the Stack into a Local Variable: `pop` x

$$[\ldots\ l : \texttt{pop}\ x\ \ldots]; \langle S, (\sigma, v), l\rangle \to \langle S[x \mapsto v], \sigma, l+1\rangle$$

Binary Operations: $\texttt{binop}_{\mathrm{op}}$

$$[\dots\ l : \texttt{binop}_{\mathrm{op}}\ \dots]; \langle S, (\sigma, v_1, v_2), l\rangle \rightarrow \langle S, (\sigma, v_1 \,\mathrm{op}\, v_2), l+1\rangle$$

2.1.4 Instructions that Modify the Control Flow

Conditional Jump: $\texttt{brtrue}\ l'$

$$[\dots\ l : \texttt{brtrue}\ l'\ \dots]; \langle S, (\sigma, true), l\rangle \rightarrow \langle S, \sigma, l'\rangle$$

and

$$[\dots\ l : \texttt{brtrue}\ l'\ \dots]; \langle S, (\sigma, false), l\rangle \rightarrow \langle S, \sigma, l+1\rangle$$

Unconditional Jump: $\texttt{goto}\ l'$

$$[\dots\ l : \texttt{goto}\ l'\ \dots]; \langle S, \sigma, l\rangle \rightarrow \langle S, \sigma, l'\rangle$$

`goto` is not strictly necessary: $\texttt{goto}\ l'$ is equivalent to the sequence

$$\begin{aligned} l_1 &: \texttt{pushc}\ true \\ l_1 + 1 &: \texttt{brtrue}\ l' \end{aligned}$$

In section 5 on page 46, we will look at `goto` as a “compound instruction”.

2.1.5 Instructions for Objects

Casting a Reference: $\texttt{checkcast}\, T$

$$\frac{\tau(v) \preceq T}{[\dots\ l : \texttt{checkcast}\, T\ \dots]; \langle S, (\sigma, v), l\rangle \rightarrow \langle S, (\sigma, v), l+1\rangle}$$

The condition $\tau(v) \preceq T$ ensures that execution gets stuck if the reference is not of the required type. The axiomatization in [PH97] relates the value of the τ function and the allocation primitives $\$\langle T\rangle$ and $\mathrm{new}(\$, T)$.

Object Creation: $\texttt{newobj}\ T$

$$[\dots\ l : \texttt{newobj}\ T\ \dots]; \langle S, \sigma, l\rangle \rightarrow \langle S[\$ \mapsto S(\$)\langle T\rangle], (\sigma, \mathrm{new}(S(\$), T))), l+1\rangle$$

Calling a Virtual Method (with One Argument): `invokevirtual` $T : m$

$$\frac{\begin{array}{c}\mathfrak{p}' = \mathit{body}_{\mathrm{VM_K}}(\mathit{impl}(\tau(y), m)) \qquad \mathfrak{p}'(l') = \texttt{end\_method} \\ \mathfrak{p}'; \langle \{\text{this} \mapsto y, \text{p} \mapsto v, \$ \mapsto S(\$)\}, (), 0\rangle \to^* \langle S', \sigma', l'\rangle \\ S_p = S[\$ \mapsto S'(\$)] \\ \sigma_p = (\sigma, S'(\text{result}))\end{array}}{[\ldots\ l : \texttt{invokevirtual}\ T : m\ \ldots]; \langle S, (\sigma, y, v), l\rangle \to \langle S_p, \sigma_p, l+1\rangle}$$

Non-virtual and virtual methods are treated very similarly – virtual methods are just a bit more complicated. We will thus omit explicit arguments about non-virtual methods.

Loading a Field of an Object: `getfield` $T@a$

$$\frac{y \neq \text{null}}{[\ldots\ l : \texttt{getfield}\ T@a\ \ldots]; \langle S, (\sigma, y), l\rangle \to \langle S, (\sigma, S(\$)(\text{iv}(y, T@a)), l+1\rangle}$$

Storing into a Field of an Object: `putfield` $T@a$

$$\frac{\begin{array}{c}y \neq \text{null} \\ S_p = S[\$ \mapsto S(\$)\langle \text{iv}(y, T@a) := v\rangle]\end{array}}{[\ldots\ l : \texttt{putfield}\ T@a\ \ldots]; \langle S, (\sigma, y, v), l\rangle \to \langle S_p, \sigma, l+1\rangle}$$

2.1.6 Additional Instructions

No Operation: `nop`

$$[\ldots\ l : \texttt{nop}\ \ldots]; \langle S, \sigma, l\rangle \to \langle S, \sigma, l+1\rangle$$

2.1.7 Comments

Observation 1. There is no transition for `end_method`. Execution will stop when reaching an `end_method` instruction.

Observation 2. The operational semantics is deterministic.

There is at most one transition for every statement and program point. And program points are unique in a program.

Observation 3. There should be constraints on the state at a given program point.

Example 8. The effect of an execution step starting in the configuration $[\ldots\ l :$ `pushv` $x\ \ldots]; \langle S', \sigma, l\rangle$ cannot be reasonably legitimated if the variable x is not yet initialized.

The configuration $[\ldots\ l : \texttt{binop}_{\mathrm{op}}\ \ldots]; \langle S', (\sigma, a, b), l\rangle$ should not have a successor if op is not an operation on $\tau(a) \times \tau(b) \to \alpha$ for some type α.

We assume that every $\mathrm{VM_K}$ program satisfies some basic well-formedness constraints that ensure that such situations can never occur.

We do so not only because this is rarely prohibitive and quite usual for real machines – both the JVM and the CLI have bytecode verifiers – but also because this additional abstraction helps us keep the logic we will construct simple. We are ruling out invalid behavior (type errors, popping the empty stack, ...) of our programs before our logic is applied to them. An alternative approach would be to combine type checking[5] and verification. Compare [Ben04] on how this can be done.

It should be noted that type-checking bytecode is itself a non-trivial undertaking: Research on this topic has led to the publication of a vast quantity of articles and several Ph.D. theses. It is only recently that the implications of the complex interplay between unrestricted control flow, exception handling and unstructured subroutines has been completely understood[6]. The approach taken in [Ben04] is therefore unlikely to scale to the extended instruction set of a real virtual machine.

3 A Programming Logic for the $\mathrm{VM_K}$ Kernel Language

The intuitive meaning of the Hoare-triple

$$\{P\}\ \mathrm{comp}\ \{Q\}$$

is that if P holds in some initial state and the execution of comp terminates then Q will hold in the halting state of comp.

We are mainly concerned with the specification and verification of individual methods with pre- and postconditions: given some precondition, what condition will hold when the method terminates? In this case, comp is a *method implementation*. We denoted these method bodies by $\mathtt{p}$:

$$\{P\}\ \mathtt{p}\ \{Q\}$$

As illustrated by the invocation rules (section 2.1.5 on page 13), a method body terminates if and only if control flow reaches the `end_method` instruction.[7]

[5] in a very general sense

[6] [JAR03] gives an overview, [SS03] discusses the problems of bytecode verification

[7] Jumping out of the method body is not possible by the well-formedness condition of the bytecode.

Taking the focus on methods as a motivation, we can extend Hoare-triples to method declaration identifiers that represent a method implementation or a set of method implementations in the case of a virtual method identifiers. We call Hoare triples for method identifiers and method bodies collectively *method specifications*. Specifications of virtual method identifiers capture the common properties of all the overriding implementations in subtypes. Method pre- and postconditions may not reference local variables or stack elements. They may however depend on the object store. The precondition is also allowed to depend on the input parameters. (Remember: A statically bound method m in class T has the method identifier $T@m$. A virtual method m in the context of a class T is denoted by $T : m$)

Example 9. For a statically bound method $T@m$

$$\{P\}\ T@m\ \{Q\}$$

holds if

$$\{P \land \text{this} \neq \text{null}\}\ \mathit{body}_{\mathrm{VM_K}}(T@m)\ \{Q\}$$

I.e., if the triple holds for the method implementation for which we can quite reasonably assume that this $\neq$ null.

Example 10. For a dynamically bound method $T : m$

$$\{P\}\ T : m\ \{Q\}$$

holds if

$$\{P \land \text{this} \neq \text{null}\}\ \mathit{body}_{\mathrm{VM_K}}(S@m)\ \{Q\}$$

holds for all subtypes S of T ($S \preceq T$), i.e. for all concrete implementations of $T : m$

The fact that we are using classical Hoare-triples shows that we also need the usual deduction rules. Because our logic is based on [PHM99], we will need all the statement-independent rules that are introduced there. These rules are then only applicable to method implementations and method identifiers – *individual instructions are treated differently.* For the treatment of recursive methods, we use sequents of the form $\mathcal{A} \vdash \{P\}$ comp $\{Q\}$ where $\mathcal{A}$ is a set of method specifications.

3.1 Rules for Method Specifications

There are two groups of rules for method specifications: general rules that can be found in most axiomatic semantics and method specific rules. The method specific rules just formalize the examples above: A method specification holds for a method identifier if it holds for all implementations. How the required triples can be derived modularly is described in [PHM99] at the end of section 3.

3.1.1 Method Specific Rules

implementation

$$\text{implementation}\frac{\mathcal{A}, \{P\}\ T@m\ \{Q\} \vdash \{P \wedge \text{this} \neq \text{null}\}\ \mathit{body}_{\mathrm{VM_K}}(T@m)\ \{Q\}}{\mathcal{A} \vdash \{P\}\ T@m\ \{Q\}}$$

The following rules are needed to derive virtual method specifications (see [PHM99] for how this can be done):

class

$$\text{class}\frac{\begin{array}{c}\mathcal{A} \vdash \{P \wedge \tau(\text{this}) = T\}\ \mathit{impl}(T, m)\ \{Q\} \\ \mathcal{A} \vdash \{P \wedge \tau(\text{this}) \prec T\}\ T : m\ \{Q\}\end{array}}{\mathcal{A} \vdash \{P \wedge \tau(\text{this}) \preceq T\}\ T@m\ \{Q\}}$$

subtype

$$\text{subtype}\frac{\begin{array}{c}S \preceq T \\ \mathcal{A} \vdash \{P \wedge \tau(\text{this}) \preceq S\}\ S : m\ \{Q\}\end{array}}{\mathcal{A} \vdash \{P \wedge \tau(\text{this}) \preceq S\}\ T@m\ \{Q\}}$$

3.1.2 Method Independent Rules

$$\text{conjunct}\frac{\begin{array}{c}\mathcal{A} \vdash \{P_1\}\ \text{comp}\ \{Q_1\} \\ \mathcal{A} \vdash \{P_2\}\ \text{comp}\ \{Q_2\}\end{array}}{\mathcal{A} \vdash \{P_1 \wedge P_2\}\ \text{comp}\ \{Q_1 \wedge Q_2\}}$$

$$\text{disjunct}\frac{\begin{array}{c}\mathcal{A} \vdash \{P_1\}\ \text{comp}\ \{Q_1\} \\ \mathcal{A} \vdash \{P_2\}\ \text{comp}\ \{Q_2\}\end{array}}{\mathcal{A} \vdash \{P_1 \vee P_2\}\ \text{comp}\ \{Q_1 \vee Q_2\}}$$

$$\text{consequence}\frac{\begin{array}{c}P \Rightarrow P' \qquad Q' \Rightarrow Q \\ \mathcal{A} \vdash \{P'\}\ \text{comp}\ \{Q'\}\end{array}}{\mathcal{A} \vdash \{P\}\ \text{comp}\ \{Q\}}$$

$$\text{inv}\frac{\begin{array}{c}R \text{ doesn't contain references to the program state} \\ \mathcal{A} \vdash \{P\}\ \text{comp}\ \{Q\}\end{array}}{\mathcal{A} \vdash \{P \wedge R\}\ \text{comp}\ \{Q \wedge R\}}$$

$$\text{subst}\frac{\begin{array}{c}t \text{ doesn't contain references to the program state} \\ Z \text{ is a logical variable} \\ \mathcal{A} \vdash \{P\}\ \text{comp}\ \{Q\}\end{array}}{\mathcal{A} \vdash \{P[t/Z]\}\ \text{comp}\ \{Q[t/Z]\}}$$

$$\text{all}\frac{\begin{array}{c} Z, Y \text{ are distinct logical variables} \\ \mathcal{A} \vdash \{P[Y/Z]\} \text{ comp } \{Q\} \end{array}}{\mathcal{A} \vdash \{P[Y/Z]\} \text{ comp } \{\forall Z : Q\}}$$

$$\text{ex}\frac{\begin{array}{c} Z, Y \text{ are distinct logical variables} \\ \mathcal{A} \vdash \{P\} \text{ comp } \{Q[Y/Z]\} \end{array}}{\mathcal{A} \vdash \{\exists Z : P\} \text{ comp } \{Q[Y/Z]\}}$$

3.2 The Specification and Verification of Method Bodies

It is clear that Hoare triples cannot be extended to parts of method bodies – sequences of instructions with unstructured control flow. We don't want to rediscover high level control structures in our code sequences either because this would preclude the verification of arbitrary instruction sequences that do not adhere to any patterns. Instead, we look at only *one instruction at a time*. We don't use pre- and post-condition for every statement like in structured programming languages. We use only *preconditions* for individual instructions in a method body $\mathtt{p}$:

$$\{E_l\}\, l : I_l$$

Obviously, the meaning of the *instruction specification* $\{E_l\}\, l : I_l$ cannot be defined in isolation. The meaning of $\{E_l\}\, l : I_l$ in a method body $\mathtt{p}$ is that if the "labeled assertion" E_l holds when the program counter is just before the instruction (at position l) then the precondition $E_{l'}$ of the successor instruction at label l' will also hold after successful termination of instruction l. By induction on the number of instructions executed, this is equivalent to claiming that the precondition of the `end_method` holds if the method terminates. We have thus already established the necessary connection between method and instruction specifications: If all instructions in a method body $\mathtt{p}$ are well specified (i.e., $\{E_l\}\, l : I_l$ holds for all $l \in \Lambda_{\mathtt{p}}$) then the postcondition of $\mathtt{p}$ is the precondition of the `end_method` instruction and the precondition of $\mathtt{p}$ is the precondition of the first instruction (that is where execution of a method body starts). The following two definitions formalize this idea.

Definition 8. A specified $\mathrm{VM_K}$ instruction consists of

1. a labeled $\mathrm{VM_K}$ instruction $l : I_l$
2. a precondition E_l

We write this specified instruction as $\{E_l\}\, l : I_l$. We can only prove or deduce the validity of the instruction specification $\{E_l\}\, l : I_l$ in the context of a method implementation $\mathtt{p}$.

Definition 9. A specified $\mathrm{VM_K}$ method implementation $\mathtt{p}$ is a $\mathrm{VM_K}$ method implementation where all instructions have a single precondition, i.e., $|\mathtt{p}|$ is the number of instructions in the method (excluding `end_method`), the labels of the instructions are in $\Lambda_{\mathfrak{p}} = \{0..|\mathfrak{p}|\}$, there is exactly one instruction for each label $\mathfrak{p}(l) = I_l$, and, what is new, there is exactly one precondition for every label:

$$\text{precondition}_{\mathfrak{p}}(l) = E_l$$

We abbreviate

$$\text{spec}_{\mathfrak{p}}(l) = \{E_l\}\, l : I_l$$

Definition 10. A specified method implementation $\mathfrak{p}$ can be verified by verifying all of its components. The precondition for $\mathfrak{p}$ is the precondition of the first instruction, the postcondition of $\mathfrak{p}$ is the precondition of the `end_method` instruction.

$$\text{body}\,\frac{\begin{array}{r}\text{precondition}_{\mathfrak{p}}(0)[\mathit{undef}\,/v \text{ for all method variables } v] = P\\ \text{precondition}_{\mathfrak{p}}(|\mathfrak{p}|) = Q\\ {}[\forall i \in \Lambda_{\mathfrak{p}} : \text{spec}_{\mathfrak{p}}(i)]\end{array}}{\{P\}\ \mathfrak{p}\ \{Q\}}$$

Needless to say, the method body must be well formed: $\mathfrak{p}(|\mathfrak{p}|) =$ `end_method` and $\forall i < |\mathfrak{p}| : \mathfrak{p}(i) \neq$ `end_method`. We have to replace all method variables[8] by *undef* in the precondition for formal reasons.[9] It is easy to see however that this replacement does not change the value of the precondition: all local variables are *undef* at the beginning of a method. We do not allow references to any local data (i.e., stack elements, local variables, parameters) in the method postcondition.

Definition 11. Substitutions $E[e'/x]$ or $E[e'/s(i)]$, the simultaneous substitutions $E[e'_1/z_1, e'_2/z_2, \ldots]$ and the evaluation $[\![.]\!]$ of assertions E_l in a configuration $\langle S, \sigma, l\rangle$ are defined as usual. Assertions may not depend on the program counter l, so we can omit it:

$$[\![E_l]\!] : \mathit{State} \times \mathit{Stack} \to \mathit{Value}$$

The formulas that can be used as assertions are not restricted in any significant way. One obvious possibility would be to use sorted first-order formulas.

