A Logic for Bytecode and the Translation of Proofs from Sequential Java

Fabian Yves Bannwart

June 2, 2004

Contents

Listings

Abstract

The goal of this semester-thesis is twofold. Firstly, we are defining an operational semantics and a Hoare-style programming logic for an imaginary sequential, stack-based, bytecode language with local variables, objects and dynamic dispatch that resembles the JVM or the CLR. Secondly, we explain how to translate source-level proofs to the bytecode logic we have introduced in the first part.

1 Introduction

This paper 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 CLR. The instruction set is small to keep the logic simple, but large enough to show all the problems that may occur when trying to specify a more realistic bytecode language for an existing virtual machine. The formalisms for the bytecode logic closely follow [Ben04]. Objects and dynamic dispatch are treated similarly as in [PHM99]. It is not necessary to read these papers. We will repeat all the definitions, rules and conventions necessary to understand our ideas.[1]

Unlike the logic in [Ben04], we do not merge specification and typing information in our bytecode logic. But we naturally require certain well-typedness (see section 3.1.5 on page 14) 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 are complicated enough[2] 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.

1.1 Why is a Bytecode Logic and a Corresponding Translation Procedure useful?

Is it necessary to have a logic for a 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 CLR, 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] Needless to say, what we define here may sometimes differ from the original in [Ben04] and [PHM99]

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

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. No one will be willing to prove programs on the bytecode level and manual intervention is still necessary for formal program verification.

The logic presented here and the accompanying translation procedure for program proofs is meant to be a first step towards a system that can automatically produce formally verified bytecode from a verified input program.

With introduction of the CLR, 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 and its corresponding translation procedures of source level proofs into that logic: To define a common semantics for specifications written in different languages. It can guarantee that correctness properties survive across language boundaries and are even formally accessible.

This text treats the very simple case of a source and target language that are directly comparable.

The sections of this paper are organized as follows: We will introduce you to the source logic for the subset of Java we are considering (taken from [PHM99]). We then define an operational semantics for the $\mathrm{VM_K}$bytecode. The operational semantics *directly* leads to the Hoare-style program logic for bytecode (our target calculus). The proof transformation procedure is explained in the following section and the paper ends with an example of a translation from a source proof to a bytecode proof. There is also a simple but complete implementation of the translation procedure given in section A on page 49.

We do not prove the soundness of our logic with respect to the operational semantics. A proof for a subset of the $\mathrm{VM_K}$ instructions can be found in [Ben04]. The soundness proof of the remaining instructions should be straightforward, because there is a one-to-one correspondence between proof rules for the logic and transition rules of the operational semantics.

We tried to write the paper in a strongly intuitive style. The reader should be familiar however with the basic notions of programming language syntax and semantics.

2 The Logic for Sequential Java

In this section we introduce the Hoare-style logic that we translate from. It is a logic for a sequential Java kernel, called Java-K. We explain some of the concepts that are also used for the bytecode logic, e.g., how references and objects are handled. Only those parts of original paper ([PHM99]) are repeated that we need for the subsequent explanations of the $\mathrm{VM_K}$ logic.

As always the intuitive meaning of the Hoare-triple

$$\{P\} \text{ comp } \{Q\}$$

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 mainly concerned with the specification and verification of individual methods with pre- and postcondition: given some precondition of a method-body, what condition holds when the method terminates. The meaning of Hoare-triples ($\{P\}$ Method body $\{Q\}$) can then be extended to method declaration identifiers that represent a method implementation or a set of method implementations in the case of a virtual method. A statically bound method m in class T has the method identifier $T@m$. A virtual method m in class T is denoted by $T : m$

Example 1. For a statically bound method $T@m$

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

holds if

$$\{P \wedge \text{this} \neq \text{null}\}\ \text{Method body of } T@m\ \{Q\}$$

We write $impl(T@m)$ for the method body of $T@m$

Example 2. For a dynamically bound method $T : m$

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

holds if

$$\{P \wedge \text{this} \neq \text{null}\}\ \text{Method body of } S@m\ \{Q\}$$

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

The state of our programs consists of[3] the local parameters, the local variables and the object store. Local parameters and local variables are treated identically, so we may say local variables if we mean local variables as well as parameters. The *object store* \$ is introduced in order 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 very 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:
  $$\text{iv} : Value \times FieldDeclId \rightarrow InstVar$$
  instance variables are written as $Type@fieldname$

  Example 3. The class T has the fields a, b, c they are referred to as $T@a$, $T@b$, $T@c$, resp.

- instance variable update:
  $$\$\langle f := v\rangle : ObjectStore \times InstVar \times Value \rightarrow ObjectStore$$

- instance variable load:
  $$\$(f) : ObjectStore \times InstVar \rightarrow Value$$

- new object allocation: this function returns the object-store after the allocation of a new object of type T
  $$\$\langle T\rangle : ObjectStore \times ClassTypeId \rightarrow ObjectStore$$

- return a new object of type T
  $$new(\$, T) : ObjectStore \times ClassTypeId \rightarrow Value$$

Before we dive into the rules that enable us to handle method invocations and the object store, we will explain some peculiarities of Java-K and list the standard Hoare-rules that are required in almost every Hoare style logic.

[3] remember that we are considering individual methods only

2.1 Conventions, Restrictions and Limitations of Java-K

The language Java-K follows some conventions and has a number of limitations:

- There is no *abrupt termination*:
  - no `break`
  - no `continue`
  - no exception handling support
  - no `return` statement: the return value of a method has to be assigned to a special variable `result`

  the special variables `this` and `result` are treated just like any other parameter or local variable resp.
- No support for global data. This is not a real restriction as global variables can always be represented as fields of some metaclass objects. This is not practically useful in the restricted context of specifications that consists only of pre- and postconditions of method implementations and is therefore omitted.
- While there can be basically many parameters, we give rules for methods with only one formal parameter `p`. All in all, we usually have two parameters: the customary `this` and `p`.

For an example on how some of these rules are used in a proof, see section 3.2 on page 15.

2.2 Base Rules

There are a few rules common to practically every Hoare-style logic. These rules are taken from figure 1 of [PHM99] and are presented here without comment.

2.2.1 assign

$$\text{assign}\,\frac{}{\{P[e/x]\}\ x = e\ \{P\}}$$

2.2.2 if

$$\text{if}\,\frac{\begin{array}{c}\{e = \top \wedge P\}\ \text{comp}_1\ \{Q\}\\ \{e = \bot \wedge P\}\ \text{comp}_2\ \{Q\}\end{array}}{\{P\}\ \texttt{if}(e)\{\text{comp}_1\}\texttt{else}\{\text{comp}_2\}\ \{Q\}}$$

2.2.3 seq

$$\text{seq}\,\frac{\begin{array}{c}\{P\}\ \text{comp}_1\ \{R\}\\ \{R\}\ \text{comp}_2\ \{Q\}\end{array}}{\{P\}\ \text{comp}_1;\text{comp}_2\ \{Q\}}$$

2.2.4 while

$$\text{while}\,\frac{\{e = \top \land P\}\ \text{comp}\ \{P\}}{\{P\}\ \texttt{while}(e)\{\text{comp}\}\ \{Q\}}$$

2.2.5 conjunct

$$\text{conjunct}\,\frac{\begin{array}{c}\{P_1\}\ \text{comp}\ \{Q_1\}\\ \{P_2\}\ \text{comp}\ \{Q_2\}\end{array}}{\{P_1 \land P_2\}\ \text{comp}\ \{Q_1 \land Q_2\}}$$

2.2.6 disjunct

$$\text{disjunct}\,\frac{\begin{array}{c}\{P_1\}\ \text{comp}\ \{Q_1\}\\ \{P_2\}\ \text{comp}\ \{Q_2\}\end{array}}{\{P_1 \lor P_2\}\ \text{comp}\ \{Q_1 \lor Q_2\}}$$

2.2.7 strength

$$\text{strength}\,\frac{\begin{array}{c}P \Rightarrow P'\\ \{P'\}\ \text{comp}\ \{Q\}\end{array}}{\{P\}\ \text{comp}\ \{Q\}}$$

2.2.8 weak

$$\text{weak}\,\frac{\begin{array}{c}Q' \Rightarrow Q\\ \{P\}\ \text{comp}\ \{Q'\}\end{array}}{\{P\}\ \text{comp}\ \{Q\}}$$

2.2.9 inv

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

2.2.10 subst

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

2.2.11 all

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

2.2.12 ex

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

2.3 Rules for References, Objects and Dynamic Dispatch

We are now going to examine the rules that are necessary to support object-oriented features. We begin with the most straightforward rules and progress to the more sophisticated ones. ($\tau(x)$ denotes the runtime-type of x)

2.3.1 cast

$$\text{cast}\,\frac{}{\{\tau(e) \preceq T \wedge P[e/x]\}\ x = (T)e\ \{P\}}$$

The cast-axiom is similar to the classical assignment axiom, but the stronger precondition ensures that the cast cannot fail.

2.3.2 construction

$$\text{construction}\,\frac{}{\{P[\text{new}(\$, T)/x, \$\langle T\rangle/\,\$]\}\ x = \texttt{new}\,T()\ \{P\}}$$

The construction axiom allocates a new object on the heap and assigns this newly created object to x. Keep in mind that the object-store \$ is the entity that allows us to track the state of the heap syntactically.

2.3.3 fieldread

$$\text{fieldread}\,\frac{}{\{y \neq \text{ null} \wedge P[\$(\text{iv}(y, S@a))/x]\}\ x = y.S@a\ \{P\}}$$

This is a direct adaptation of the assign-axiom.

2.3.4 fieldwrite

$$\text{fieldwrite}\,\frac{}{\{y \neq \text{ null} \wedge P[\$\langle \text{iv}(y, S@a) := e\rangle]\}\ y.S@a = e\ \{P\}}$$

This is a direct adaptation of the assign-axiom as well.

2.3.5 invocation

$$\text{invocation}\,\frac{\{P\}\ T:m\ \{Q\}}{\{y \neq \text{null} \wedge P[y/\text{this}, e/\text{p}]\}\ \ x = y.T:m(e)\ \ \{Q[x/\text{result}]\}}$$

The effect of a method invocation is the same as the effect of the method body expanded at the call site with the formal replaced by the actual parameters and the result-variable in the method replaced by the actual local variable that should receive that result value. Because we do not know which actual method implementation we are calling, we take the Hoare-triple that holds for all bodies (implementations) of $T:m$ (i.e. $\{P\}\ T:m\ \{Q\}$)

2.3.6 invocation-var

$$\text{invocation-var}\,\frac{\begin{array}{c} w \text{ is a program variable} \\ Z \text{ is a logical variable} \\ w \neq x \\ \{P\}\ \ x = y.T:m(e)\ \ \{Q\} \end{array}}{\{P[w/Z]\}\ \ x = y.T:m(e)\ \ \{Q[w/Z]\}}$$

This rule formalizes that local variables (w) are not modified by a function invocation.