Definition 12. The current stack is referred to as s, and its elements are denoted by non-negative integers: element 0 is the top element, etc.:

$$\begin{aligned}[\![s(0)]\!]\langle S, (\sigma, v)\rangle &= v\\ [\![s(i+1)]\!]\langle S, (\sigma, v)\rangle &= [\![s(i)]\!]\langle S, \sigma\rangle\end{aligned}$$

Definition 13. Helper functions

$$\begin{aligned}\text{shift}(E) &= E[s(i+1)/s(i) \text{ for all } i \in \mathbb{N}]\\ \text{unshift} &= \text{shift}^{-1}\end{aligned}$$

With these definitions at hand, we now give a system of rules that allow to prove preconditions *for individual instructions*. The required enclosing specified method body ($\mathfrak{p}$) for the instructions is left implicit in the rules.

[8] i.e., local variables and stack elements

[9] the soundness proof for method invocations in section 4.1 on page 31 is simpler when the precondition of a method body does not depend on the value of local variables.

3.2.1 Instructions for Expressions

pushc

$$\text{pushc}\frac{E_l \rightarrow \text{unshift}(E_{l+1}[v/s(0)])}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{pushc } v}$$

Example 11. The top of the stack $s(0)$ must be 3 after `pushc` 3:

```
0: {true}
   pushc 3
1: {s(0) = 3}
   end_method
```

To see whether instruction 0 allows this specification, let's instantiate the formula in the antecedent of the rule:

$$\begin{aligned} & \mathit{true} \rightarrow \text{unshift}((s(0) = 3)[3/s(0)]) \\ \Longleftrightarrow\ & \mathit{true} \rightarrow \text{unshift}(3 = 3) \\ \Longleftrightarrow\ & \mathit{true} \rightarrow \mathit{true} \end{aligned}$$

pushv

$$\text{pushv}\frac{E_l \rightarrow \text{unshift}(E_{l+1}[x/s(0)])}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{pushv } x}$$

pop

$$\text{pop}\frac{E_l \rightarrow (\text{shift}(E_{l+1}))[s(0)/x]}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{pop } x}$$

Example 12. The value of variable x must be equal to the value of the top of the stack after a pop:

```
0: {s(0) = s_0}
   pop x
1: {x = s_0}
   end_method
```

To see whether instruction 0 allows this specification, we instantiate the formula in the antecedent of the rule:

$$\begin{aligned} & (s(0) = s_0) \rightarrow (\text{shift}(x = s_0))[s(0)/x] \\ \Longleftrightarrow\ & (s(0) = s_0) \rightarrow (x = s_0)[s(0)/x] \\ \Longleftrightarrow\ & (s(0) = s_0) \rightarrow (s(0) = s_0) \end{aligned}$$

Again we see that the rule allows such a specification.

unop

$$\text{unop}\frac{E_l \rightarrow E_{l+1}[(\text{op}\, s(0))/s(0)]}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{unop}_{\text{op}}}$$

binop

$$\text{binop}\frac{E_l \to (\text{shift}(E_{l+1}))[(s(1)\,\text{op}\,s(0))/s(1)]}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{binop}_{\text{op}}}$$

Example 13. We calculate 3/4 programatically and want to ensure that that value is actually the topmost element after the program:

```
0: {true}
   pushc 3
1: {s(0) = 3}
   pushc 4
2: {s(1) = 3 ∧ s(0) = 4}
   binop "/"
3: {s(0) = 3/4}
   end_method
```

We will now check only the instruction specifications for `binop`.

$$\begin{aligned} & (s(1) = 3 \wedge s(0) = 4) \to (\text{shift}(s(0) = 3/4))[(s(1)/s(0))/s(1)] \\ \iff & (s(1) = 3 \wedge s(0) = 4) \to (s(1) = 3/4)[(s(1)/s(0))/s(1)] \\ \iff & (s(1) = 3 \wedge s(0) = 4) \to ((s(1)/s(0)) = 3/4) \end{aligned}$$

The proof obligations for the other instruction specifications are:

0: $\mathit{true} \to \text{unshift}((s(0) = 3)[3/s(0)])$

1: $(s(0) = 3) \to \text{unshift}((s(1) = 3 \wedge s(0) = 4)[4/s(0)])$

op

$$\text{op}\frac{\begin{array}{c}\text{op} : \alpha_1 \times \cdots \times \alpha_n \to \beta_1 \times \cdots \times \beta_m \\ Z \text{ is a fresh logical variable (or a vector of variables if } m \neq 1) \\ E_l \to (\text{shift}^{n-m}(E_{l+1}[Z/s(0..m-1)]))[\text{op}(s(n-1..0))/Z]\end{array}}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{op}_{\text{op}}}$$

Example 14. `dup` is an abbreviation for $\texttt{op}_{x \mapsto (x,x)}$. To see what a specialized rule would look like, we replace op by $x \mapsto (x, x)$ and we get $(n = 1, m = 2)$:

$$\text{dup}\frac{E_l \to (\text{unshift}(E_{l+1}[Z/s(0..1)]))[(s(0), s(0))/Z]}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{dup}}$$

which is the same as

$$\text{dup}\frac{E_l \to (\text{unshift}(E_{l+1}[Z/s(0), Z/s(1)]))[s(0)/Z]}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{dup}}$$

As a sanity check, lets try to verify

```
0:  {s(1) = 3 ∧ s(0) = 4}
    dup
1:  {s(2) = 3 ∧ s(1) = 4 ∧ s(0) = 4}
    end_method
```

We have to check that

$$
\begin{aligned}
& (s(1) = 3 \wedge s(0) = 4) \\
& \rightarrow (\text{unshift}((s(2) = 3 \wedge s(1) = 4 \wedge s(0) = 4)[Z/s(0), Z/s(1)]))[s(0)/Z] \\
\Longleftrightarrow\ & (s(1) = 3 \wedge s(0) = 4) \rightarrow (\text{unshift}(s(2) = 3 \wedge Z = 4 \wedge Z = 4))[s(0)/Z] \\
\Longleftrightarrow\ & (s(1) = 3 \wedge s(0) = 4) \rightarrow (s(1) = 3 \wedge Z = 4 \wedge Z = 4)[s(0)/Z] \\
\Longleftrightarrow\ & (s(1) = 3 \wedge s(0) = 4) \rightarrow (s(1) = 3 \wedge s(0) = 4 \wedge s(0) = 4)
\end{aligned}
$$

Example 15. $\texttt{binop}_{\text{op}}$ is an abbreviation for $\texttt{op}_{(x,y)\mapsto \text{op}(x,y)}$. Let's derive the premise of the binop rule using the rule for $\texttt{op}_{\text{op}}$ $(n = 2, m = 1)$:

$$
\begin{aligned}
& E_l \rightarrow (\text{shift}(E_{l+1}[Z/s(0)]))[\text{op}(s(1..0))/Z] \\
\Longleftrightarrow\ & E_l \rightarrow (\text{shift}(E_{l+1}[Z/s(0)]))[(s(1)\,\text{op}\,s(0))/Z] \\
\Longleftrightarrow\ & E_l \rightarrow \text{shift}(E_{l+1})[(s(1)\,\text{op}\,s(0))/s(1)]
\end{aligned}
$$

3.2.2 Instructions that Modify the Control Flow

goto

$$
\text{goto}\,\frac{E_l \rightarrow E_{l'}}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{goto}\ l'}
$$

brtrue

$$
\text{brtrue}\,\frac{E_l \rightarrow (\neg s(0) \rightarrow \text{shift}(E_{l+1})) \wedge (s(0) \rightarrow \text{shift}(E_{l'}))}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{brtrue}\ l'}
$$

A more intuitive premise for `brtrue` inspired by the if-statement rule

$$
\text{if}\,\frac{\begin{array}{c}\{e \wedge P\}\ C_1\ \{Q\} \\ \{\neg e \wedge P\}\ C_2\ \{Q\}\end{array}}{\{P\}\ \texttt{if}(e)\{C_1\}\texttt{else}\{C_2\}\ \{Q\}}
$$

would probably be

$$
(E_l \wedge \neg s(0) \rightarrow \text{shift}(E_{l+1})) \wedge (E_l \wedge s(0) \rightarrow \text{shift}(E_{l'}))
$$

It is equivalent to our antecedent:

$$
\begin{aligned}
& (E_l \wedge \neg s(0) \rightarrow \text{shift}(E_{l+1})) \wedge (E_l \wedge s(0) \rightarrow \text{shift}(E_{l'})) \\
\Longleftrightarrow\ & (\neg(E_l \wedge \neg s(0)) \vee \text{shift}(E_{l+1})) \wedge (\neg(E_l \wedge s(0)) \vee \text{shift}(E_{l'})) \\
\Longleftrightarrow\ & (\neg E_l \vee s(0) \vee \text{shift}(E_{l+1})) \wedge (\neg E_l \vee \neg s(0) \vee \text{shift}(E_{l'})) \\
\Longleftrightarrow\ & \neg E_l \vee (s(0) \vee \text{shift}(E_{l+1}) \wedge (\neg s(0) \vee \text{shift}(E_{l'}))) \\
\Longleftrightarrow\ & E_l \rightarrow (\neg s(0) \rightarrow \text{shift}(E_{l+1})) \wedge (s(0) \rightarrow \text{shift}(E_{l'}))
\end{aligned}
$$

3.2.3 Instructions for Objects

checkcast

$$\text{checkcast}\frac{E_l \rightarrow E_{l+1} \wedge \tau(s(0)) \preceq T}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{checkcast}\, T}$$

The condition $\tau(s(0)) \preceq T$ guarantees that verified programs do not fail due to invalid casts – our operational semantics gets stuck if $\tau(s(0)) \preceq T$ does not hold. But this guarantee is a source of incompleteness. E.g., we cannot prove anything about the following method (T and S are unrelated)

```
0: newobj T
1: checkcast S
2: end_method
```

For this method however, any specification will do because there is *no terminating transition* $\langle S_0, \sigma_0, 0\rangle \rightarrow^* \langle S, \sigma, 2\rangle$. The only clean way out of this dilemma is to introduce exceptions – section 6 on page 49. The absence of stuck configurations[10] comes at the cost of introducing incompleteness. This is also true for `getfield` and `putfield`.

newobj

$$\text{newobj}\frac{E_l \rightarrow \text{unshift}(E_{l+1}[\text{new}(\$, T)/s(0), \$\langle T\rangle/\,\$])}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{newobj}\ T}$$

getfield

$$\text{getfield}\frac{E_l \rightarrow E_{l+1}[\$(\text{iv}(s(0), T@a))/s(0)] \wedge s(0) \neq \text{ null}}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{getfield}\ T@a}$$

putfield

$$\text{putfield}\frac{E_l \rightarrow (\text{shift}^2(E_{l+1}))[\$\langle \text{iv}(s(1), T@a) := s(0)\rangle/\,\$] \wedge s(1) \neq \text{ null}}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{putfield}\ T@a}$$

invokevirtual

$$\text{invokevirtual}\frac{\begin{array}{c}\mathcal{A} \vdash \{P\}\ T : m\ \{Q\} \\ Z \text{ is a vector of logical variables} \\ w \text{ is a vector of local or a stack elements } \neq s(0) \\ E_l \rightarrow s(1) \neq \text{ null} \wedge P[s(1)/\,\text{this}, s(0)/\,\text{p}][\text{shift}(w)/Z] \\ Q[s(0)/\,\text{result}][w/Z] \rightarrow E_{l+1}\end{array}}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{invokevirtual}\ T : m}$$

[10] i.e., a stronger condition than soundness

The simple invokevirtual rule used for methods with one explicit parameter (p) and a return value is the most important one for we are only going to reason about that possibility of invoking a method. The rule captures the fact that locals and stack elements are not modified by the invocation of a method. Other invocation rules would be very similar. We may summarize therefore summarize all the invocation rules. Let M be the method identifier. n is the number of arguments of M (without a possible this parameter), $m \in \mathbb{N}$ is the number of logical variables to be replaced by locals or stack elements because they aren't changed. $Z = (Z_i)_{i \in \{1..m\}}$, Z_i is a logical variable, $w = (w_i)_{i \in \{1..m\}}$, w_i is a stack element. $static(M)$ indicates whether M is a static method. $retval(M)$ indicates whether M returns a value. IND is the indicator function:

$$\mathrm{IND}(b) = \begin{cases} 1 & \text{if } b \\ 0 & \text{if } \neg b \end{cases}$$

invocation

$$\text{invocation}\;\frac{\begin{array}{c} \mathcal{A} \vdash \{P\}\ M\ \{Q\} \\ retval(M) \rightarrow \forall i : w_i \neq s(0) \\ \sigma = \begin{cases} \{\} & \text{if } static(M) \\ \{s(n)/\,\text{this}\} & \text{otherwise} \end{cases} \\ \delta = \begin{cases} \{\} & \text{if } \neg retval(M) \\ \{s(0)/\,\text{result}\} & \text{otherwise} \end{cases} \\ j = n - \mathrm{IND}(static(M)) + \mathrm{IND}(\neg retval(M)) \\ G = s(n) \neq \text{null} \vee static(M) \\ E_l \rightarrow G \wedge P\sigma[s(n-1..0)/\,\mathrm{p}_{1..n}][\text{shift}^j(w)/Z] \\ Q\delta[w/Z] \rightarrow E_{l+1} \end{array}}{\mathcal{A} \vdash \{E_l\}\, l : \text{invocation-instr}M}$$

Here is the explanation:

- $retval(M) \rightarrow \forall i : w_i \neq s(0)$: if there is a return value, w may not contain $s(0)$ because $s(0)$ is not preserved, it contains the return-value after the execution of the method
- σ: The this parameter need not be passed when the method is static. Similarly, the return value need not be passed back using substitution δ if the method does not have a return value.
- j is the number of elements the stack contains more before the invocation than afterwards. Basically, $j = n$: The stack contains "this" and n arguments before and the return value after the invocation. If the method is static, "this" is missing ($-\mathrm{IND}(static(M))$) but if the method does not return a value, the stack will contain one element more before the execution of M ($+\mathrm{IND}(\neg retval(M))$)
- $s(n) \neq$ null $\vee\, static(M)$: We need only check the this parameter if M is not static.

- invocation-instr is `call` if M is statically bound (either a static or non-virtual method, i.e., $M = T@w$ or `invokevirtual` if M is dynamically bound, i.e., $M = T : v$.

Example 16. Our method invocation rules formalize two things: Capture the effect of the execution of a method and save locals across a method invocation instruction. Separating these orthogonal concerns as in [PHM99] would lead to simpler rules (here for the `invokevirtual` instruction with one argument and a return value):

$$\text{invokevirtual}\,\frac{\begin{array}{c}\mathcal{A} \vdash \{P\}\ T : m\ \{Q\} \\ E_l \to s(1) \neq \text{null} \land P[s(1)/\,\text{this}, s(0)/\,\text{p}] \\ Q[s(0)/\,\text{result}] \to E_{l+1}\end{array}}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{invokevirtual}\ T : m}$$

$$\text{invokevar}\,\frac{\begin{array}{c}Z \text{ is a logical variable} \\ w \text{ is a local variable or a stack element} \neq s(0) \\ \mathcal{A} \vdash \{E'_l\}\ l : \texttt{invokevirtual}\ T : m\ \left\{E'_{l+1}\right\} \\ E_l \to E'_l[(\text{shift}(w))/Z] \qquad E'_{l+1}[w/Z] \to E_{l+1}\end{array}}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{invokevirtual}\ T : m}$$

The problem here is that we now have an arbitrary number of preconditions for one instructions (one additional E'_l for every invokevar). We have only associated one specification per instruction to every method body proof. What is more, we would have to define what is needed to prove the triple

$$\{E'_l\}\ l : \texttt{invokevirtual}\ T : m\ \left\{E'_{l+1}\right\}$$

in invokevar.

3.2.4 The special operation `nop`

$$\text{nop}\,\frac{E_l \to E_{l+1}}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{nop}}$$

Observation 4. There is exactly one transition for every instruction in our operational semantics and exactly one rule in our programming logic. It is no surprise that they look similar.

This one-to-one correspondence can help us prove the soundness of the logic: we just need to show that if there is a small step transition from one state $K = \langle S, \sigma, l\rangle$ to its next state $K' = \langle S', \sigma', l'\rangle$[11] *and* the precondition E_l holds in K *then* $E_{l'}$ must also hold in K'.

We may have reached the end of the method if there is no transition. I.e. we may have reached the `end_method` instruction. In that case, we know the precondition of `end_method` holds. This is also the postcondition of our method.

[11] which means: $\mathfrak{p}; \langle S, \sigma, l\rangle \to \langle S', \sigma', l'\rangle$

Because the `end_method` instruction is the only means of returning from a method[12], we know that if a method terminates, its postcondition will always hold.

Example 17. Calculating x^n recursively based on the observation that $x^n = (x \cdot x)^{n/2}$.

$$\{P \equiv x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}\ \texttt{Rec@pow}\ \{Q \equiv \text{result} = x_0^{n_0}\}$$

An assumption *during* verification is $\{P\}$ `Rec : pow` $\{Q\}$. Again see [PHM99] for exactly when such an assumption can be deduced from the assumption $\{P\}$ `Rec@pow` $\{Q\}$. The reasoning is simple in our case because `Rec` is the only class in our program.