2.3.7 implementation

$$\text{implementation}\,\frac{\{\text{this} \neq \text{null} \wedge P\}\ \ \mathit{body}(T@m)\ \ \{Q\}}{\{P\}\ \ T@m\ \ \{Q\}}$$

This rule formalizes what we have said before: The meaning of a function body can be transferred to its signature. For the proof of the body, we can assume that $\{P\}\ T@m\ \{Q\}$ holds (this is necessary for recursive procedures). In [PHM99] there are also formal rules that link the specifications of a virtual method to the actual implementations of that method. They allow statements to be made about virtual method identifiers:

$$\ldots\,\frac{\ldots}{\{P\}\ T:m\ \{Q\}}$$

we do not consider such rules in our text. We just note that it is possible to deduce statements $\{P\}\ T:m\ \{Q\}$ if we know $\{P\}\ S@m\ \{Q\} \forall S \preceq T$

3 The $\mathrm{VM_K}$ bytecode

In this section, we're introducing the design of $\mathrm{VM_K}$ virtual machine bytecode language. 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. It does not impose any limit on the elements that may be pushed onto the stack.

We restrict our attention to the specification and verification of individual method implementations by the means of simple pre- and post-conditions for that method. This is the same abstraction as for Java-K. In

Java-K the method bodies consist of a statement. In $\mathrm{VM_K}$ they consist of sequences of instructions with labels. We therefore have to reason about these sequences.

Because the specification of a program consists only of a pre- and a post-condition of the method, we can use the same abstractions for methods[4] as we have used for Java-K. One of the first goals of this section is therefore to link the two entities we are dealing with: specifications of method identifiers and the conformance of actual implementations to these specifications. I.e., we need a $\mathrm{VM_K}$-rule corresponding to the implementation rule for Java-K:

$$\text{implementation}\frac{\{\text{this} \neq \text{null} \wedge P\}\ \mathit{body}(T@m)\ \{Q\}}{\{P\}\ T@m\ \{Q\}}$$

Unfortunately, we cannot directly adapt this rule from Java-K, because our $\mathrm{VM_K}$ method body consists not of a single computation, but of a *sequence* of *individual* instructions. To be able to give an implementation rule for $\mathrm{VM_K}$, we have to define first what

$$\{\text{this} \neq \text{null} \wedge P\}\ \mathit{body}(T@m)\ \{Q\}$$

means for a $\mathit{body}(T@m)$ being a sequence of bytecode instructions. From now on, we write $\mathit{body}_{\mathrm{VM_K}}(T@m)$ to indicate that we refer to the method body in bytecode form.

As we have jumps in our program, we can easily verify only one bytecode instruction at a time and we say that a sequence $\mathtt{p}$ of instructions is well specified if all of its components are: The following definitions formalize this idea step by step. We define what $\mathrm{VM_K}$ instructions are, how they can be specified, what they do and how they can be assembled to specified instruction sequences. It is then easy to see how the $\mathrm{VM_K}$ implementation rule can be constructed.

Definition 1. A specified $\mathrm{VM_K}$ instruction consists of

1. an instruction

$$\begin{aligned}\text{instr} \in (&\{\mathtt{pushc}\ c, \mathtt{pushvar}\ x, \mathtt{pop}\ x, \mathtt{binop}_{\text{op}}, \mathtt{unop}_{\text{op}}, \mathtt{brtrue}\ l', \mathtt{goto}\ l'\}\\ &\cup \{\mathtt{checkcast}\, T, \mathtt{invokevirtual}\ T:m, \mathtt{getfield}\ T:a, \mathtt{putfield}\ T:a, \mathtt{nop}, \mathtt{end\_method}\})\end{aligned}$$

2. a label $l \in \mathbb{N}$ (unique in the instruction sequence that contains the instruction)
3. a precondition E_l
4. a postcondition E_{l+1}

We write this specified instruction as $\{E_l\}\ l : \text{instr}\ \{E_{l+1}\}$

The meaning of $\{E_l\}\ l : \text{instr}\ \{E_{l+1}\}$ can not been defined in isolation when instr is a control transferring instruction. What we can say however is that if E_l holds when the program counter is just before the instruction (at position l) and instr does not transfer control (instr $\notin \{\mathtt{goto}\ , \mathtt{brtrue}\ l'\}$) then E_{l+1} will hold after the successful execution of instr. We will define the meaning of general instruction specifications together with the definition of full method body specifications.

We can prove (verify) a specified instruction $\{E_l\}\ l : \text{instr}\ \{E_{l+1}\}$ by giving a proof tree for it.

Definition 2. The operational semantics is normative, but this definition gives an *informal* overview of the instructions available in the $\mathrm{VM_K}$.

[4] method identifiers of the form $T : m$ or $T@m$

- `pushc` v pushes a constant v onto the stack
- `pushvar` 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
- $\texttt{unop}_{\text{op}}$ replaces the top element by applying the unary operation op to it
- $\texttt{binop}_{\text{op}}$ replaces the two top elements of the stack by applying the binary operation op to them
- `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` $T : m$ invokes the method $T : m$ on the object reference that is the second-topmost element with the parameter that is the topmost element and replaces these two values by the return value of the invoked method $T : m$
- `getfield` $T@a$ replaces the top element by its field $T@a$
- `putfield` $T@a$ sets the field $T@a$ of the object denoted by the second-topmost element to the top element of the stack and pops both values.
- `nop` has no effect
- `end_method` halts the execution. The intuition is that it transfers control back to the invoking method. It serves therefore as an end marker for a method. There must be exactly one `end_method` instruction per method body at the end of the method.

Definition 3. A specified $\mathrm{VM_K}$ method implementation $\mathfrak{p}$ consists of a sequence of specified $\mathrm{VM_K}$ instructions.

1. $|\mathfrak{p}|$ is the number of instructions in the method
2. The labels of the instructions are in $\Lambda_{\mathfrak{p}} = \{1..|\mathfrak{p}|\}$
3. For every label, we can retrieve the corresponding instruction, pre- and postcondition.

$$\begin{aligned}\mathfrak{p}(l) &= I_l\\ \text{precondition}_{\mathfrak{p}}(l) &= E_l\\ \text{postcondition}_{\mathfrak{p}}(l) &= E_{l+1}\\ \text{spec}_{\mathfrak{p}}(l) &= \{E_l\}\ l : I_l\ \{E_{l+1}\}\end{aligned}$$

4. $\mathfrak{p}, \text{precondition}_{\mathfrak{p}}, \text{postcondition}_{\mathfrak{p}}, \text{spec}_{\mathfrak{p}}$ are *functions*, i.e. there is exactly one instruction, precondition, postcondition for every label in a method body. Putting it differently, labels are *unique* within an instruction sequence.
5. The postcondition of one instruction is the precondition of the next

$$\text{postcondition}_{\mathfrak{p}}(l) = \text{precondition}_{\mathfrak{p}}(l+1)$$

The meaning of $\{E_l\}\ l : \text{instr}\ \{E_{l+1}\}$ in such a instruction sequence is that if 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.

Definition 4. A specified method implementation $\mathtt{p}$ can be verified by verifying all of its components. The precondition for $\mathtt{p}$ is the precondition of the first instruction, the postcondition of $\mathtt{p}$ is the postcondition of the last instruction, which, moreover, is of kind `end_method`. It is the only `end_method` instruction.

$$\text{body}\;\frac{[\forall i \in \Lambda_{\mathtt{p}} : \text{spec}_{\mathtt{p}}(i)] \qquad \begin{array}{r} I_{|\mathtt{p}|} = \texttt{end\_method} \\ \forall i < |\mathtt{p}| : \mathtt{p}(i) \neq \texttt{end\_method} \\ \text{precondition}_{\mathtt{p}}(1) = P \\ \text{postcondition}_{\mathtt{p}}(|\mathtt{p}|) = Q \end{array}}{\{P\}\ \mathtt{p}\ \{Q\}}$$

Definition 5. We can verify the specification of a method $T@m$ by verifying the method implementation $\mathtt{p} = \mathit{body}_{\mathrm{VM_K}}(T@m)$.

$$\mathrm{VM_K}\text{-implementation-0}\;\frac{\{\text{this} \neq \text{null} \wedge P\}\ \mathit{body}_{\mathrm{VM_K}}(T@m)\ \{Q\}}{\{P\}\ T@m\ \{Q\}}$$

And because there is exactly one rule we necessarily have to apply to $\mathit{body}_{\mathrm{VM_K}}(T@m)$ we combine the body and the implementation-0 rule:

$$\mathrm{VM_K}\;\frac{\mathtt{p} = \mathit{body}_{\mathrm{VM_K}}(T@m) \qquad [\forall i \in \Lambda_{\mathtt{p}} : \text{spec}_{\mathtt{p}}(i)] \qquad \begin{array}{r} I_{|\mathtt{p}|} = \texttt{end\_method} \\ \forall i < |\mathtt{p}| : \mathtt{p}(i) \neq \texttt{end\_method} \\ \text{precondition}_{\mathtt{p}}(1) = P \wedge \text{this} \neq \text{null} \\ \text{postcondition}_{\mathtt{p}}(|\mathtt{p}|) = Q \end{array}}{\{P\}\ T@m\ \{Q\}}$$

Remember that proofs in [PHM99] are not only about statements but also about method identifiers. We need this functionality as well for our logic as we are interested in translating Java-K proofs. We will therefore include all the rules of the Java-K logic in the $\mathrm{VM_K}$ logic and leave the proofs for other program entities than method implementations as is. We do however not use the usual Java-K rules for verifying method bodies.

Our translation procedure is then limited to translating method implementation proofs for Java-K to method implementation proofs for $\mathrm{VM_K}$. Parts of the derivation trees outside the method bodies stay the same:

$$\frac{\{\text{this} \neq \text{null} \wedge P\}\ \mathit{body}(T@m)\ \{Q\}}{\{P\}\ T@m\ \{Q\}} \Mapsto \mathrm{VM_K}\,\frac{\ldots}{\{P\}\ T@m\ \{Q\}}$$

There are the same restrictions for our $\mathrm{VM_K}$ as for Java-K: see section 2.1 on page 5.

With these definitions, we have a simple goal: to give a sound and complete system of rules that allow to prove pre- and postconditions for individual instructions. Before we can do that, we should know what our instructions are and what they are supposed to do, i.e., what their effect on the program state is. We will therefore define an operational semantics of all instructions first, and then give rules for proving pre- and postconditions about them.

3.1 Operational Semantics

Definition 6. The configuration $\langle S, \sigma, l\rangle$ of a program during execution consists of the program environment S, the stack σ and the program counter l (the label of the next instruction to be executed).