Our assumption

$$\{P \equiv x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}\ \texttt{Rec : pow}\ \{Q \equiv \text{result} = x_0^{n_0}\}$$

cannot be used directly for the recursive method invocations. We have to adapt it and deduce two other triples that are more useful in our calling contexts with the help of the rules for method specifications in section 3.1.1 on page 17. We use a common linear notation here that makes it easier to grasp the idea of the proof ("proof outline"). The ▼ and ▲ symbols are used to indicate nesting:

▼$\{P\}$ `(rule)`
▼$\{P'\}$
`comp`
▲$\{Q'\}$
▲$\{Q\}$ `(rule)`

stands for

$$\text{rule}\frac{\{P'\}\ \text{comp}\ \{Q'\}}{\{P\}\ \text{comp}\ \{Q\}}$$

Note that in a fully formal presentation, the proof would consist of a list of derivation trees for individual instructions. We could then integrate the modifications we make to our original assumption into a derivation tree of an instruction (the `invokevirtual` instruction).

- The adaption of the method specification for the case $n \mod 2 = 0$.

  ▼$\{x > 0 \wedge n \geq 0 \wedge x = x_0 \cdot x_0 \wedge n = n_0/2 \wedge n_0 \mod 2 = 0\}$ `(inv-rule)`
  ▼$\{x > 0 \wedge n \geq 0 \wedge x = x_0 \cdot x_0 \wedge n = n_0/2\}$ `(subst-rule)`
  ▼$\{x > 0 \wedge n \geq 0 \wedge x = x'_0 \wedge n = n'_0\}$ `(subst-rule)`
  ▼$\{x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
  `Rec : pow`
  ▲$\{\text{result} = x_0^{n_0}\}$
  ▲$\left\{\text{result} = {x'_0}^{n'_0}\right\}$ `(subst-rule)`

[12] see the definition of the invokevirtual rule

$\blacktriangle\left\{\text{result} = (x_0 \cdot x_0)^{(n_0/2)}\right\}$ (subst-rule)
$\blacktriangle\left\{\text{result} = (x_0 \cdot x_0)^{(n_0/2)} \wedge n_0 \mod 2 = 0\right\}$ (inv-rule)

- The adaption of the method specification for the case $n \mod 2 \neq 0$. The inv rule is used to introduce auxiliary logical variables that can then be used to save x across the `invokevirtual` call.

$\blacktriangledown\{x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0 - 1 \wedge x_f = x_0\}$ (inv-rule)
$\blacktriangledown\{x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0 - 1\}$ (subst-rule)
$\blacktriangledown\{x > 0 \wedge n \geq 0 \wedge x = x_0' \wedge n = n_0'\}$ (subst-rule)
$\blacktriangledown\{x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
Rec:pow
$\blacktriangle\{\text{result} = x_0^{n_0}\}$
$\blacktriangle\left\{\text{result} = {x_0'}^{n_0'}\right\}$ (subst-rule)
$\blacktriangle\left\{\text{result} = {x_0}^{n_0-1}\right\}$ (subst-rule)
$\blacktriangle\left\{\text{result} = {x_0}^{n_0-1} \wedge x_f = x_0\right\}$ (inv-rule)
$\rightarrow \{\text{result} \cdot x_f = {x_0}^{n_0} \wedge x_f = x_0\}$

```
 0: {x > 0 ∧ n ≥ 0 ∧ x = x0 ∧ n = n0}
    pushv n
 1: {x > 0 ∧ n ≥ 0 ∧ x = x0 ∧ n = n0 ∧ (s(0) ≠ 0) = (n ≠ 0)}
    unop <>0 // is n ≠ 0
 2: {x > 0 ∧ n ≥ 0 ∧ x = x0 ∧ n = n0 ∧ s(0) = (n ≠ 0)}
    brtrue 6 // if n ≠ 0, perform actual algorithm

// otherwise, n = 0, just return 1
 3: {n = 0 ∧ x > 0 ∧ n ≥ 0 ∧ x = x0 ∧ n = n0}
    pushc 1
 4: {s(0) = x0^n0}
    pop result // store result 1 in special variable
 5: {result = x0^n0}
    goto 30 // goto end

// actual algorithm
 6: {n ≠ 0 ∧ x > 0 ∧ n ≥ 0 ∧ x = x0 ∧ n = n0}
    pushv n
 7: { ((s(0) mod 2) ≠ 0) = (n mod 2 ≠ 0)
      ∧ n ≠ 0 ∧ x > 0 ∧ n ≥ 0 ∧ x = x0 ∧ n = n0 }

    pushc 2
 8: { ((s(1) mod s(0)) ≠ 0) = (n mod 2 ≠ 0)
      ∧ n ≠ 0 ∧ x > 0 ∧ n ≥ 0 ∧ x = x0 ∧ n = n0 }

    binop "%"
 9: {(s(0) ≠ 0) = (n mod 2 ≠ 0) ∧ n ≠ 0 ∧ x > 0 ∧ n ≥ 0 ∧ x = x0 ∧ n = n0}
    unop <>0 // n mod 2 ≠ 0?
10: {s(0) = (n mod 2 ≠ 0) ∧ n ≠ 0 ∧ x > 0 ∧ n ≥ 0 ∧ x = x0 ∧ n = n0}
    brtrue 21
```

`// otherwise,` $n \mod 2 = 0$`, so we can apply` $x^n = (x{\cdot}x)^{n/2}$

11: $\{n \mod 2 = 0 \wedge n \neq 0 \wedge x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
`pushv this`
12: $\{n \mod 2 = 0 \wedge n \neq 0 \wedge x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
`pushv x`
13: $\{s(0) = x \wedge n \mod 2 = 0 \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
`pushv x`
14: $\{s(1){\cdot}s(0) = x{\cdot}x \wedge \ \wedge n \mod 2 = 0 \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
`binop "*"`
15: $\{s(0) = x{\cdot}x \wedge n \mod 2 = 0 \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
`pushv n`
16: $\left\{\begin{array}{l} \quad (s(0)/2) = n/2 \wedge s(1) = x{\cdot}x \wedge n \mod 2 = 0 \\ \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0 \end{array}\right\}$

`pushc 2`
17: $\left\{\begin{array}{l} \quad (s(1)/s(0)) = n/2 \wedge s(2) = x{\cdot}x \wedge n \mod 2 = 0 \\ \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0 \end{array}\right\}$

`binop "/"`
18: $\left\{\begin{array}{l} \quad (x > 0 \wedge n \geq 0 \wedge x = x_0{\cdot}x_0 \wedge n = n_0/2 \\ \wedge n_0 \mod 2 = 0)[s(2)/\,\mathrm{this}, s(1)/x, s(0)/n] \end{array}\right\}$

`invokevirtual Rec:pow`
19: $\{s(0) = x_0^{n_0}\}$
`pop result`
20: $\{\mathrm{result} = x_0^{n_0}\}$
`goto 30`

`// if` $n \mod 2 \neq 0$`, subtract one`

21: $\{n \mod 2 \neq 0 \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
`pushv x`
22: $\{s(0) = x \wedge x > 0 \wedge n > 0 \wedge x = x_0 \wedge n = n_0\}$
`pushv this`
23: $\{x = x \wedge s(1) = x \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
`pushv x`
24: $\{n - 1 = n - 1 \wedge s(0) = x \wedge s(2) = x \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
`pushv n`
25: $\{(s(0) - 1) = n - 1 \wedge s(1) = x \wedge s(3) = x \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
`pushc 1`
26: $\left\{\begin{array}{l} \quad (s(1) - s(0)) = n - 1 \wedge s(2) = x \wedge s(4) = x \\ \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0 \end{array}\right\}$

`binop "-"`
27: $\{s(1) > 0 \wedge s(0) \geq 0 \wedge s(1) = x_0 \wedge s(0) = n_0 - 1 \wedge s(3) = x_0\}$
`invokevirtual Rec:pow`
28: $\{s(1){\cdot}s(0) = x_0^{n_0}\}$
`binop "*"`
29: $\{s(0) = x_0^{n_0}\}$
`pop result`

30: $\{\mathrm{result} = x_0^{n_0}\}$

```
end_method
```

This example should give a rough idea how the programming logic can be used in practice to prove method implementations correct. Although the instructions specification are quite long, they can still be understood intuitively: they express the programmer's assertions about the state at every program point.

3.3 The Special Shape of the Rules

There is exactly one rule for every instruction. All rules for verifying instruction specifications $\{E_l\}\, l : I_l$, have a premise of the following form:

$$E_l \to \mathrm{wp}^1_{\mathtt{p}}(I_l, (E_i)_{i \in succ(l:I_l)})$$

where $succ(l : I_l)$ is the successor function returning the set of possible successor labels for an instruction $l : I_l$.

$$succ(l : I_l) = \begin{cases} (l') & \text{if } I_l = \mathtt{goto}\ l' \\ (l+1, l') & \text{if } I_l = \mathtt{brtrue}\ l' \\ (l+1) & \text{otherwise} \end{cases}$$

I_l	$\mathrm{wp}^1_{\mathtt{p}}(I_l, (E_i)_{i \in succ(l:I_l)})$
$\mathtt{pushc}\ v$	$\mathrm{unshift}(E_{l+1}[v/s(0)])$
$\mathtt{pushv}\ x$	$\mathrm{unshift}(E_{l+1}[x/s(0)])$
$\mathtt{pop}\ x$	$(\mathrm{shift}(E_{l+1}))[s(0)/x]$
$\mathtt{binop}_{\mathrm{op}}$	$(\mathrm{shift}(E_{l+1}))[(s(1)\,\mathrm{op}\,s(0))/s(1)]$
$\mathtt{goto}\ l'$	$E_{l'}$
$\mathtt{brtrue}\ l'$	$(\neg s(0) \to \mathrm{shift}(E_{l+1})) \wedge (s(0) \to \mathrm{shift}(E_{l'}))$
$\mathtt{checkcast}\, T$	$E_{l+1} \wedge \tau(s(0)) \preceq T$
$\mathtt{newobj}\ T$	$\mathrm{unshift}(E_{l+1}[\mathrm{new}(\$, T)/s(0), \$\langle T\rangle/\,\$])$
$\mathtt{getfield}\ T@a$	$E_{l+1}[\$(\mathrm{iv}(s(0), T@a))/s(0)] \wedge s(0) \neq \mathrm{null}$
$\mathtt{putfield}\ T@a$	$(\mathrm{shift}^2(E_{l+1}))[\$\langle \mathrm{iv}(s(1), T@a) := s(0)\rangle/\,\$] \wedge s(1) \neq \mathrm{null}$

Figure 3: The values of the $\mathrm{wp}^1_{\mathtt{p}}$ function. The value of $\mathrm{wp}^1_{\mathtt{p}}$ for invocation instructions is discussed in section 4.2.1 on page 45.

Lemma 1. $\mathrm{wp}^1_{\mathtt{p}}$ is distributive with respect to the logical conjunction $\wedge$ and disjunction $\vee$ in the successor assertions:

$$\mathrm{wp}^1_{\mathtt{p}}(I, (F_i^{(1)} \odot F_i^{(2)})_i) = \mathrm{wp}^1_{\mathtt{p}}(I, (F_i^{(1)})_i)) \odot \mathrm{wp}^1_{\mathtt{p}}(I, (F_i^{(1)})_i)$$

It is easy to check for every instruction separately. The equivalence transformations are based on the fact that shift and substitution for $A\,\mathrm{op}\,B$ are defined as applications on the constituents A and B. $\mathtt{brtrue}\ l'$ is the only interesting case because it has two successors. The following list is just for the sake of completeness.

- `brtrue` l':

$$\begin{aligned}
& \mathrm{wp}^1_{\mathtt{p}}(\mathtt{brtrue}\ l', F_1^{(1)} \odot F_1^{(2)}, F_2^{(1)} \odot F_2^{(2)}) \\
\iff & (\neg s(0) \rightarrow \mathrm{shift}(F_1^{(1)} \odot F_1^{(2)})) \wedge (s(0) \rightarrow \mathrm{shift}(F_2^{(1)} \odot F_2^{(2)})) \\
& \text{by case distinction on } s(0) \\
\iff & ((\neg s(0) \rightarrow \mathrm{shift}(F_1^{(1)})) \wedge (s(0) \rightarrow \mathrm{shift}(F_2^{(1)}))) \odot \\
& ((\neg s(0) \rightarrow \mathrm{shift}(F_1^{(2)})) \wedge (s(0) \rightarrow \mathrm{shift}(F_2^{(2)}))) \\
\iff & \mathrm{wp}^1_{\mathtt{p}}(\mathtt{brtrue}\ l', F_1^{(1)}, F_2^{(1)}) \odot \mathrm{wp}^1_{\mathtt{p}}(\mathtt{brtrue}\ l', F_1^{(2)}, F_2^{(2)})
\end{aligned}$$

- `pushc` v:

$$\begin{aligned}
& \mathrm{unshift}((F^{(1)} \odot F^{(2)})[v/s(0)]) \\
\iff & \mathrm{unshift}(F^{(1)}[v/s(0)]) \odot \mathrm{unshift}(F^{(2)}[v/s(0)])
\end{aligned}$$

- `pushv` x: same as for `pushc` c
- `pop` x:

$$\begin{aligned}
& (\mathrm{shift}(F^{(1)} \odot F^{(2)}))[s(0)/x] \\
\iff & (\mathrm{shift}(F^{(1)}) \odot \mathrm{shift}(F^{(2)}))[s(0)/x] \\
\iff & \mathrm{shift}(F^{(1)})[s(0)/x] \odot \mathrm{shift}(F^{(2)})[s(0)/x]
\end{aligned}$$

- $\mathtt{binop}_{\mathrm{op}}$:

$$\begin{aligned}
& (\mathrm{shift}(F^{(1)} \odot F^{(2)}))[(s(1)\,\mathrm{op}\,s(0))/s(1)] \\
\iff & (\mathrm{shift}(F^{(1)}) \odot \mathrm{shift}(F^{(2)}))\underbrace{[(s(1)\,\mathrm{op}\,s(0))/s(1)]}_{R} \\
\iff & \mathrm{shift}(F^{(1)})R \odot \mathrm{shift}(F^{(2)})R
\end{aligned}$$

- `goto` l':

$$\begin{aligned}
& (F^{(1)} \odot F^{(2)}) \\
\iff & (F^{(1)}) \odot (F^{(2)})
\end{aligned}$$

- `checkcast` T:

$$\begin{aligned}
& (F^{(1)} \odot F^{(2)}) \wedge \tau(s(0)) \preceq T \\
\iff & F^{(1)} \wedge \tau(s(0)) \preceq T \odot F^{(2)} \wedge \tau(s(0)) \preceq T
\end{aligned}$$

- `newobj` T:

$$\begin{aligned}
& \mathrm{unshift}((F^{(1)} \odot F^{(2)})\underbrace{[\mathrm{new}(\$, T)/s(0), \$\langle T\rangle/\$]}_{R}) \\
\iff & \mathrm{unshift}(F^{(1)}R) \odot \mathrm{unshift}(F^{(2)}R)
\end{aligned}$$

- `getfield` $T@a$:

$$
\begin{aligned}
& (F^{(1)} \odot F^{(2)}) \underbrace{[\$(\mathrm{iv}(s(0), T@a))/s(0)]}_{R} \wedge s(0) \neq \text{ null} \\
\Longleftrightarrow\ & (F^{(1)}R \odot F^{(2)}R) \wedge s(0) \neq \text{ null} \\
\Longleftrightarrow\ & F^{(1)}R \wedge s(0) \neq \text{ null} \odot F^{(2)}R \wedge s(0) \neq \text{ null}
\end{aligned}
$$

- `putfield` $T@a$:

$$
\begin{aligned}
& (\mathrm{shift}^2(F^{(1)} \odot F^{(2)})) \underbrace{[\$\langle \mathrm{iv}(s(1), T@a) := s(0)\rangle / \$]}_{R} \wedge s(1) \neq \text{ null} \\
\Longleftrightarrow\ & (\mathrm{shift}^2(F^{(1)}) \odot \mathrm{shift}^2(F^{(2)}))R \wedge s(1) \neq \text{ null} \\
\Longleftrightarrow\ & (\mathrm{shift}^2(F^{(1)})R \odot \mathrm{shift}^2(F^{(2)})R) \wedge s(1) \neq \text{ null} \\
\Longleftrightarrow\ & \mathrm{shift}^2(F^{(1)})R \wedge s(1) \neq \text{ null} \odot \mathrm{shift}^2(F^{(2)})R \wedge s(1) \neq \text{ null}
\end{aligned}
$$

Lemma 2. $\mathrm{wp}^1_{\mathtt{p}}(I, (false)_i \in succ(l : I_l)) = false$

This is only a quick check.