- The State S maps variables and parameters to values and \$ to the current object store

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

- The stack is a list of values

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

- The program counter is always at a valid position

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

Definition 7. The small step transition

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

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

we also define the multistep relation $\mathtt{p}; \langle S, \sigma, l \rangle \to^* \langle S', \sigma', l' \rangle$ inductively exactly as in [Ben04]:

$$\mathtt{p}; \langle S, \sigma, l \rangle \to^* \langle S, \sigma, l \rangle$$

The $\to^*$ multistep relation frees us from having to model procedure activation frames explicitly.

$$\frac{\mathtt{p}; \langle S, \sigma, l \rangle \to \langle S', \sigma', l' \rangle \qquad \mathtt{p}; \langle S', \sigma', l' \rangle \to^* \langle S'', \sigma'', l'' \rangle}{\mathtt{p}; \langle S, \sigma, l \rangle \to^* \langle S'', \sigma'', l'' \rangle}$$

Equipped with these definitions, we can now define the individual instructions of our virtual machine. See partition II of [ECM02] and [LY99] to compare them with the actual instructions of our example VMs.

Also note that we do not have subroutines (`jsr` in the JVM) although they would be easy to add. We do not need them because our source language does not support exception handling (which is what subroutines are used for in the JVM).

3.1.1 Instructions for the Compilation of Expressions

Pushing a Constant onto the Stack: `pushc` v The instruction `pushc` v is defined by the transition

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

Pushing the Value of a Local Variable onto the Stack: `pushvar` x The instruction `pushvar` x is defined by the transition

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

Popping the Stack into a Local Variable: `pop` x The instruction `pop` x is defined by the transition

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

Unary Operations: $\texttt{unop}_{\text{op}}$ The instruction $\texttt{unop}_{\text{op}}$ is defined by the transition

$$[\ldots\ l : \texttt{unop}_{\text{op}}\ \ldots]; \langle S, (\sigma, v), l\rangle \rightarrow \langle S, (\sigma, \text{op}\, v), l+1\rangle$$

Binary Operations: $\texttt{binop}_{\text{op}}$ The instruction $\texttt{binop}_{\text{op}}$ is defined by the transition

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

3.1.2 Instructions that Modify the Control Flow

Unconditional Jump: `goto` l' The instruction `goto` l' is defined by the transition

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

Conditional Jump: `brtrue` l' The instruction `brtrue` l' is defined by the transitions

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

and

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

3.1.3 Instructions for Objects

Casting a Reference: `checkcast` T The instruction `checkcast` T is defined by the transition

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

Object Creation: `newobj` T The instruction `newobj` T is defined by the transition

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

Calling a Virtual Method (with Exactly one Argument): `invokevirtual` $T : m$ The instruction `invokevirtual` $T : m$ is defined by the transition

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

Loading a Field of an Object: `getfield` $T@a$ The instruction `getfield` $T@a$ is defined by the transition

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

Storing into a Field of an Object: `putfield` $T@a$ The instruction `putfield` $T@a$ is defined by the transition

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

3.1.4 Additional Instructions

No Operation: `nop` The instruction `nop` is defined by the transition

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

3.1.5 Comments

Observation 1. 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 2. There should be constraints on the state at a given program point.

Example 4. The effect of an execution step starting in the configuration $[\ldots\ l : \texttt{pushvar}\ x\ \ldots]; \langle S', \sigma, l\rangle$ cannot be reasonably legitimated if the variable x is not yet initialized.

The configuration $[\ldots\ l : \texttt{binop}_{\text{op}}\ \ldots]; \langle S', (\sigma, a, b), l\rangle$ 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 CLR 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.

3.2 An Example: Calculating x^n Recursively

The sample code has been compiled by the Sun JDK 1.4.2 compiler and has been transcribed for the $\mathrm{VM_K}$.

3.2.1 Java-K

```
package pow;
public class Recursiv{
  public int pow(int x, int n) {
    if(n == 0) {
      result = 1;
    }else {
      if(n%2 == 0) {
        result = this.pow(x*x, n/2);
      }else {
        result = this.pow(x, n-1);
        result = result*x;
      }
    }
  }
}
```

Listing 1: Recursive calculation of x^n in Java-K: This is an example of a complete Java-K program. The Java-K method as well as the corresponding bytecode method implementation (section 3.2.2) are later verified (see section 3.5.1 on page 19 for the Java-K verification).

3.2.2 $\mathrm{VM_K}$

```
             pushc 1
             pop result
test-1:      pushvar n
             unop <>0,
             brtrue path-2
path-1:      pushc 1
             pop result
             goto end
test-2:      pushvar n
             pushc 2
             binop "%"
             unop <>0
```

[5] in a very general sense

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

```
              brtrue path-3
path-2:       pushvar this
              pushvar x
              pushvar x
              binop "*"
              pushvar n
              pushc 2
              binop "/"
              invokevirtual Recursiv:pow
              pop result
              goto end
path-3:       pushvar x
              pushvar this
              pushvar x
              pushvar n
              pushc 1
              binop "-"
              invokevirtual Recursiv:pow
              binop "*"
              pop result
end:          end_method
```

Listing 2: Recursive calculation of x^n in $\mathrm{VM_K}$ bytecode: This the translation of the method body of the corresponding Java-K implementation (section 3.2.1 on the previous page). Both implementations are verified in section 3.5.1 on page 19 and section 3.5.2 on page 20.

3.3 A Programming Logic for the Virtual Machine

In the next section 3.4 we are finally going to present the rules of our logic for the $\mathrm{VM_K}$ bytecode. Having done that, we will verify the specification of the running example `Recursiv@pow`. (section 3.2.2 on the previous page)

Definition 8. The current stack is referred to as s, and its elements are denoted by non-negative integers: element 0 is the top element, etc.:

$$\llbracket s(0)\rrbracket\langle S,(\sigma,v),l\rangle = v$$
$$\llbracket s(i+1)\rrbracket\langle S,(\sigma,v),l\rangle = \llbracket s(i)\rrbracket\langle S,\sigma,l\rangle$$

Definition 9. 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 $\llbracket . \rrbracket$ of assertions E_l are defined as usual. We do not want to restrict the assertions that can be made and we therefore do not define them here (except for the access to stack elements in the previous definition).