4 Soundness, Completeness and Weakest Preconditions

In this section, we discuss soundness and completeness properties of the axiomatic semantics with respect to the operational semantics. We introduce a weakest precondition calculus that can be used to derive weakest preconditions for arbitrary method bodies. As mentioned in the introduction, the calculus can be used to derive additional rules for "compound instructions": instructions whose effect is defined as the sequential composition of primitive[13] instructions. Compound instructions may even contain loops.

4.1 Soundness

We prove the soundness of the $\mathrm{VM_K}$ bytecode logic as discussed section 3 on page 15. Operational semantics are more intuitive and suitable for automatic generation of interpreter prototypes that can be used to validate the definition while axiomatic semantics is a less intuitively accessible higher level abstraction intended for program verification. That's why soundness is proven with respect to the operational semantics.[14] The argument may not be that convincing for simple instruction because the operational semantics is very close to the axiomatic definition but soundness is certainly worth checking for method

[13] instructions of bounded complexity, i.e., all except method invocation instructions

[14] As noted before, this is not true for very complex instructions that do multiple rather unrelated things like the CLI `box` instruction. It is better to define them directly as translations to more basic instructions. section 5 on page 46.

invocations. Nonetheless, we'll prove soundness for the simple instructions as well.

The proof is based on an embedding of both operational and axiomatic semantics into higher order logic. Exactly the same method is used in [PHM99].[15] As we have based our axiomatic semantics on the ideas developed in [PHM99] and as we have taken care to make the operational semantics as directly comparable to the source semantics in [PHM99] as possible, we can even recycle parts of their proof – exactly those that deal with method specifications. Cf. [PHM99] for the motivation and more details about the proof.

What soundness theorem do we want to prove? There are two levels of abstractions: method specifications and instruction specifications. Instruction specifications cannot be defined without the notion of enclosing method specifications (or at least specifications for instructions in an enclosing method body). Our goal is the modular specification and verification of methods, so we will not prove soundness for instruction specifications explicitly. Our soundness is theorem then is

$$\vdash \{P\}\ M\ \{Q\} \Rightarrow\ \models \{P\}\ M\ \{Q\}$$

Definition 14.

$$sem(C, \mathfrak{p}, C') \equiv \mathfrak{p}; C \to^* C'$$

$$nsem1(N+1, C \equiv \langle S, \sigma, l\rangle, \mathfrak{p}, C' \equiv \langle S', \sigma', l'\rangle) \equiv$$
$$\begin{cases} \text{if } I_l \neq \texttt{invokevirtual } T : m \\ \quad \mathfrak{p}; C \to C' \\ \text{if } I_l = \texttt{invokevirtual } T : m \wedge C = \langle S, (\sigma_0, y, v), l\rangle \wedge N > 0 \\ \quad \mathfrak{p}' = body_{\mathrm{VM_K}}(impl(\tau(y), m)) \wedge \\ \quad \mathfrak{p}'(l') = \texttt{end\_method} \wedge \\ \quad nsem(N, \langle \{\text{this} \mapsto y, \text{p} \mapsto v, \$ \mapsto S(\$)\}, (), 0\rangle, \mathfrak{p}', \langle S', \sigma', l'\rangle) \\ false \text{ otherwise} \end{cases}$$

$nsem$ is the reflexive transitive closure of $nsem(N, ., \mathfrak{p}, .)$, i.e.

$$nsem(N, C, \mathfrak{p}, C)$$

and

$$\frac{nsem1(N, C, \mathfrak{p}, C') \qquad nsem(N, C', \mathfrak{p}, C'')}{nsem(N, C, \mathfrak{p}, C'')}$$

The relation between $nsem1$ and $nsem$ is thus the same as between $\to$ and $\to^*$. $nsem(N, S, C, S')$ means that S is a state that leads to a terminating execution with poststate S' with a recursion depth that is at most N.

[15] Including names of relations so that the two proofs are directly comparable.

Lemma 3.

$$sem(P, C, Q) \iff \exists N : nsem(N, P, C, Q)$$

Proof. "$\Leftarrow$" trivial

"$\Rightarrow$" by induction on N (we only consider $I_l = \texttt{invokevirtual}$):

$$\frac{\cdots \quad sem(\langle\{\text{this} \mapsto y, \text{p} \mapsto v, \$ \mapsto S(\$)\}, (), 0\rangle, \mathfrak{p}', \langle S', \sigma', l'\rangle)}{sem(C, \mathfrak{p}, C')}$$

$$sem(\langle\{\text{this} \mapsto y, \text{p} \mapsto v, \$ \mapsto S(\$)\}, (), 0\rangle, \mathfrak{p}', \langle S', \sigma', l'\rangle)$$

by the induction hypothesis

$$\Rightarrow nsem(N, \langle\{\text{this} \mapsto y, \text{p} \mapsto v, \$ \mapsto S(\$)\}, (), 0\rangle, \mathfrak{p}', \langle S', \sigma', l'\rangle)$$

From that we obtain:

$$\frac{\cdots \quad nsem(N, \langle\{\text{this} \mapsto y, \text{p} \mapsto v, \$ \mapsto S(\$)\}, (), 0\rangle, \mathfrak{p}', \langle S', \sigma', l'\rangle)}{nsem(N+1, C, \mathfrak{p}, C')}$$

□

Definition 15. $H(P, M, Q)$ formalizes the meaning of the specification $\{P\}\ \ M\ \ \{Q\}$.

$$\begin{aligned} H(P, \mathfrak{p}, Q) &\equiv \forall C \equiv \langle\{\text{this} \mapsto \text{this}_0, \text{p} \mapsto \text{p}_0, \$ \mapsto \$_0\}, (), 0\rangle, C' \equiv \langle S', \sigma', l'\rangle : \\ &\qquad sem(C, \mathfrak{p}, C') \wedge I_{l'} = \texttt{end\_method} \wedge [\![P]\!]C \Rightarrow [\![Q]\!]C' \\ H(P, T@m, Q) &\equiv H(\text{this} \neq \text{null} \wedge P, body(T@m), Q) \\ H(P, T : m, Q) &\equiv \forall T \preceq T_0 : H(\tau(\text{this}) = T \wedge P, impl(T, m), Q) \end{aligned}$$

Definition 16.

$$\begin{aligned} K(N, P, \mathfrak{p}, Q) &\equiv \forall C \equiv \langle\{\text{this} \mapsto \text{this}_0, \text{p} \mapsto \text{p}_0, \$ \mapsto \$_0\}, (), 0\rangle, \\ &\qquad C' \equiv \langle S', \sigma', l'\rangle : \\ &\qquad nsem(N, C, \mathfrak{p}, C') \wedge I_{l'} = \texttt{end\_method} \\ &\qquad \wedge [\![P]\!]C \Rightarrow [\![Q]\!]C' \\ K(0, P, T@m, Q) &\equiv true \\ K(N+1, P, T@m, Q) &\equiv K(N, \text{this} \neq \text{null} \wedge P, body_{\mathrm{VM_K}}(T@m), Q) \\ K(N, P, T_0 : m, Q) &\equiv \forall T \preceq T_0 : K(N, \tau(\text{this}) \preceq T \wedge P, impl(T, m), Q) \end{aligned}$$

Lemma 4.

$$H(P, C, Q) \iff \forall N : K(N, P, C, Q)$$

Proof.

$$\begin{aligned} & H(P, \mathfrak{p}, Q) \\ \iff & \forall C \equiv \langle\{\text{this} \mapsto \text{this}_0, \text{p} \mapsto \text{p}_0, \$ \mapsto \$_0\}, (), 0\rangle, C' \equiv \langle S', \sigma', l'\rangle : \\ & sem(C, \mathfrak{p}, C') \wedge I_{l'} = \texttt{end\_method} \wedge [\![P]\!]C \Rightarrow [\![Q]\!]C' \\ \iff & \forall C \equiv \langle\{\text{this} \mapsto \text{this}_0, \text{p} \mapsto \text{p}_0, \$ \mapsto \$_0\}, (), 0\rangle, C' \equiv \langle S', \sigma', l'\rangle : \\ & \exists N : nsem(N, C, \mathfrak{p}, C') \wedge I_{l'} = \texttt{end\_method} \wedge [\![P]\!]C \Rightarrow [\![Q]\!]C' \\ \iff & \forall N \forall C \equiv \langle\{\text{this} \mapsto \text{this}_0, \text{p} \mapsto \text{p}_0, \$ \mapsto \$_0\}, (), 0\rangle, C' \equiv \langle S', \sigma', l'\rangle : \\ & nsem(N, C, \mathfrak{p}, C') \wedge I_{l'} = \texttt{end\_method} \wedge [\![P]\!]C \Rightarrow [\![Q]\!]C' \\ \iff & \forall N : K(N, P, \mathfrak{p}, Q) \end{aligned}$$

$\square$

Using the translation of $\models \{A\}\ B\ \{C\}$ the soundness theorem reads

$$\vdash \{A\}\ B\ \{C\} \Rightarrow H(A, B, C)$$

The next few sections are devoted to actually proving soundness. We do that by showing $\vdash \{P\}\ C\ \{Q\} \Rightarrow \forall N : K(N, P, C, Q)$ by induction on the shape of the derivation tree of $\{P\}\ C\ \{Q\}$. We do not cover the method specification specific rules. They are already proven correct in [PHM99]. In fact, there is only one induction case left to prove: When the root of the derivation for $\vdash \{A\}\ B\ \{C\}$ is the body rule:

$$\text{body}\frac{\begin{array}{r}\text{precondition}_{\mathfrak{p}}(0)[\mathit{undef}\ /v \text{ for all method variables } v] = P\\ \text{precondition}_{\mathfrak{p}}(|\mathfrak{p}|) = Q\\ {[\forall i \in \Lambda_{\mathfrak{p}} : \text{spec}_{\mathfrak{p}}(i)]}\end{array}}{\{P\}\ \mathfrak{p}\ \{Q\}}$$

Then we have to show that:

$$\vdash \{P\}\ \mathfrak{p}\ \{Q\} \Rightarrow \forall N : K(N, P, \mathfrak{p}, Q)$$

We do this by induction over N. The case $N = 0$ is trivial, we may therefore assume for the rest of the proof that $N > 0$. Our only hope is expanding the right hand side. The end-result should therefore be:

$$\begin{array}{r}\forall C \equiv \langle\{\text{this} \mapsto \text{this}_0, \text{p} \mapsto \text{p}_0, \$ \mapsto \$_0\}, (), 0\rangle, C' \equiv \langle S', \sigma', l'\rangle :\\ \mathit{nsem}(N, C, \mathfrak{p}, C') \wedge I_{l'} = \texttt{end\_method} \wedge [\![P]\!]C \Rightarrow [\![Q]\!]C'\end{array}$$

In order to prove the body rule, we may assume its antecedents:

$$[\forall i \in \Lambda_{\mathfrak{p}} : \text{spec}_{\mathfrak{p}}(i)]$$

and

$$\begin{array}{r}I_{|\mathfrak{p}|} = \texttt{end\_method}\\ \forall i < |\mathfrak{p}| : \mathfrak{p}(i) \neq \texttt{end\_method}\\ \text{precondition}_{\mathfrak{p}}(0)[\mathit{undef}\ /v \text{ for all method variables } v] = P \wedge \text{this} \neq \text{null}\\ \text{postcondition}_{\mathfrak{p}}(|\mathfrak{p}|) = Q\end{array}$$

Instead of the complex formula

$$\begin{array}{r}\forall C \equiv \langle\{\text{this} \mapsto \text{this}_0, \text{p} \mapsto \text{p}_0, \$ \mapsto \$_0\}, (), 0\rangle, C' \equiv \langle S', \sigma', l'\rangle :\\ \mathit{nsem}(N, C, \mathfrak{p}, C') \wedge I_{l'} = \texttt{end\_method} \wedge [\![P]\!]C \Rightarrow [\![Q]\!]C'\end{array}$$

we prove the more general and more easily reusable and generalizable

$$\forall C \equiv \langle S, \sigma, l\rangle, C' \equiv \langle S', \sigma', l'\rangle \quad : \quad nsem(N, C, \mathfrak{p}, C') \wedge [\![E_l]\!]C \quad \Rightarrow \quad [\![E_{l'}]\!]C'$$

by induction on the shape of the derivation of $nsem(N, C, \mathfrak{p}, C')$.

When we have that, we can easily get the original formula by constraining C to $\langle \{\text{this} \mapsto \text{this}_0, \text{p} \mapsto \text{p}_0, \$ \mapsto \$_0\}, (), 0\rangle$. We would thus have shown the inductive step $(N-1) \rightarrow (N)$ and we can conclude:

$$\forall N : K(N, P, \mathfrak{p}, Q)$$

Now that we have presented the overall structure of the proof, we'll give the deduction of

$$ZZ(N) \equiv \forall C \equiv \langle S, \sigma, l\rangle, C' \equiv \langle S', \sigma', l'\rangle : nsem(N, C, \mathfrak{p}, C') \wedge [\![E_l]\!]C \Rightarrow [\![E_{l'}]\!]C'$$

We will need the induction hypothesis $M < N \Rightarrow ZZ(M)$.

4.1.1 Proof of Soundness for Sequences of Instructions

We'll prove the propositions

$$ZZ(N) \equiv \forall C \equiv \langle S, \sigma, l\rangle, C' \equiv \langle S', \sigma', l'\rangle : nsem(N, C, \mathfrak{p}, C') \wedge [\![E_l]\!]C \Rightarrow [\![E_{l'}]\!]C'$$

We do this by induction on the length m of the derivation of $nsem(N, C, \mathfrak{p}, C')$:

$(ZZ(m = 0))$: trivial: $l = l'' \Rightarrow (E_l = E_{l''} \wedge C = C')$.

$(ZZ(m) \rightarrow ZZ(m + 1))$: After proving the first step we get:

$$\forall C \equiv \langle S, \sigma, l\rangle, C' \equiv \langle S', \sigma', l'\rangle : nsem1(N, C, \mathfrak{p}, C') \wedge [\![E_l]\!]C \Rightarrow [\![E_{l'}]\!]C')$$

and with the induction hypothesis

$$\forall C' \equiv \langle S', \sigma', l'\rangle, C'' \equiv \langle S'', \sigma'', l''\rangle : nsem^m(N, C', \mathfrak{p}, C'') \wedge [\![E_{l'}]\!]C' \Rightarrow [\![E_{l''}]\!]C'')$$

We may prove the implication $ZZ(m+1) = nsem^{m+1}(N, C, \mathfrak{p}, C0') \wedge [\![E_l]\!]C \Rightarrow [\![E_{l''}]\!]$:

$$\begin{aligned}
&nsem^{m+1}(N, C, \mathfrak{p}, C'') \wedge [\![E_l]\!]C \\
\Rightarrow &[\![E_l]\!]C \wedge nsem1(N, C, \mathfrak{p}, C') \wedge nsem^m(N, C', \mathfrak{p}, C'') \\
\Rightarrow &\cdots \wedge ([\![E_l]\!]C \wedge nsem1(N, C, \mathfrak{p}, C') \Rightarrow [\![E_{l'}]\!]C') \\
&\wedge (nsem^m(N, C', \mathfrak{p}, C'') \wedge [\![E_{l'}]\!]C' \Rightarrow [\![E_{l''}]\!]C'') \\
\Rightarrow &[\![E_{l''}]\!]
\end{aligned}$$

Which proves the inductive step under the assumptions that

$$\forall C \equiv \langle S, \sigma, l\rangle, C' \equiv \langle S', \sigma', l'\rangle : nsem1(N, C, \mathfrak{p}, C') \wedge [\![E_l]\!]C \Rightarrow [\![E_{l'}]\!]C')$$

actually holds. This is what we're going to prove next.

Proof of Single-Step Soundness This part of the soundness proof is later recycled for the completeness proof.

$$\forall C \equiv \langle S, \sigma, l\rangle, C' \equiv \langle S', \sigma', l'\rangle : nsem1(N, C, \mathfrak{p}, C') \wedge [\![E_l]\!]C \Rightarrow [\![E_{l'}]\!]C')$$

We assume $nsem1(...) \wedge [\![E_l]\!]C$ and then show that this implies $[\![E_{l'}]\!]C'$ by induction on the shape of the derivation tree of the assertion $\{E_l\}\, l : I_l$. Although the inductive hypothesis will be needed only for the invokevirtual rule. All other rules are shallow: they have only logical formulas as antecedents. So we can prove them directly. The proof is simple for primitive instructions and uses the following two lemmas. invokevirtual is more difficult because it has to cope with the reference to some virtual method $T : m$.

Lemma 5.

$$[\![E]\!]\langle S, \sigma, l\rangle \iff [\![\mathrm{shift}^{|\kappa|}(E)]\!]\langle S, (\sigma, \kappa), l\rangle$$

Proof by induction on the structure of the expression.

Lemma 6.

$$\begin{aligned}&[\![E[s_0/s(i_0), \ldots, s_n/s(i_n), y_0/x_0, \ldots, y_m/x_m]]\!]\langle S, \sigma, l\rangle \iff \\ &\qquad [\![E]\!]\langle S[x_0 \mapsto [\![y_0]\!]\langle S, \sigma, l\rangle, \ldots, x_m \mapsto [\![y_m]\!]\langle S, \sigma, l\rangle], \\ &\qquad\qquad \sigma[i_0 \mapsto [\![s_0]\!]\langle S, \sigma, l\rangle, \ldots, i_n \mapsto [\![s_n]\!]\langle S, \sigma, l\rangle], l\rangle\end{aligned}$$

Proof by induction on the structure of the expression.

As mentioned before, the proofs always follow the same pattern: We assume $nsem1(N, C, \mathfrak{p}, C') \wedge [\![E_l]\!]C$ and deduce, with the antecedents of the rule that was used to infer E_l, that $[\![E_{l'}]\!]$ holds. The operational semantics is deterministic. We therefore have to look at only one possible derivation for $nsem1(N, C, \mathfrak{p}, C')$. Keep in mind, that we're considering the cases

$$I_l \neq \texttt{invokevirtual}\ T : m$$

first and therefore $nsem1(N, ., \mathfrak{p}, .) = (\rightarrow)$ according to the definition of $nsem1$.

`pushc` v We know: $E_l \rightarrow \mathrm{unshift}(E_{l+1}[v/s(0)])$

$$\begin{aligned}&[\![E_l]\!]\langle S, \sigma\rangle \\ &\Rightarrow [\![\mathrm{unshift}(E_{l+1}[v/s(0)])]\!]\langle S, \sigma\rangle \\ \iff &[\![E_{l+1}[v/s(0)]]\!]\langle S, (\sigma, t)\rangle \\ \iff &[\![E_{l+1}]\!]\langle S, (\sigma, v)\rangle\end{aligned}$$

`pushv` x exactly as above for `pushc` v

`pop` x We know: $E_l \to (\text{shift}(E_{l+1}))[s(0)/x]$

$$\begin{aligned} & [\![E_l]\!]\langle S, (\sigma, v)\rangle \\ \Rightarrow & [\![(\text{shift}(E_{l+1}))[s(0)/x]]\!]\langle S, (\sigma, v)\rangle \\ \iff & [\![(\text{shift}(E_{l+1}))]\!]\langle S[x \mapsto v], (\sigma, v)\rangle \\ \iff & [\![E_{l+1}]\!]\langle S[x \mapsto v], \sigma\rangle \end{aligned}$$

`binop`$_{\text{op}}$ We know: $E_l \to (\text{shift}(E_{l+1}))[(s(1)\,\text{op}\,s(0))/s(1)]$

$$\begin{aligned} & [\![E_l]\!]\langle S, (\sigma, v_1, v_2)\rangle \\ \Rightarrow & [\![(\text{shift}(E_{l+1}))]\!]\langle S, (\sigma, (v_1\,\text{op}\,v_2), v_2)\rangle \\ \iff & [\![E_{l+1}]\!]\langle S, (\sigma, v_1\,\text{op}\,v_2)\rangle \end{aligned}$$

`goto` l' We know: $E_l \to E_{l'}$

$$\begin{aligned} & [\![E_l]\!]\langle S, \sigma\rangle \\ \Rightarrow & [\![E_{l'}]\!]\langle S, \sigma\rangle \end{aligned}$$

`brtrue` l' We know

$$E_l \to (\neg s(0) \to \text{shift}(E_{l+1})) \wedge (s(0) \to \text{shift}(E_{l'}))$$

Case 1: $s(0) = true$

$$\begin{aligned} & [\![E_l]\!]\langle S, (\sigma, true)\rangle \\ \Rightarrow & [\![\neg s(0) \to \text{shift}(E_{l+1})) \wedge (s(0) \to \text{shift}(E_{l'})]\!]\langle S, (\sigma, true)\rangle \\ \iff & [\![\text{shift}(E_{l'})]\!]\langle S, (\sigma, true)\rangle \\ \iff & [\![E_{l'}]\!]\langle S, \sigma\rangle \end{aligned}$$

Case 2: $s(0) = false$: similar

`checkcast` T We know: $E_l \to E_{l+1} \wedge \tau(s(0)) \preceq T$

$$\begin{aligned} & [\![E_l]\!]\langle S, (\sigma, v)\rangle \\ \Rightarrow & [\![E_{l+1} \wedge \tau(s(0)) \preceq T]\!]\langle S, (\sigma, v)\rangle \\ \iff & [\![E_{l+1}]\!]\langle S, (\sigma, v)\rangle \wedge [\![\tau(s(0)) \preceq T]\!]\langle S, (\sigma, v)\rangle \\ \iff & [\![E_{l+1}]\!]\langle S, (\sigma, v)\rangle \wedge \underbrace{\tau(v) \preceq T}_{true\ (nsem1(..)\ \text{holds})} \\ \iff & [\![E_{l+1}]\!]\langle S, (\sigma, v)\rangle \end{aligned}$$

`newobj` T We know: $E_l \to \text{unshift}(E_{l+1}[\text{new}(\$, T)/s(0), \$\langle T\rangle/\,\$])$

$$
\begin{aligned}
&[\![E_l]\!]\langle S, \sigma\rangle\\
\Rightarrow&[\![\text{unshift}(E_{l+1}[\text{new}(\$, T)/s(0), \$\langle T\rangle/\,\$])]\!]\langle S, \sigma\rangle\\
\iff& [\![E_{l+1}]\!]S[\$ \mapsto S(\$)\langle T\rangle], (\sigma, \text{new}(S(\$), T))
\end{aligned}
$$

`getfield` $T@a$ We know: $E_l \to E_{l+1}[\$(\text{iv}(s(0), T@a))/s(0)] \wedge s(0) \neq \text{ null}$

$$
\begin{aligned}
&[\![E_l]\!]\langle S, (\sigma, y)\rangle\\
\Rightarrow&[\![E_{l+1}[\$(\text{iv}(s(0), T@a))/s(0)] \wedge s(0) \neq \text{ null}]\!]\langle S, (\sigma, y)\rangle\\
\iff& [\![E_{l+1}[\$(\text{iv}(s(0), T@a))/s(0)]]\!]\langle S, (\sigma, y)\rangle \wedge \underbrace{y \neq \text{ null}}_{nsem(\ldots)}\\
\iff& [\![E_{l+1}[\$(\text{iv}(s(0), T@a))/s(0)]]\!]\langle S, (\sigma, y)\rangle \wedge true\\
\iff& [\![E_{l+1}[\$(\text{iv}(s(0), T@a))/s(0)]]\!]\langle S, (\sigma, y)\rangle\\
\iff& [\![E_{l+1}]\!]\langle S, (\sigma, S(\$)(\text{iv}(y, T@a)))\rangle
\end{aligned}
$$

`putfield` $T@a$ We know:

$$E_l \to (\text{shift}^2(E_{l+1}))[\$\langle \text{iv}(s(1), T@a) := s(0)\rangle/\,\$] \wedge s(1) \neq \text{ null}$$

$$
\begin{aligned}
&[\![E_l]\!]\langle S, (\sigma, y, v)\rangle\\
\Rightarrow&[\![(\text{shift}^2(E_{l+1}))[\$\langle \text{iv}(s(1), T@a) := s(0)\rangle/\,\$] \wedge s(1) \neq \text{ null}]\!]\langle S, (\sigma, y, v)\rangle\\
\iff& [\![(\text{shift}^2(E_{l+1}))[\$\langle \text{iv}(s(1), T@a) := s(0)\rangle/\,\$]]\!]\langle S, (\sigma, y, v)\rangle \wedge y \neq \text{ null}\\
\iff& [\![(\text{shift}^2(E_{l+1}))[\$\langle \text{iv}(s(1), T@a) := s(0)\rangle/\,\$]]\!]\langle S, (\sigma, y, v)\rangle \wedge true\\
\iff& [\![(\text{shift}^2(E_{l+1}))]\!]\langle S[\$ \mapsto S(\$)\langle \text{iv}(y, T@a) := v\rangle], (\sigma, y, v)\rangle\\
\iff& [\![E_{l+1}]\!]\langle S[\$ \mapsto S(\$)\langle \text{iv}(y, T@a) := v\rangle], \sigma\rangle
\end{aligned}
$$

Single-Step Soundness for the invokevirtual Rule E_l has been deduced by the invokevirtual rule:

$$
\text{invokevirtual}\,\frac{\begin{array}{c}\mathcal{A} \vdash \{P\}\ T : m\ \{Q\}\\ L \text{ is a vector of logical variables}\\ w \text{ is a vector of local or a stack elements } \neq s(0)\\ E_l \to s(1) \neq \text{ null} \wedge P[s(1)/\,\text{this}, s(0)/\,\text{p}][\text{shift}(w)/L]\\ Q[s(0)/\,\text{result}][w/L] \to E_{l+1}\end{array}}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{invokevirtual}\ T : m}
$$

we have to prove:

$$\forall C \equiv \langle S, \sigma, l\rangle, C' \equiv \langle S', \sigma', l_2\rangle : nsem1(N, C, \mathfrak{p}, C') \wedge [\![E_l]\!]C \Rightarrow [\![E_{l_2}]\!]C')$$

($nsem1(N, C, \mathfrak{p}, C')$ is possible because $N > 0$).

We assume $nsem1(N, C, \mathfrak{p}, C') \wedge [\![E_l]\!]C$ and then deduce $[\![E_{l_2}]\!]C'$. Here are the assumptions in detail:

$$C = \langle S, (\sigma, y, v), l\rangle \tag{1}$$
$$C' = \langle S[\$ \mapsto S'(\$)], (\sigma, S'(\text{result})), l+1\rangle \tag{2}$$

i.e. $l_2 = l + 1$ and

$$nsem(N-1, \langle\{\text{this} \mapsto y, \text{p} \mapsto v, \$ \mapsto S(\$)\}, (), 0\rangle, \mathfrak{p}', \langle S', \sigma', l'\rangle) \tag{3}$$

where $\mathfrak{p}' = body_{\text{VM}_\text{K}}(impl(\tau(y), m))$ and

$$\mathfrak{p}'(l') = \texttt{end\_method} \tag{4}$$

$$[\![E_l]\!]C \tag{5}$$

According to the derivation of E_l we can also assume that

$$\begin{aligned}
&\mathcal{A} \vdash \{P\}\ T : m\ \{Q\},\\
&E_l \to s(1) \neq \text{null} \wedge P[s(1)/\,\text{this}, s(0)/\,\text{p}][\text{shift}(w)/L] \text{ and}\\
&Q[s(0)/\,\text{result}][w/L] \to E_{l+1}
\end{aligned}$$

From

$$\mathcal{A} \vdash \{P\}\ T : m\ \{Q\}$$

we deduce with the inductive hypothesis that

$$\forall M : K(M, P, T : m, Q)$$

therefore

$$\forall S \preceq T : K(M, \tau(\text{this}) \preceq S \wedge P, impl(S, m), Q)$$

from the well-formedness we know that

$$\tau(y) \preceq T$$

and we can rewrite the formula as

$$\begin{aligned}
K(M, y \neq \text{null} \wedge \tau(\text{this}) \preceq \tau(y) \wedge P,&\\
&body_{\text{VM}_\text{K}}(impl(\tau(y), m)), Q)
\end{aligned}$$

or equivalently

$$K(M, y \neq \text{null} \wedge \tau(\text{this}) \preceq \tau(y) \wedge P, \mathfrak{p}', Q)$$

Replacing the definition of K:

$$\begin{aligned}
\forall M, Z \equiv \langle\{\text{this} \mapsto \text{this}_0, \text{p} \mapsto \text{p}_0, \$ \mapsto \$_0\}, (), 0\rangle, Z' \equiv \langle A', \sigma', l'\rangle :&\\
nsem(M, Z, \mathfrak{p}', Z') \wedge I_{l'} = \texttt{end_method}&\\
\wedge [\![y \neq \text{null} \wedge \tau(\text{this}) \preceq \tau(y) \wedge P]\!]Z&\\
&\Rightarrow [\![Q]\!]Z'
\end{aligned}$$

Setting

$$Z = \langle \{\text{this} \mapsto y, \text{p} \mapsto v, \$ \mapsto S(\$)\}, (), 0\rangle$$
$$Z' = \langle S', \sigma', l'\rangle$$

and

$$M = N - 1$$

we are almost done:

$$\begin{aligned} &nsem(N-1, \langle \{\text{this} \mapsto y, \text{p} \mapsto v, \$ \mapsto S(\$)\}, (), 0\rangle, \mathfrak{p}', \langle S', \sigma', l'\rangle) \wedge \\ &\qquad I_{l'} = \texttt{end\_method} \wedge [\![y \neq \text{ null } \wedge \tau(\text{this}) \preceq \tau(y) \wedge P]\!]Z \\ &\qquad\qquad \Rightarrow [\![Q]\!]Z' \end{aligned}$$

Because we can conclude by (3) and equation (4) on the preceding page that

$$[\![y \neq \text{ null } \wedge \tau(\text{this}) \preceq \tau(y) \wedge P]\!]Z \Rightarrow [\![Q]\!]Z'$$

To make it clear, lets replace Z, Z'

$$\begin{aligned} &[\![y \neq \text{ null } \wedge \tau(\text{this}) \preceq \tau(y) \wedge P]\!] \\ &\qquad \langle \{\text{this} \mapsto y, \text{p} \mapsto v, \$ \mapsto S(\$)\}, (), 0\rangle \\ &\qquad\qquad \Rightarrow [\![Q]\!]\langle S', \sigma', l'\rangle \end{aligned}$$

and after some evaluation:

$$\begin{aligned} &[\![y \neq \text{ null } \wedge \tau(y) \preceq \tau(y) \wedge P]\!] \\ &\qquad \langle \{\text{this} \mapsto y, \text{p} \mapsto v, \$ \mapsto S(\$)\}, (), 0\rangle \\ &\qquad\qquad \Rightarrow [\![Q]\!]\langle S', \sigma', l'\rangle \end{aligned}$$

we recognize that we can equally well write

$$[\![y \neq \text{ null } \wedge P]\!]\langle \{\text{this} \mapsto y, \text{p} \mapsto v, \$ \mapsto S(\$)\}, (), 0\rangle \Rightarrow [\![Q]\!]\langle S', \sigma'\rangle$$

Taking advantage of our assumption $[\![E_l]\!]C$ (equation (5) on the previous page) we can also use

$$[\![s(1) \neq \text{ null } \wedge P[s(1)/\text{ this}, s(0)/\text{ p}][\text{shift}(w)/L]]\!]C$$

or just as well when considering the structure of C:

$$[\![s(1) \neq \text{ null } \wedge P[s(1)/\text{ this}, s(0)/\text{ p}][\text{shift}(w)/L]]\!]\langle S, (\sigma, y, v)\rangle$$

Let's first reformulate by separate the auxiliary substitution $[\text{shift}(w)/L]$:

$$([\![s(1) \neq \text{ null } \wedge P[s(1)/\text{ this}, s(0)/\text{ p}]]\!]\langle S, (\sigma, y, v)\rangle)[[\![w]\!]\langle S, (\sigma, y), l\rangle/L]$$

We use the substitution lemma and the fact that P does only depend on this, on the parameter p and the object store \$, i.e. its value does not change when we reduce the evaluation environment:

$$\underbrace{([\![y \neq \text{ null} \wedge P]\!]\langle\{\text{this} \mapsto y, \text{p} \mapsto v, \$ \mapsto S(\$)\}, (), 0\rangle)}_{\text{lhs assumpt.}}[[\![w]\!]\langle S, (\sigma, y)\rangle / L]$$

The left hand side of the implication we have deduced from the assumption can be replaced by its right hand side! We can deduce

$$([\![Q]\!]\langle S', \sigma'\rangle)\,\underbrace{[[\![w]\!]\langle S, (\sigma, y)\rangle / L]}_{\text{just carry around}}$$

By the same line of reasoning as before (Q does only depend on result and \$):

$$([\![Q[s(0)/\text{ result}]]\!]\langle S[\$ \to S'(\$)], (\sigma', S'(\text{result}))\rangle)[[\![w]\!]\langle S, (\sigma, y)\rangle / L]$$

We can re-integrate the substitution of L taking advantage of the fact that $w \neq s(0)$, i.e., $[[\![w]\!]\langle S, (\sigma, x)\rangle$ is the same for all x:

$$[\![Q[s(0)/\text{ result}][w/L]]\!]\langle S[\$ \to S'(\$)], (\sigma', S'(\text{result}))\rangle$$

With the implication $Q[s(0)/\text{ result}][w/L] \to E_{l+1}$

$$[\![E_{l+1}]\!]\langle S[\$ \to S'(\$)], (\sigma', S'(\text{result}))\rangle$$

Which is what we wanted to prove. Done.

4.2 Completeness and Weakest Preconditions

In this section, we prove the following theorem

if $\models \{P\}\ \mathfrak{p}\ \{Q\}$ then $\vdash \{P\}\ \mathfrak{p}\ \{Q\}$ for any method body $\mathfrak{p}$.