Definition 10. Helper functions

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

3.4 The Rules

With the explanations at the beginning of this section (section 3 on page 8) and the operational semantics at hand, we can define the rules for individual instructions.

Observation 3. 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 $K' = \langle S', \sigma', l'\rangle$[7] *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 the precondition of `end_method` holds. This is also the postcondition of our method. Because the `end_method` instruction is the only means of returning from a method[8], we know that if a method terminates, its postcondition will always hold.

3.4.1 Instructions for Expressions

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

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

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

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

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

3.4.2 Instructions that Modify the Control Flow

$$\text{goto}\,\frac{E_l \rightarrow E_{l'}}{\{E_l\}\ l : \texttt{goto}\ l'\ \{E_{l+1}\}}$$

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

3.4.3 Instructions for Objects

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

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

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

[7] which means: $\mathtt{p}; \langle S, \sigma, l\rangle \rightarrow \langle S', \sigma', l'\rangle$

[8] see the definition of the invokevirtual rule

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

for exactly one argument, this is what we are going to reason about

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

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

or more generally for an arbitrary number n of arguments

$$\text{invokevirtual-n}\,\frac{\begin{array}{c}\{P\}\ T : m\ \{Q\}\\ E_l \rightarrow s(1) \neq \text{ null} \wedge P[s(n)/\,\text{this}, \forall i \in 1..n : s(n-i)/\,\text{p}_i] \qquad Q[s(0)/\,\text{result}] \rightarrow E_{l+1}\end{array}}{\{E_l\}\ l : \texttt{invokevirtual}\ T : m\ \{E_{l+1}\}}$$

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

3.4.4 Other Instructions

$$\text{nop}\,\frac{E_l \rightarrow E_{l+1}}{\{E_l\}\ l : \texttt{nop}\ \{E_{l+1}\}}$$

$$\text{method}\,\frac{E_l \rightarrow E_{l+1}}{\{E_l\}\ l : \texttt{end\_method}\ \{E_{l+1}\}}$$

3.5 An Example: Calculating x^n

This section continues section 3.2 on page 15. Here we specify and verify the bytecode method presented there. The specification is:

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

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

We also list a Java-K proof to allow comparison with the proof system we ultimately want to translate from.

3.5.1 Java-K

```
$\{\text{this} \neq \text{null} \wedge x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
$\Longrightarrow$
$\{x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
if(n == 0) {
        $\{n = 0 \wedge x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
        result = 1;
        $\{\text{result} = x_0^{n_0}\}$
}else {
        $\{n \neq 0 \wedge x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
        if(n%2 == 0) {
                $\{n \mod 2 = 0 \wedge n \neq 0 \wedge x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
                $\Longrightarrow$
                $\Downarrow \{x{\cdot}x > 0 \wedge n/2 \geq 0 \wedge x{\cdot}x = x_0{\cdot}x_0 \wedge n/2 = n_0/2 \wedge n_0 \mod 2 = 0\}$ (inv-rule)
                $\Downarrow \{x{\cdot}x > 0 \wedge n/2 \geq 0 \wedge x{\cdot}x = x_0{\cdot}x_0 \wedge n/2 = n_0/2\}$ (subst-rule)
                $\Downarrow \{x{\cdot}x > 0 \wedge n/2 \geq 0 \wedge x{\cdot}x = x_0' \wedge n/2 = n_0'\}$ (subst-rule)
                $\{x{\cdot}x > 0 \wedge n/2 \geq 0 \wedge x{\cdot}x = x_0 \wedge n/2 = n_0\}$
                $\Longrightarrow$
                $\Downarrow \{\text{this} \neq \text{null} \wedge x{\cdot}x > 0 \wedge n/2 \geq 0 \wedge x{\cdot}x = x_0 \wedge n/2 = n_0\}$ (invocation-rule)
                result = this.pow(x*x, n/2);
                $\Uparrow \{\text{result} = x_0^{n_0}\}$ (invocation-rule)
                $\Uparrow \left\{\text{result} = x_0^{\prime n_0'}\right\}$ (subst-rule)
                $\Uparrow \left\{\text{result} = (x_0{\cdot}x_0)^{n_0/2}\right\}$ (subst-rule)
                $\Uparrow \left\{\text{result} = (x_0{\cdot}x_0)^{n_0/2} \wedge n_0 \mod 2 = 0\right\}$ (inv-rule)
                 $\Longrightarrow$
                $\{\text{result} = {x_0}^{n_0}\}$
        }else{
                $\{n \mod 2 \neq 0 \wedge n \neq 0 \wedge x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
                $\Longrightarrow$
                $\{n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
                $\Longrightarrow$
                $\Downarrow \{x > 0 \wedge n - 1 \geq 0 \wedge x = x_0 \wedge n - 1 = n_0 - 1 \wedge x = x_0\}$ (invocation-var-rule)
                $\Downarrow \{x > 0 \wedge n - 1 \geq 0 \wedge x = x_0 \wedge n - 1 = n_0 - 1 \wedge x_f = x_0\}$ (inv-rule)
                $\Downarrow \{x > 0 \wedge n - 1 \geq 0 \wedge x = x_0 \wedge n - 1 = n_0 - 1\}$ (subst-rule)
                $\Downarrow \{x > 0 \wedge n - 1 \geq 0 \wedge x = x_0' \wedge n - 1 = n_0'\}$ (subst-rule)
                $\{x > 0 \wedge n - 1 \geq 0 \wedge x = x_0 \wedge n - 1 = n_0\}$
                $\Longrightarrow$
                $\Downarrow \{\text{this} \neq \text{null} \wedge x > 0 \wedge n - 1 \geq 0 \wedge x = x_0 \wedge n - 1 = n_0\}$ (invocation-rule)
                result = this.pow(x, n-1);
                $\Uparrow \{\text{result} = x_0^{n_0}\}$ (invocation-rule)
                $\Uparrow \left\{\text{result} = x_0^{\prime n_0'}\right\}$ (subst-rule)
                $\Uparrow \left\{\text{result} = {x_0}^{n_0 - 1}\right\}$ (subst-rule)
                $\Uparrow \left\{\text{result} = {x_0}^{n_0 - 1} \wedge x_f = x_0\right\}$ (inv-rule)
                $\Longrightarrow$
                $\{\text{result} \cdot x_f = {x_0}^{n_0} \wedge x_f = x_0\}$
                $\Uparrow \{\text{result} \cdot x = {x_0}^{n_0} \wedge x = x_0\}$ (invocation-var-rule)
                $\Longrightarrow$
                $\Uparrow \{\text{result} \cdot x = {x_0}^{n_0}\}$
                $\Longrightarrow$
                $\{\text{result} \cdot x = x_0^{n_0}\}$
                result = result*x;
                $\{\text{result} = x_0^{n_0}\}$
        }
        $\{\text{result} = x_0^{n_0}\}$
```