There are limitations of our notion of completeness due to modularity concerns. They are discussed in section 4.2.1 on page 45. Unlike checking soundness of a programming logic – a mere necessity – completeness can reveal real new insights. The weakest precondition (wp) approach to proving completeness chosen here yields a constructive method[16] to derive for any given method postcondition[17] a weakest precondition which is implied by all *valid preconditions* if the program terminates. This is helpful and often used for automatic/interactive program verification.

Proving completeness of Hoare-style programming logics using a wlp-calculus is a standard technique. There are other, less commonly used and less rewarding approaches that do not fit nicely into our framework. ([Kle99])

In *structured programming languages*, the weakest liberal precondition $\text{wlp}(S, Q)$ of a statement S for a predicate Q is defined as[18]

$$[\![\text{wlp}(S, Q)]\!]s \equiv \langle S, s\rangle \to s' \Rightarrow [\![Q]\!]s'$$

[16] up to expressiveness problems of the assertion language
[17] i.e., for any desired *result* condition of a method
[18] $\langle S, s\rangle \to s'$ is the big step transition relation from state s to s'.

Trivial consequences of this definition are

$$\models \{\mathrm{wlp}(S,Q)\}\ S\ \{Q\}$$

and

$$(\models \{P\}\ S\ \{Q\}) \Rightarrow (P \Rightarrow \mathrm{wlp}(S,Q))$$

The completeness theorem follows if we can deduce

$$\vdash \{\mathrm{wlp}(S,Q)\}\ S\ \{Q\}$$

because then, there is a corresponding derivation tree for all valid $\{P\}\ S\ \{Q\}$

$$\text{consequence}\ \frac{P \rightarrow \mathrm{wlp}(S,Q) \qquad \{\mathrm{wlp}(S,Q)\}\ S\ \{Q\}}{\{P\}\ S\ \{Q\}}$$

As explained in section 3.2.3 on page 23, we cannot prove a program correct using our rules if it can get stuck due to invalid object operations. We will thus only prove completeness for programs that are guaranteed to terminate. I.e., we will prove that the weakest precondition for total correctness can be derived.

The weakest precondition for total correctness for method bodies $\mathrm{wp}(\mathfrak{p},Q)$ is constrained by all preconditions of all individual instructions of $\mathfrak{p}$. We shall therefore construct all the preconditions of all instructions simultaneously such that the desired postcondition Q holds in the terminating state (when the `end_method` instruction is reached). We define another wp-like attribute for the preconditions of all instructions:

Definition 17.

$$[\![\mathrm{wp}_{\mathfrak{p}}(l,Q)]\!]\langle S,\sigma\rangle \equiv [\![Q]\!]\langle S',\sigma'\rangle \wedge \mathfrak{p}; \langle S,\sigma,l\rangle \rightarrow^{*} \langle S',\sigma',|\mathfrak{p}|\rangle$$

If we can prove the assertion $\{\mathrm{wp}_{\mathfrak{p}}(l,Q)\}\, l : I_l$ for all instructions in any given method body, then the our programming logic for bytecode is definitely relatively complete.

The proof is organized as follows: We first define a predicate that can be deduced given a method body $\mathfrak{p}$ and its postcondition Q and then show that it actually *is at least as weak as* the predicate $\mathrm{wp}_{\mathfrak{p}}$.

1. We define instruction specifications ψ_l that can be deduced given a method body postcondition Q for $\mathfrak{p}$
2. We show that $\psi_l \Leftarrow \mathrm{wp}_{\mathfrak{p}}(l,Q)$

Definition 18. For a given postcondition Q of a method implementation $\mathfrak{p}$ we define

$$\begin{aligned} \psi_{|\mathfrak{p}|} &= Q \\ \psi_l^{(0)} &= \mathit{false} \\ \psi_l^{(k+1)} &= \mathrm{wp}^1_{\mathfrak{p}}(I_l, (\psi_i^{(k)})_{i \in succ(l:I_l)}) \\ \psi_l &= \bigvee_{n \in \mathbb{N}_0} \psi_l^{(n)} \end{aligned}$$

In order to show that $\{\psi_l\}\, l : I_l$ is derivable, we prove that

$$\psi_l \iff \mathrm{wp}^1_{\mathfrak{p}}(I_l, (\psi_i)_{i \in succ(l:I_l)}) \tag{6}$$

holds. Assume ψ_l. There must be an $m > 0$ such that $\psi_l^{(m)}$. From its definition, we know that

$$\mathrm{wp}^1_{\mathfrak{p}}(I_l, (\psi_i^{(m-1)})_{i \in succ(l:I_l)})$$

or equivalently

$$\bigvee_{n \in \mathbb{N}_0} \mathrm{wp}^1_{\mathfrak{p}}(I_l, (\psi_i^{(n)})_{i \in succ(l:I_l)})$$

Lemma 1 tells us that this is equal to

$$\mathrm{wp}^1_{\mathfrak{p}}(I_l, (\underbrace{\bigvee_{n \in \mathbb{N}_0} \psi_i^{(n)}}_{\psi_i})_{i \in succ(l:I_l)})$$

Therefore $\mathrm{wp}^1_{\mathfrak{p}}(I_l, (\psi_i)_{i \in succ(l:I_l)})$ which is what we wanted to prove. The opposite direction is similar. Done.

To see that equation (6) holds, it is even easier just to rewrite the expression:

$$\begin{aligned}
& \mathrm{wp}^1_{\mathfrak{p}}(I_l, (\psi_i)_{i \in succ(l:I_l)}) \\
\iff & \mathrm{wp}^1_{\mathfrak{p}}(I_l, (\bigvee_{n \in \mathbb{N}_0} \psi_i^{(n)})_{i \in succ(l:I_l)}) \\
\iff & \bigvee_{n \in \mathbb{N}_0} \mathrm{wp}^1_{\mathfrak{p}}(I_l, (\psi_i^{(n)})_{i \in succ(l:I_l)}) \\
\iff & \bigvee_{n \in \mathbb{N}_0} \psi_l^{(n+1)} \\
\iff & \bigvee_{n \in \mathbb{N}_0} \psi_l^{(n+1)} \vee false \\
\iff & \bigvee_{n \in \mathbb{N}_0} \psi_l^{(n)} \\
\iff & \psi_l
\end{aligned}$$

showing $\psi_l \Leftarrow \mathrm{wp}_{\mathfrak{p}}(l, Q)$ We have to prove

$$[\![\psi_l]\!]\langle S, \sigma, l\rangle \Leftarrow [\![Q]\!]\langle S', \sigma', |\mathfrak{p}|\rangle \wedge \mathfrak{p}; \langle S, \sigma, l\rangle \to^* \langle S', \sigma', |\mathfrak{p}|\rangle$$

We prove the more general

$$[\![\psi_l]\!]\langle S, \sigma, l\rangle \Leftarrow [\![\psi_{l'}]\!]\langle S', \sigma', l'\rangle \wedge \mathfrak{p}; \langle S, \sigma, l\rangle \to^* \langle S', \sigma', l'\rangle$$

All the soundness proves for individual instructions (section 4.1.1 on page 36) have the same structure:

$$\begin{aligned}
& [\![E_l]\!]\langle S, \sigma\rangle \\
\Rightarrow & [\![\mathrm{wp}^1_{\mathfrak{p}}(E_l, \ldots)]\!]\langle S, \sigma\rangle \\
\iff & \text{[some equivalence transformations]} \\
\iff & [\![E_{l'}]\!]\langle S', \sigma'\rangle
\end{aligned}$$

Reading the soundness proves in the opposite direction, we can follow

$$[\![\mathrm{wp}^1_{\mathtt{p}}(E_l, (E_i)_{i\in succ(l:I_l)})]\!]\langle S, \sigma\rangle$$

from $[\![E_{l'}]\!]\langle S', \sigma'\rangle$ by reading the soundness proof in the opposite direction. Just as for soundness an induction on the length of the derivation yields the required result for all derivations of arbitrary length.

The reader may have noticed that

$$\begin{aligned}
\phi_{|\mathtt{p}|} &= Q\\
\phi_l^{(0)} &= true\\
\phi_l^{(k+1)} &= \mathrm{wp}^1_{\mathtt{p}}(I_l, (\phi_i^{(k)})_{i\in succ(l:I_l)})\\
\phi_l &= \bigwedge_{n\in\mathbb{N}_0} \phi_l^{(n)}
\end{aligned}$$

do satisfy the conditions on the weakest preconditions as well. The difference is that ϕ_l is the greatest fixed point and ψ is the least fixed point of $\mathrm{wp}_{\mathtt{p}}$. The greatest fixed point normally coincides with the weakest liberal precondition. As explained above, this is not the case for the $\mathrm{VM_K}$ bytecode language because of the `checkcast` T, `getfield` $T@a$, `putfield` $T@a$ and invocation instructions.

Example 18. Consider the following code snippet

```
0: goto 0
1: end_method
```

We want to find the weakest precondition for this method:

k	$\psi_0^{(k)}$	$\psi_1^{(k)}$
0	$false$	Q
1	$\mathrm{wp}^1_{\mathtt{p}}(\mathtt{goto}\,, false) = false$	Q
2	$\mathrm{wp}^1_{\mathtt{p}}(\mathtt{goto}\,, false) = false$	Q
$\vdots$		

We derive that $\psi_0 = false$ indicating that the program does not terminate. We should expect the weaker precondition ϕ to be $true$ because the program always loops.

k	$\phi_0^{(k)}$	$\phi_1^{(k)}$
0	$true$	Q
1	$\mathrm{wp}^1_{\mathtt{p}}(\mathtt{goto}\,, true) = true$	Q
2	$\mathrm{wp}^1_{\mathtt{p}}(\mathtt{goto}\,, true) = true$	Q
$\vdots$		

Example 19. ϕ_0 and ψ_0 should both be $false$ for the following method because it gets stuck (S and T are unrelated):

```
0: newobj T
1: checkcast S
2: end_method
```

k	$\psi_0^{(k)}$	$\psi_1^{(k)}$	$\psi_2^{(k)}$
0	$false$	$false$	Q
1	$false$	$Q \wedge \tau(s(0)) \preceq S$	Q
2	$Q \wedge \tau(\text{new}(\$, T)) \preceq S$	$Q \wedge \tau(s(0)) \preceq S$	Q
$\vdots$			

ψ_0 is therefore $Q \wedge \tau(\text{new}(\$, T)) \preceq S$, which according to axiom (env11) in [PH97]:

$$\tau(\text{new}(\$, T)) = T$$

is $false$.

For ϕ_0, we get the same result

k	$\phi_0^{(k)}$	$\phi_1^{(k)}$	$\phi_2^{(k)}$
0	$true$	$true$	Q
1	$true$	$Q \wedge \tau(s(0)) \preceq S$	Q
2	$Q \wedge \tau(\text{new}(\$, T)) \preceq S$	$Q \wedge \tau(s(0)) \preceq S$	Q
$\vdots$			

4.2.1 Invocations

Until now, we did not define a *local precondition function* for invocation instructions: $\text{wp}^1_{\mathtt{p}}(\texttt{invokevirtual } T : m, \ldots)$. The reason is simple. We're considering a scenario for program verification where the programmer associates with every method one or more specifications. The method specifications are fixed at the beginning and the code is verified to conform to these specifications. This allows *modular reasoning*. For verifying a method, we do not have to follow the invocations and verify all the transitively called methods. It is sufficient to use the known method specifications as summaries of the effect of a method body. These method specifications are however not tailored towards a specific call site. Hence there can be no weakest precondition for a method invocation given a method specification for the invoked method. Giving up method this modularity idea, we could extend the fixed point iteration to work simultaneously on all method bodies in a program. This is completely impractical. Furthermore, programs are often open to extensions in a highly dynamic environment.

Instead of *weakest* preconditions for method invocations, we will instead consider preconditions that are weak enough in practice. The idea is based on [Rau02].

We assume $\{P\}\ T:m\ \{R\}$

$$\begin{aligned}&\mathrm{wp}^1_{\mathtt{p}}(\texttt{invokevirtual}\ T:m, E_{l+1}) = P[s(1)/\,\mathrm{this}, s(0)/\,\mathrm{p}]\\ &\wedge (\forall T, H : \rho(s(1), s(0), \$, H, E) \wedge R[T/\,\mathrm{result}, H/\,\$] \Rightarrow E_{l+1}[T/s(0), H/\,\$])\end{aligned}$$

$\rho(s(1), s(0), \$, H, E)$ is a general constraint on the object, its argument and the object store, and the possible end values H and E. ρ is used to weaken the postcondition. A good ρ is crucial to make this weak precondition usable. Details can be found in [Rau02].

5 Application: Deriving Rules For Complex Instructions

Axiomatic definitions for instructions whose effect is defined as the effect of a sequential execution of simpler instructions can be easily derived. We call these instructions "compound instructions". To see how rules for them are assembled, let's make the deductive system more restrictive. The premises for primitive instructions have the following form:

$$E_l \rightarrow \mathrm{wp}^1_{\mathtt{p}}(I_l, (E_i)_{i \in succ(l:I_l)})$$

We now change the implication to an equivalence

$$E_l \longleftrightarrow \mathrm{wp}^1_{\mathtt{p}}(I_l, (E_i)_{i \in succ(l:I_l)})$$

Soundness is obviously not affected. Neither is completeness: the weakest preconditions ψ and ϕ actually satisfy the desired equivalence. For a basic block B of instructions,

$$\begin{aligned}1 :&B_1\\ 2 :&B_2\\ &\vdots\\ |B| :&B_{|B|}\end{aligned}$$

valid specifications β satisfy the equations

$$\begin{aligned}&\beta_1 \longleftrightarrow \mathrm{wp}^1_{\mathtt{p}}(B_1, \beta_2)\\ &\vdots\\ &\beta_{|B|-1} \longleftrightarrow \mathrm{wp}^1_{\mathtt{p}}(B_{|B|-1}, \beta_{|B|})\end{aligned}$$

These equations can be substituted into each other yielding for a block B the rule for which we replace $\longleftrightarrow$ by $\rightarrow$ to make it less awkward to use.

$$\mathrm{B}\frac{E_l \rightarrow \mathrm{wp}^1_{\mathtt{p}}(B_1, \mathrm{wp}^1_{\mathtt{p}}(B_2, \cdots \mathrm{wp}^1_{\mathtt{p}}(B_{|B|}, \overbrace{(E_t)_{t \in succ(l.|B|:B_{|B|})}}^{(E_i)_{i \in succ(l:B)}})))}{\{E_l\}\, l : B} \tag{7}$$

Replacing $\rightarrow$ by $\longleftrightarrow$ also helps when blocks contain method invocations. The corresponding (simplified) invokevirtual rule is:

$$\text{invokevirtual}\,\frac{\begin{array}{c}\mathcal{A} \vdash \{P\}\ T : m\ \{Q\} \\ E_l \longleftrightarrow s(1) \neq \text{ null} \wedge P[s(1)/\,\text{this}, s(0)/\,\text{p}] \\ Q[s(0)/\,\text{result}] \longleftrightarrow E_{l+1}\end{array}}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{invokevirtual}\ T : m}$$

For blocks of instructions with loops, their weakest precondition as an instruction is the weakest precondition of the block as an implementation:

$$\text{wp}^1_{\mathfrak{p}}(B, (E_i)_{i \in succ(l:B)}) = \text{wp}_B(1, (E_i)_{i \in succ(l:B)}) \tag{8}$$

Their rules naturally have the antecedent $E_l \rightarrow \text{wp}_B(1, E_{l+1})$.

These constructions can be shown to conform to an operational interpretation of compound instructions as macros:[19]

$$\begin{array}{rl} 0 :& I_0 \\ 1 :& I_1 \\ & \vdots \\ l :& B \\ & \vdots \\ |\mathfrak{p}| :& \texttt{end\_method} \end{array} \quad \equiv \quad \begin{array}{rl} 0 :& I_0 \\ 1 :& I_1 \\ & \vdots \\ l-1 :& I_{l-1} \\ l.1 :& B_1 \\ l.2 :& B_2 \\ & \vdots \\ l.|B| :& B_{|B|} \\ l+1 :& I_{l+1} \\ & \vdots \\ |\mathfrak{p}| :& \texttt{end\_method} \end{array}$$

Mutual implication by induction on the shape of the derivation shows that the expansion is equivalent to assigning the following semantics to a block B:

$$\frac{\begin{array}{c} l' \neq l.i \\ B; \langle S, \sigma, l.1 \rangle \rightarrow^* \langle S', \sigma', l' \rangle \end{array}}{[\ldots\ l : B\ \ldots]; \langle S, \sigma, l \rangle \rightarrow \langle S', \sigma', l' \rangle}$$

[19] To make this argument formal, we should redefine the operational semantics, relax the conditions on labels and introduce a function $nextlabel(l)$ that is used instead of $l+1$ in the operational semantics to access the label of the next instruction in a sequence.

$$nextlabel(l-1) = \begin{cases} l.1 & \text{if } I_l \text{ is a block} \\ l & \text{otherwise} \end{cases}$$

$$nextlabel(l.i) = \begin{cases} l+1 & \text{if } i = |I_l| \\ l.(i+1) & \text{otherwise} \end{cases}$$

To see that equation (8) on the preceding page yields correct results, remember that $\mathrm{wp}_B(1, E_{l+1})$ constructs preconditions β for which

$$\beta_i \iff \mathrm{wp}^1_{\mathfrak{p}}(B_i, (\beta_i)_{i \in succ(l.i:B_i)})$$

The β_is can thus be identified with $E_{l.i}$ yieliding a proof for the in-line version of the program. The other assemblage rule, replacing $\rightarrow$ by $\longleftrightarrow$ is only a practical simplification and works for the same reason. The βs can be computed directly when the equations are not recursive, so we do not need a fixed point iteration.

Example 20. The `goto` l' instruction may be defined as

$$\texttt{goto} \;\equiv\; \begin{cases} 1: & \texttt{pushc}\ \mathit{true} \\ 2: & \texttt{brtrue}\ l' \end{cases}$$

Its operational semantics is

$$\frac{[1:\texttt{pushc}\ \mathit{true}, 2:\texttt{brtrue}\ l']; \langle S, \sigma, l.1\rangle \rightarrow^* \langle S'', \sigma'', l''\rangle}{[\ldots\ l:\texttt{goto}\ l'\ \ldots]; \langle S, \sigma, l\rangle \rightarrow \langle S'', \sigma'', l''\rangle}$$

Substituting the finite transition relation $\rightarrow^*$ for $[1:\texttt{pushc}\ \mathit{true}, 2:\texttt{brtrue}\ l']$ results in

$$\frac{\begin{array}{c}\langle S, \sigma, l.1\rangle \rightarrow \langle S, (\sigma, \mathit{true}), l.2\rangle \\ \langle S, (\sigma, \mathit{true}), l.2\rangle \rightarrow \langle S, \sigma, l'\rangle\end{array}}{[\ldots\ l:\texttt{goto}\ l'\ \ldots]; \langle S, \sigma, l\rangle \rightarrow \langle S, \sigma, l'\rangle}$$

The same transition for `goto` l' we already have found before.

We now derive the antecedent of the rule for `goto` l' as a compound instruction using equation (7) on page 46 where $(E_i)_{i \in succ(l:\texttt{brtrue}\ l')} = (E_{l+1}, E_{l'})$:

$$\begin{aligned}
& E_l \rightarrow \mathrm{wp}^1_{\mathfrak{p}}(\texttt{pushc}\ \mathit{true}, \mathrm{wp}^1_{\mathfrak{p}}(\texttt{brtrue}\ l', E_{l+1}, E_{l'})) \\
\iff & E_l \rightarrow \mathrm{wp}^1_{\mathfrak{p}}(\texttt{pushc}\ \mathit{true}, (\neg s(0) \rightarrow \mathrm{shift}(E_{l+1})) \wedge (s(0) \rightarrow \mathrm{shift}(E_{l'}))) \\
\iff & E_l \rightarrow \mathrm{unshift}(((\neg s(0) \rightarrow \mathrm{shift}(E_{l+1})) \wedge (s(0) \rightarrow \mathrm{shift}(E_{l'})))[\mathit{true}/s(0)]) \\
\iff & E_l \rightarrow \mathrm{unshift}(((\neg \mathit{true} \rightarrow \mathrm{shift}(E_{l+1})) \wedge (\mathit{true} \rightarrow \mathrm{shift}(E_{l'})))) \\
\iff & E_l \rightarrow \mathrm{unshift}(\mathrm{shift}(E_{l'})) \\
\iff & E_l \rightarrow E_{l'}
\end{aligned}$$

Leading us to the rule we already had

$$\frac{E_l \rightarrow E_{l'}}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{goto}\ l'}$$

Example 21. The CLI `ret` instruction returns from the current method returning the topmost element of the stack to the caller (if the method is not void). Its translation to $\mathrm{VM_K}$ instructions is:

$$\texttt{ret}_{\mathrm{CLI}} \equiv \begin{cases} 1: & \texttt{pop}\ \mathrm{result} \\ 2: & \texttt{goto}\ |\mathfrak{p}| \end{cases}$$

The new rule is according to equation (7) on page 46

$$\frac{E_l \to \mathrm{wp}^1_{\mathtt{p}}(\mathtt{pop}\ \mathrm{result}, \mathrm{wp}^1_{\mathtt{p}}(\mathtt{goto}\ |\mathfrak{p}|, E_l + 1))}{\mathcal{A} \vdash \{E_l\}\, l : \mathtt{ret}_{\mathrm{CLI}}}$$

Replacing $\mathrm{wp}^1_{\mathtt{p}}$ by its definition gives us

$$\frac{E_l \to \mathrm{shift}(E_{|\mathfrak{p}|})[s(0)/\,\mathrm{result}]}{\mathcal{A} \vdash \{E_l\}\, l : \mathtt{ret}_{\mathrm{CLI}}}$$

Example 22. Value types in the CLI have two representations, (section 4.1, partition III, [ECM02])

- "A raw form used when a value type is embedded within another object or on the stack."
- "A boxed form, where the data in the value type is wrapped (boxed) into an object so it can exist as an independent entity."

We call the type of the boxed form $Box(T)$ and assume this class has the single field *value*. The instructions `box` T can be defined as

$$\mathtt{box}\ T \equiv \begin{cases} 1: & \mathtt{pop}\ t \\ 2: & \mathtt{newobj}\ Box(T) \\ 3: & \mathtt{dup} \\ 4: & \mathtt{pushv}\ t \\ 5: & \mathtt{putfield}\ Box(T)@value \end{cases}$$

where t is a new temporary variable. t is not present in the rule we derive:

$$\frac{\begin{array}{c} E_l \to \mathrm{wp}^1_{\mathtt{p}}(\mathtt{pop}\ t, \mathrm{wp}^1_{\mathtt{p}}(\mathtt{newobj}\ Box(T), \mathrm{wp}^1_{\mathtt{p}}(\mathtt{dup}, \mathrm{wp}^1_{\mathtt{p}}(\mathtt{pushv}\ t, \\ \mathrm{wp}^1_{\mathtt{p}}(\mathtt{putfield}\ Box(T)@a, E_{l+1}))))) \end{array}}{\mathcal{A} \vdash \{E_l\}\, l : \mathtt{box}\ T}$$

Evaluating simplifies the premise to

$$\frac{\begin{array}{c} E_l \to E_{l+1}[\$\langle Box(T)\rangle\langle \mathrm{iv}(\mathrm{new}(\$, Box(T)), Box(T)@value) := s(0)\rangle/\,\$, \\ \mathrm{new}(\$, Box(T))/s(0)] \end{array}}{\mathcal{A} \vdash \{E_l\}\, l : \mathtt{box}\ T}$$

For more examples on how blocks with jumps and method calls are used, see section 6 on extensions that are defined using compound instructions.

6 Extensions: Exception Handling and Class Initialization

This section discusses two important extensions to the basic "kernel" bytecode language and its axiomatic semantics: class initialization – the implicit call of a static method (class initializer) upon the first access of a class – and structured exception handling that has become a standard to deal with error conditions and leads to abrupt termination of parts of the program. We were careful to define as many of the new features as possible as translations to the kernel instructions.

6.1 Global Data and Class Initialization

In order to support class initialization, we need global variables. For the operational semantics, we can do so by extending the abstract state by an additional state $G : \mathit{GlobalVariable} \hookrightarrow \mathit{Value}$ for global variables. The configuration then becomes $K \equiv \langle S, G, \sigma, l\rangle$. We need two additional instructions `putstatic` and `getstatic` to access global variables just in the same way `pop` and `pushv` are used for locals. There are other possibilities to cope with global variables like extending the state $S \in \mathit{State} = \cdots \cup \mathit{Class} \hookrightarrow \mathit{Value}$ and treating classes as objects with fields.

$$[\ldots\ l : \texttt{popg}\ x\ \ldots]; \langle S, G, \sigma, l\rangle \to \langle S, G, (\sigma, G(x)), l+1\rangle$$

$$[\ldots\ l : \texttt{pushg}\ x\ \ldots]; \langle S, G, (\sigma, v), l\rangle \to \langle S, G[x \mapsto v], \sigma, l+1\rangle$$

The semantics of other instructions does not change. The exception is `invokevirtual` , for which we have to pass the global environment just like the heap \$:

$$\frac{\begin{array}{c}\mathfrak{p}' = \mathit{body}_{\mathrm{VM_K}}(\mathit{impl}(\tau(y), m)) \qquad \mathfrak{p}'(l') = \texttt{end\_method} \\ \mathfrak{p}'; \langle \{\mathrm{this} \mapsto y, \mathrm{p} \mapsto v, \$ \mapsto S(\$)\}, G, (), 0\rangle \to^* \langle S', G', \sigma', l'\rangle\end{array}}{[\ldots\ l : \cdots\ \ldots]; \langle S, G, (\sigma, y, v), l\rangle \to \langle S[\$ \mapsto S'(\$)], G', (\sigma, S'(\mathrm{result})), l+1\rangle}$$

The axiomatic semantics does not change much either. The new instructions `popg` and `pushg` are easy to handle:

pushg

$$\mathrm{pushg}\frac{E_l \to \mathrm{unshift}(E_{l+1}[x/s(0)])}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{pushg}\ x}$$

popg

$$\mathrm{popg}\frac{E_l \to (\mathrm{shift}(E_{l+1}))[s(0)/x]}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{popg}\ x}$$

To handle class initialization, we impose a structure on the global data. Global variables are static fields of classes. A static field s of a class T is denoted by

$T.s$. All classes have a special static field `initialized` and a static method `.cctor` that initializes a class and sets the `initialized` field to *true* before doing anything else.

We assume that classes are initialized only when they are first accessed by means of an instance allocation, a static field reference or a static method invocation. I.e., as late as possible.[20] This first access is abstracted by the new instruction `ensureinit` T (cf. section 5 on page 46)

$$\texttt{ensureinit}\ T \equiv \begin{cases} 1: & \texttt{pushg}\ T.\texttt{initialized} \\ 2: & \texttt{brtrue}\ 4 \\ 3: & \texttt{call}\ T@\texttt{.cctor} \\ 4: & \texttt{nop} \end{cases}$$

The semantics of the instructions that take care of class initializations are defined by the following straightforward translations:

- $\texttt{putstatic}_{\,init}\, T.s \equiv \begin{cases} 1: & \texttt{ensureinit}\ T \\ 2: & \texttt{popg}\ T.s \end{cases}$
- $\texttt{getstatic}_{\,init}\, T.s \equiv \begin{cases} 1: & \texttt{ensureinit}\ T \\ 2: & \texttt{pushg}\ T.s \end{cases}$
- $\texttt{newobj}_{\,init}\, T.m \equiv \begin{cases} 1: & \texttt{ensureinit}\ T \\ 2: & \texttt{newobj}\ T \end{cases}$
- $\texttt{invokestatic}_{\,init}\, T@m \equiv \begin{cases} 1: & \texttt{ensureinit}\ T \\ 2: & \texttt{call}\ T@m \end{cases}$

Example 23. Derivation of the axiomatic semantics for `ensureinit` T. We use the method described in section 5 on page 46 to derive a closed axiomatic definition of the `ensureinit` T instruction: replace all $\rightarrow$ by $\longleftrightarrow$. The call rule we need for static methods without arguments or results is call-static-void[21]. w and Z are vectors of locals variables/stack elements and logical variables.

$$\text{call-static-void}\,\frac{\begin{array}{c}\mathcal{A} \vdash \{P\}\ T@\texttt{.cctor}\ \{Q\} \\ E_l \longleftrightarrow P[w/Z] \\ Q[w/Z] \longleftrightarrow E_{l+1}\end{array}}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{call}\ T@\texttt{.cctor}}$$

Summarizing all obligations yields

- $\beta_1 \longleftrightarrow \text{unshift}(\beta_2[T.\texttt{initialized}/s(0)])$ from $1 : \texttt{pushg}\ T.\texttt{initialized}$

[20]This is the case for the JVM but not for the CLI, which allows classes to have the special flag `.beforefieldinit` meaning they can be initialized at any time before the first access or even afterwards but before the first field read. See section 5.5 in [LY99] and sections 8.9.5, 9.5.3 in [ECM02] or [BFGS04] for a readable ASM specification of the CLI class initialization.

[21]derived from section 3.2.3 on page 24

- $\beta_2 \longleftrightarrow (\neg s(0) \rightarrow \text{shift}(\beta_3)) \wedge (s(0) \rightarrow \text{shift}(\beta_4))$ from $2 : \texttt{brtrue}\ 4$
- $\beta_3 \longleftrightarrow P[w/Z]$ and $Q[w/Z] \longleftrightarrow \beta_4$ from $3 : \texttt{call}\ T@\texttt{.cctor}$ and
- $\beta_4 \longleftrightarrow \beta_5$ from $4 : \texttt{nop}$

Solving the equations for β_1 and β_5 yields:

$$\begin{aligned}
\beta_1 \longleftrightarrow\ & \text{unshift}(((\neg s(0) \rightarrow \text{shift}(P[w/Z])) \\
& \wedge (s(0) \rightarrow \text{shift}(Q[w/Z])))[T.\texttt{initialized}/s(0)]) \\
& \longleftrightarrow (\neg T.\texttt{initialized} \rightarrow P[w/Z]) \wedge (T.\texttt{initialized} \rightarrow \beta_5) \\
\beta_5 \longleftrightarrow\ & Q[w/Z]
\end{aligned}$$

The final rule thus is after replacing back $\longleftrightarrow$ by $\rightarrow$ for the boundary conditions:

$$\frac{\begin{array}{c} E_l \rightarrow (\neg T.\texttt{initialized} \rightarrow P[w/Z]) \wedge (T.\texttt{initialized} \rightarrow E_{l+1}) \\ Q[w/Z] \rightarrow E_{l+1} \end{array}}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{ensureinit}\ T}$$

Note that there are ambiguities which βs should be substituted (β_4 either be substituted by $Q[w/Z]$ or by E_{l+1}). Deciding for one alternative may make the resulting rule more awkward to use, but never less correct or complete as argued in section 5 on page 46.

Example 24. By the same construction as above we get for $\texttt{putstatic}\ T.s$

$$\frac{\begin{array}{c} E_l \rightarrow (\neg T.\texttt{initialized} \rightarrow P[w/Z]) \\ \wedge (T.\texttt{initialized} \rightarrow (\text{shift}(E_{l+1}))[s(0)/x]) \\ Q[w/Z] \rightarrow (\text{shift}(E_{l+1}))[s(0)/T.s] \end{array}}{\mathcal{A} \vdash \{E_l\}\, l : \texttt{putstatic}\ T.s}$$

6.2 Exception Handling

Exception handling in this section does only refer to the structured mechanism that is used to catch exceptions in languages like C♯ and Java (`try{...}catch(E e){...}`). In particular it does not discuss how `try{...}finally{...}` constructs are best compiled to bytecode. The JVM uses method local subroutines and the `jsr` instruction (chapter 7.13 in [LY99]). Even the Sun JVM implementation has known problems correctly verifying the resulting bytecode programs (chapter 16, [SBS01]). New versions of Sun's Java compiler eliminate the problems by expansion of `finally`-code. This shows that *polyvariant analyses* that allow more than one state per program point[22] and are the easiest solution to handling subroutines and also feasible in practice.

6.2.1 The Operational View

Example 25. Exceptions in the JVM are thrown by the `athrow` instruction which takes one argument from the stack:

[22] `jsr` and `ret` can then be treated like unconditional jumps to each of the various invocation sites

`athrow`

The JVM uses method local exception tables that define to where control should be transfered when some exception happens in a protected region of code:

$$ExcTable_{\mathtt{p}} = (from : Label, upto : Label, handler : Label, filtertype : Type)*$$

The exception is catched by an exception table entry if the program counter l is in the range $[from, upto)$ and$filtertype$ is compatible with the actual reference that is thrown. The evaluation stack is cleared an control is transfered to the instruction at label *handler*. Exception handling in the CLI is similar in its simplest form. It is more flexible through custom exception filters and more structured – finally blocks are made explicit. Also there are more restrictions that a method body must conform to like using `leave` instead of `br` to exit a protected region.

Example 26. Class initialization can fail. Both the CLI and the JVM mark classes whose static initializer (`.cctor`) has failed are marked as unusable. On each subsequent access, an exception is thrown. (`TypeInitializationException` or `ExceptionInInitializerError` resp.)