}
{result $= x_0^{n_0}$}

Listing 3: Verification of the recursive calculation of x^n: Continuing section 3.2.1 on page 15, we prove the Java-K implementation to allow comparison with the $\mathrm{VM_K}$ proof that follows.

3.5.2 $\mathrm{VM_K}$

Our assumption

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

cannot be used directly for the recursive method invocations. We have to deduce two other triples that are more useful in our calling contexts. As for Java-K, we use a linear notation here that makes it easier to grasp the idea of the proof. 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). See section 5.3 on page 44 for a fully formal bytecode proof that has been automatically generated from a source proof.

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

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

```
              $\{x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
test-1:       pushvar n
              $\{x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0 \wedge (s(0) \neq 0) = (n \neq 0)\}$
              unop <>0,
              $\{x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0 \wedge s(0) = (n \neq 0)\}$
              brtrue test-2
path-1:       $\{n = 0 \wedge x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
```

```
          pushc 1
          $\{s(0) = x_0^{n_0}\}$
          pop result
          $\{\text{result} = x_0^{n_0}\}$
          goto end
test-2:   $\{n \neq 0 \wedge x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
          pushvar n
          $\{((s(0) \mod 2) \neq 0) = (n \mod 2 \neq 0) \wedge n \neq 0 \wedge x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
          pushc 2
          $\{((s(1) \mod s(0)) \neq 0) = (n \mod 2 \neq 0) \wedge n \neq 0 \wedge x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
          binop "%"
          $\{(s(0) \neq 0) = (n \mod 2 \neq 0) \wedge n \neq 0 \wedge x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
          unop <>0
          $\{s(0) = (n \mod 2 \neq 0) \wedge n \neq 0 \wedge x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
          brtrue path-3
path-2:   $\{n \mod 2 = 0 \wedge n \neq 0 \wedge x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
          pushvar this
          $\{n \mod 2 = 0 \wedge n \neq 0 \wedge x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0\}$
          pushvar x
          $\{s(0) = x \wedge n \mod 2 = 0 \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
          pushvar x
          $\{s(1) \cdot s(0) = x \cdot x \wedge \ \wedge n \mod 2 = 0 \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
          binop "*"
          $\{s(0) = x \cdot x \wedge n \mod 2 = 0 \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
          pushvar n
          $\{(s(0)/2) = n/2 \wedge s(1) = x \cdot x \wedge n \mod 2 = 0 \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
          pushc 2
          $\{(s(1)/s(0)) = n/2 \wedge s(2) = x \cdot x \wedge n \mod 2 = 0 \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
          binop "/"
          $\{s(0) = n/2 \wedge s(1) = x \cdot x \wedge n \mod 2 = 0 \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
          $\rightarrow \ \{(x > 0 \wedge n \geq 0 \wedge x = x_0 \cdot x_0 \wedge n = n_0/2 \wedge n_0 \mod 2 = 0)[s(2)/\text{this}, s(1)/x, s(0)/n]\}$
          invokevirtual Recursiv:pow
          $\left\{(\text{result} = (x_0 \cdot x_0)^{(n_0/2)} \wedge n_0 \mod 2 = 0)[s(0)/\text{result}]\right\}$
          $\rightarrow \ \{s(0) = x_0^{n_0}\}$
          pop result
          $\{\text{result} = x_0^{n_0}\}$
          goto end
path-3:   $\{n \mod 2 \neq 0 \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
          pushvar x
          $\{s(0) = x \wedge x > 0 \wedge n > 0 \wedge x = x_0 \wedge n = n_0\}$
          pushvar this
          $\{x = x \wedge s(1) = x \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
          pushvar x
          $\{n - 1 = n - 1 \wedge s(0) = x \wedge s(2) = x \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
          pushvar n
          $\{(s(0) - 1) = n - 1 \wedge s(1) = x \wedge s(3) = x \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
          pushc 1
          $\{(s(1) - s(0)) = n - 1 \wedge s(2) = x \wedge s(4) = x \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
          binop "-"
          $\{s(0) = n - 1 \wedge s(1) = x \wedge s(3) = x \wedge n > 0 \wedge x > 0 \wedge x = x_0 \wedge n = n_0\}$
          $\Downarrow \{s(1) > 0 \wedge s(0) \geq 0 \wedge s(1) = x_0 \wedge s(0) = n_0 - 1 \wedge s(3) = x_0\}$ (invokevar)
          $\{(x > 0 \wedge n \geq 0 \wedge x = x_0 \wedge n = n_0 - 1 \wedge x_f = x_0)[s(2)/\text{this}, s(1)/x, s(0)/n]\}$
          invokevirtual Recursiv:pow
          $\rightarrow \ \{(\text{result} \cdot x_f = {x_0}^{n_0} \wedge x_f = x_0)[s(0)/\text{result}]\}$
          $\Uparrow \{s(0) \cdot s(1) = {x_0}^{n_0} \wedge s(1) = x_0\}$ (invokevar)
          $\rightarrow \ \{s(1) \cdot s(0) = x_0^{n_0}\}$
          binop "*"
```

```
           $\{s(0) = x_0^{n_0}\}$
           pop result
end:       $\{\mathrm{result} = x_0^{n_0}\}$
           end_method
```