Exception handling in the $\mathrm{VM_K}$ works as follows: Exceptions are identified with reference types. The *State* S of the program configuration is extended to store the current exception (or null if no exception occurred):

$$State_{exc} = State \cup (\{exc\} \rightarrow Value)$$

There is an exception table for every single instruction mapping an exception type to some destination label.

$$ExcTable_{\mathtt{p},l} : Exc \mapsto l' : Type \hookrightarrow Label$$

It indicates to where control should be transfered if an exception occurred. If $ExcTable_{\mathtt{p},l}(Exc)$ returns $undef$ (is undefined) when an exception occurs, control is transfered to the `end_method` instruction and exc is not cleared (i.e., the exception is propagated). Otherwise, the evaluation stack is cleared, the exception exc pushed onto the stack and control transfered to the instruction at $ExcTable_{\mathtt{p},l}(S)$. An exception can be thrown by the new instruction `throw`.

$$\frac{l' = ExcTable_{\mathtt{p},l}(\tau(e)) \neq undef}{[\ldots\ l : \mathtt{throw}\ \ldots]; \langle S, (\sigma, e), l\rangle \rightarrow \langle S, (e), l'\rangle}$$

$$\frac{ExcTable_{\mathtt{p},l}(\tau(e)) = undef}{[\ldots\ l : \mathtt{throw}\ \ldots]; \langle S, (\sigma, e), l\rangle \rightarrow \langle S[exc \mapsto e], (), |\mathtt{p}|\rangle}$$

The lookup in $ExcTable_{\mathtt{p},l}$ is done in the poststate of an instruction. Each instruction has to take care of possible exceptions that may occur as part of their execution. Instructions are only executed when $exc = \text{null}$. exc is therefore only a device for *propagating exceptions*. Transitions for instructions that never cause an exception do not change. If the result is not an exception, the instruction terminates normally. In a suggestive notation:

$$\frac{\cdots \qquad S'(exc) = \text{null}}{[\ldots\ l : \cdots\ \ldots]; \langle S, \sigma, l\rangle \to \langle S', \sigma', l'\rangle}$$

$$\frac{\cdots \qquad S'(exc) \neq \text{null} \qquad l'' = ExcTable_{\mathfrak{p},l}(\tau(S'(exc))) \neq undef}{[\ldots\ l : \cdots\ \ldots]; \langle S, \sigma, l\rangle \to \langle S'[exc \mapsto \text{null}], (S'(e)), l''\rangle}$$

$$\frac{\cdots \qquad S'(exc) \neq \text{null} \qquad l'' = ExcTable_{\mathfrak{p},l}(\tau(S'(exc))) = undef}{[\ldots\ l : \cdots\ \ldots]; \langle S, \sigma, l\rangle \to \langle S', (), |\mathfrak{p}|\rangle}$$

Observation 5. JVM and $\mathrm{VM_K}$ exception tables are equivalent.

Example 27. The new rules for binary operations are defined as follows. $OpOK_{\text{op}}(v_1, v_2)$ tests whether the operation is valid on the given arguments. for op = (/), this could be

$$(v_1, v_2) \mapsto v_2 \neq 0$$

In case the operation is valid:

$$\frac{OpOK_{\text{op}}(v_1, v_2)}{[\ldots\ l : \texttt{binop}_{\text{op}}\ \ldots]; \langle S, (\sigma, v_1, v_2), l\rangle \to \langle S, (\sigma, v_1 \,\text{op}\, v_2), l+1\rangle}$$

and if the operation is not valid

$$\frac{\begin{array}{c} \neg OpOK_{\text{op}}(v_1, v_2) \\ S_p = OpExc_{\text{op}}(S, v_1, v_2) \\ l_p = ExcTable_{\mathfrak{p},l}(\tau(S_p(exc))) \\ (S', \sigma', l') = \begin{cases} (S_p, (), |\mathfrak{p}|) & \text{if } l_p = undef \\ (S_p[exc \mapsto \text{null}], (S_p(exc)), l_p) & \text{if } l_p \neq undef \end{cases} \end{array}}{[\ldots\ l : \texttt{binop}_{\text{op}}\ \ldots]; \langle S, (\sigma, v_1, v_2), l\rangle \to \langle S', \sigma', l'\rangle}$$

Where $OpExc_{\text{op}}(S, \ldots)$ is a function that adds an exception to S describing the reason why op failed. A simple possibility would be

$$(S, v_1, v_2) \mapsto S[\$ \mapsto S(\$)\langle InvalidOp\rangle, exc \mapsto \text{new}(S(\$), InvalidOp)]$$

It is convenient to use the abbreviation

$$ExcTrans_{\mathfrak{p},l} : S_p \mapsto \begin{cases} (S_p, (), |\mathfrak{p}|) & \text{if } l_p = undef \\ (S_p[exc \mapsto \text{null}], (S_p(exc)), l_p) & \text{if } l_p \neq undef \end{cases}$$

where $l_p = ExcTable_{\mathfrak{p},l}(\tau(S_p(exc)))$

Example 28.

$$\frac{\tau(v) \preceq T}{[\ldots\ l : \texttt{checkcast}\, T\ \ldots]; \langle S, (\sigma, v), l\rangle \to \langle S, (\sigma, v), l+1\rangle}$$

$$\frac{\begin{array}{c}\tau(v) \not\preceq T \\ S_p = InvalidCast(S) \\ (S', \sigma', l') = ExcTrans_{\mathfrak{p},l}(S_p)\end{array}}{[\ldots\ l : \texttt{checkcast}\, T\ \ldots]; \langle S, (\sigma, v), l\rangle \to \langle S', \sigma', l'\rangle}$$

$InvalidCast$ has the same function as $OpExc_{\text{op}}$ in the previous example.

Example 29. The `invokevirtual` rule just propagates any exception that was not handled in the invoked method on y if $y \neq$ null.

$$\frac{\begin{array}{c} y \neq \text{null} \\ \mathfrak{p}' = body_{\text{VM}_\text{K}}(impl(\tau(y), m)) \qquad \mathfrak{p}'(l') = \texttt{end\_method} \\ \mathfrak{p}'; \langle \{\text{this} \mapsto y, \text{p} \mapsto v, \$ \mapsto S(\$)\}, (), 0\rangle \to^* \langle S', \sigma', l'\rangle \\ S_p = S[\$ \mapsto S'(\$), exc \mapsto S'(exc)] \\ \sigma_p = (\sigma, S'(\text{result})) \\ (S'', \sigma'', l'') = \begin{cases} (S_p, \sigma_p, l+1) & \text{if } S_p(exc) = \text{null} \\ ExcTrans_{\mathfrak{p},l}(S_p) & \text{if } S_p(exc) \neq \text{null} \end{cases} \end{array}}{[\ldots\ l : \texttt{invokevirtual}\ T : m\ \ldots]; \langle S, (\sigma, y, v), l\rangle \to \langle S'', \sigma'', l''\rangle}$$

If invoked on null, it throws an exception without executing the method:

$$\frac{\begin{array}{c} y = \text{null} \\ S_p = NullRef(S) \\ (S'', \sigma'', l'') = ExcTrans_{\mathfrak{p},l}(S_p) \end{array}}{[\ldots\ l : \texttt{invokevirtual}\ T : m\ \ldots]; \langle S, (\sigma, y, v), l\rangle \to \langle S'', \sigma'', l''\rangle}$$

6.2.2 The Axiomatic View

We can cope with exceptions in instruction specifications by allowing the special value exc in assertions. Using different specifications depending on whether an exception has occurred or not[23] would require more work.

Exception handling effectively turn each instruction into a branching instruction. The new premises therefore look very much like the ones we already had for `brtrue`. Again in a suggestive notation where $\text{wp}^1_{\mathfrak{p}}$ is the local weakest precondition *as defined earlier*:

$$\frac{\begin{array}{c} E_l \to (NoExcCond \to \text{wp}^1_{\mathfrak{p}}(I_l, (E_i)_{i \in succ(l:I_l)})) \\ \wedge\, Handled_{\mathfrak{p},l} \wedge Unhandled_{\mathfrak{p},l} \end{array}}{\mathcal{A} \vdash \{E_l\}\, l : I_l}$$

where $Handled_{\mathfrak{p},l}$ is defined as

$$\bigwedge_{Exc \in HandledExcs_{\mathfrak{p},l}} (Cond_{\mathfrak{p},l}(Exc) \to raise(Exc, E_{ExcTable_{\mathfrak{p},l}(Exc)}[exc/s(0)]))$$

[23] `signals` in JML

and $Unhandled_{\mathtt{p},l}$ as

$$\bigwedge_{Exc \in UnhandledExcs_{\mathtt{p},l}} (Cond_{\mathtt{p},l}(Exc) \to raise(Exc, E_{|\mathtt{p}|}))$$

- $HandledExcs_{\mathtt{p},l}$ is the set of types for which $ExcTable_{\mathtt{p},l}(.)$ is defined. $UnhandledExcs_{\mathtt{p},l}$ is the set of types for which $ExcTable_{\mathtt{p},l}(.)$ is undefined
- $Cond_{\mathtt{p},l}(Exc)$ is the logical formula describing the reason to raise Exc for program point l in $\mathtt{p}$. For binary operators for instance, this is corresponds to $OpOK_{\mathrm{op}}(s(1), s(0))$ (as defined above)
- $raise(Exc, E)$ models the effect of throwing the exception Exc on a condition E. It is the weakest preconditon for raising the exception Exc. In the simplest version, this is:

$$raise(Exc, E) = E[\mathrm{new}(\$, Exc)/exc, \$\langle Exc\rangle/\,\$]$$

- The precondition of a handler may not reference a stack element other than $s(0)$. This is clear, as they will not be defined, but it is also required to make our exception handling method sound.[24]

Example 30. The throw instruction is easy to handle as its $NoExcCond$ is $false$. It does not however create an exception. It does merely raise the exception. We therefore need a new function $raise_0$ to raise exceptions that have already been created and are found on top of the stack.

$$raise_0(E) = E[s(0)/exc]$$

The rule for `throw` can then be defined as:

$$\text{throw}\ \frac{\begin{array}{c} E_l \to \bigwedge_{Exc \in HandledExcs_{\mathtt{p},l}} (\tau(s(0)) = Exc \to raise_0(E_{ExcTable_{\mathtt{p},l}(Exc)}[exc/s(0)])) \\ \bigwedge_{Exc \in UnhandledExcs_{\mathtt{p},l}} (\tau(s(0)) = Exc \to raise_0(E_{|\mathtt{p}|})) \end{array}}{\mathcal{A} \vdash \{E_l\}\, l : \mathtt{throw}}$$

Example 31. Let's assume that $ExcTable_{\mathtt{p},l}$ is completely undefined. The `invokevirtual` instruction has to propagate any exception unhandled by the callee, it does not *raise* an exception itself:

$$\frac{\begin{array}{c} \mathcal{A} \vdash \{P\}\ T : m\ \{Q\} \\ Z \text{ is a vector of logical variables} \\ w \text{ is a vector of local or a stack elements } \neq s(0) \\ E_l \to (s(1) \neq \mathrm{null} \to P[s(1)/\,\mathrm{this}, s(0)/\,\mathrm{p}][\mathrm{shift}(w)/Z]) \\ \wedge\, (s(1) = \mathrm{null} \to raise(NullReference, E_{|\mathtt{p}|})) \\ Q[s(0)/\,\mathrm{result}][w/Z] \to ((exc = \mathrm{null} \to E_{l+1}) \wedge (exc \neq \mathrm{null} \to E_{|\mathtt{p}|}) \end{array}}{\mathcal{A} \vdash \{E_l\}\, l : \mathtt{invokevirtual}\ T : m}$$

The full version has to do a case distinction on the type of exc.[25]

[24] The precondition of `end_method` may not contain any stack references.

[25] $\bigwedge_{Exc \in HandledExcs_{\mathtt{p},l}} (\tau(exc) = Exc \to raise_0(E_{ExcTable_{\mathtt{p},l}(Exc)}[exc/s(0)])) \wedge \bigwedge_{Exc \in UnhandledExcs_{\mathtt{p},l}} (\tau(exc) = Exc \to raise_0(E_{|\mathtt{p}|}))$ instead of just $E_{|\mathtt{p}|}$. The same applies for `checkcast` T presented for methods without exception handlers.

Example 32. Again assuming that $ExcTable_{\mathtt{p},l}$ is completely undefined, the rule for $\mathtt{checkcast}\,T$ is:

$$\frac{\begin{array}{c}E_l \rightarrow (\tau(s(0)) \preceq T \rightarrow E_{l+1}) \\ \wedge\, (\tau(s(0)) \not\preceq T \rightarrow raise(InvalidCast, E_{|\mathtt{p}|}))\end{array}}{\mathcal{A} \vdash \{E_l\}\, l : I_l}$$

Example 33. When proving programs with exception handling, method postconditions usually are of the form $exc \neq \text{null} \rightarrow Q$ if exceptions are not considered or $(exc = \text{null} \rightarrow Q_{Norm}) \wedge (exc \neq \text{null} \rightarrow Q_{Abrupt})$ if guarantees are given even if the method terminates abruptly. These postconditions correspond to the separate postconditions Q_{Norm} and Q_{Abrupt} in case of normal and abrupt termination resp. Q_{Abrupt} may itself consist of different formulas for different types of exceptions that may be thrown.

7 Related Work

A huge amount of work deals with the formalization of aspects of the JVM. [HM01] contains an overview. [SBS01] gives an almost comprehensive ASM specification of the JVM. Operational semantics for the JVM are given in many different publications. Most of them want to be faithful to the "real" JVM but fail to model non-trivial aspects like dynamic class loading and garbage collection. Examples include [Qia99], [SH01] and [Ber97]. Operational semantics have been used to proof the soundness of type systems for bytecode (e.g., [SNF03]), but they are not very suitable for program verification. [Qui03] describes a formalism that tries to rediscover structures in the bytecode for program verification, precluding the verification of arbitrary programs. There are fewer papers about the CLI, but the results are the same. The formalism for instruction specifications in our logic is based on [Ben04].

8 Conclusion

We presented the operational semantics and a programming logic for the bytecode of $\mathrm{VM_K}$, a virtual machine similar to the JVM or the CLI. Possible uses of the logic include language interoperability of specifications and trusted bytecode components. Because the bytecode logic is based on high-level abstractions and close to existing source logics, it is suitable target for source-to-bytecode proof transformations.

The bytecode logic is a simple and intuitively accessible Hoare style logic with labeled assertions. Labeled assertions allow us to express non-local requirements on instructions. They are conceptually simpler than other methods to handle unstructured control flow. Checking a proof is purely local: we can check one instruction at a time. The proof obligations for instructions are logic formulas. This makes the logic highly suitable for "extended proof carrying code".

We also presented a weakest precondition calculus. Weakest preconditions are not only useful to show completeness but also to support interactive program verification. This is especially true for bytecode where instructions have very simple weakest preconditions.

The logic and its extensions presented in this paper support most features found in the CLI and all of the JVM bytecode languages *directly*. Missing features are reference parameters and delegates. Extending the language to include these does not add any new complications but their exact model is highly application dependent.

References

[Ban04] Fabian Yves Bannwart. A logic for bytecode and the translation of proofs from sequential java. `http://sct.inf.ethz.ch/projects/student_docs/Fabian_Bannwart_paper.pdf`, 2004.

[Ben04] Nick Benton. A typed logic for stacks and jumps. 2004.

[Ber97] P. Bertelsen. Semantics of Java Byte Code. Technical report, 1997.

[Ber00] Rudolf Berghammer. Soundness of a purely syntactical formalization of weakest preconditions. In Dieter Spreen, editor, *Electronic Notes in Theoretical Computer Science*, volume 35. Elsevier, 2000.

[BFGS04] E. Börger, G. Fruja, V. Gervasi, and R. F. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 2004. Accepted for publication.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag New York, Inc., 1990.

[ECM02] *Standard ECMA-335: Common Language Infrastructure*. ECMA International, 2002.

[HM01] Pieter H. Hartel and Luc Moreau. Formalizing the safety of Java, the Java Virtual Machine, and Java Card. *ACM Computing Surveys*, 33(4):517–558, 2001.

[JAR03] Java bytecode verification. *J. of Automated Reasoning*, 30(3–4), 2003.

[Kle99] Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119. ACM Press, 1997.

[PH97] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.

[PHM99] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symposium on Programming (ESOP '99)*, volume 1576, pages 162–176. Springer-Verlag, 1999.

[Qia99] Zhenyu Qian. A formal specification of Java Virtual Machine instructions for objects, methods and subrountines. *Lecture Notes in Computer Science*, pages 271–312, 1999.

[Qui03] Claire L. Quigley. A programming logic for Java bytecode programs. *Lecture Notes in Computer Science*, 2758:41–54, 2003.

[Rau02] Nicole Rauch. Precondition generation for a Java subset. In G. Schellhorn D. Haneberg and W. Reif, editors, *FM-TOOLS 2002, The 5th Workshop on Tools for System Design and Verification, Reisensburg, Germany*, Report 2002-11, pages 1–6. Universität Augsburg, Institut für Informatik, July 2002.

[SBS01] Robert F. Stärk, E. Börger, and Joachim Schmid. *Java and the Java Virtual Machine: Definition, Verification, Validation with Cdrom.* Springer-Verlag New York, Inc., 2001.

[SH01] I. Siveroni and C. Hankin. A proposal for the JCVMLe operational semantics, 2001.

[SNF03] John C. Mitchell Stephen N. Freund. A type system for the Java bytecode language and verifier. *J. of Automated Reasoning*, 30(3–4):271–321, 2003.

[SS03] R. F. Stärk and J. Schmid. Completeness of a bytecode verifier and a certifying Java-to-JVM compiler. *J. of Automated Reasoning*, 30(3–4):323–361, 2003.