Listing 4: Verification of the recursive calculation of x^n: We prove the $\mathrm{VM_K}$ implementation of the example in section 3.2.2 on page 15. The proof should give a rough idea how the programming logic for bytecode can be used in practice to prove method implementations correct.

4 Proof transformation from Source to Bytecode

This section presents the procedure for translating proofs for method specifications from Java-K to $\mathrm{VM_K}$:

$$\frac{\{\mathrm{this} \neq \mathrm{null} \wedge P\}\ \mathit{body}(T@m)\ \{Q\}}{\{P\}\ T@m\ \{Q\}} \mapsto \mathrm{VM_K} \frac{\ldots}{\{P\}\ T@m\ \{Q\}}$$

Before we can do so, we have to fix some conventions and to make a few observations that help construct the bytecode proofs. It will be useful to have an additional notion for expressions that do not contain program variables:

Definition 11. Expressions that do not contain program variables are called Σ-terms.

4.1 Simplifications for the Translation

The translation of proofs presented here is simple. Various improvements are possible. Two important conventions can not be seen from the pattern-based translation rules.

The set of local variables and parameters is not changed during translation. No variables are eliminated, no new variables are introduced. Elimination of local variables that are dead at certain points in the analyzed region is an optimization that would either necessitate a two step translation (compilation and then optimization) or would render our formalism more complicated. Similarly for assertions: we assume that *trivial proof transformation for assertions is possible*: a source proof an assertion P can be used as the proof of the translated assertion P on the bytecode level. Since we do not translate program variables and we do not change logical variables, we do not need to translate assertions. It is sometimes necessary for the correctness of the generated proofs to keep in mind that assertions in the source proof do not contain references to the stack s.

4.2 Form of the Translation Patterns

We use three compilation function for the translation of source proofs:

∇^{S} for the translation of proof trees that prove statements

∇^{E} for the translation of expressions to simplify the notation used in the definition of ∇^{S} (the ∇^{E} function has weaker requirements on its input parameters than ∇^{X})

∇^{X} for the translation of expressions

These functions return sequences of specified instructions that fulfill the basic block property. If control flow enters the sequence at the beginning of the first statement, then control flow will leave the sequence at the end of the last statement. $\bigtriangledown^{\mathrm{S}}$, $\bigtriangledown^{\mathrm{E}}$ and $\bigtriangledown^{\mathrm{X}}$ are defined depending on the shape of their arguments, i.e., the translation is pattern based.

The compilation function for statements $\bigtriangledown^{\mathrm{S}}$ takes proof trees as its argument and its definition depends only on the rule that is used at the root, i.e., $\bigtriangledown^{\mathrm{S}}$ can handle any valid Java-K derivation tree[9] and always returns a sequence whose pre- and postconditions are the same as the pre- and post-conditions of the the root of the argument derivation tree.

Both $\bigtriangledown^{\mathrm{X}}$ and $\bigtriangledown^{\mathrm{S}}$ take three arguments (P, e, Q): they produce a sequence of instructions *calculating e on top of the stack* where P is the precondition of the sequence and Q is its postcondition. Unlike $\bigtriangledown^{\mathrm{S}}$, the translation of expression is defined only for special cases of P and Q: the *shape* of the pre- and post-conditions is constrained, $\bigtriangledown^{\mathrm{X}}$ and $\bigtriangledown^{\mathrm{E}}$ are used only with arguments that obey these constraints.

We abbreviate the instantiation of $\bigtriangledown^{\mathrm{S}}$. We write "$\bigtriangledown^{\mathrm{S}}$(consequent)" and mean "$\bigtriangledown^{\mathrm{S}}$(the whole tree that leads to the consequent)" when it is clear from the context what the proof looks like:

$$\bigtriangledown^{\mathrm{S}}(\{P\}\ S\ \{Q\}) \equiv \bigtriangledown^{\mathrm{S}}\left(\text{rule}\frac{\cdots\{P\}\ S\ \{Q\}\cdots}{\{P'\}\ S\ \{Q'\}}\right)$$

The proof for the method body is assembled from small *pieces* that are concatenated to form larger pieces that form the whole $\mathrm{VM_K}$ proof for the whole method body.

Example 5. Sequential composition is the easiest rule to translate

$$\bigtriangledown^{\mathrm{S}}\left(\text{seq}\frac{\{P\}\ S_1\ \{Q\} \qquad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}\right) =$$

$$\begin{array}{l} S_1 : \{P\} \\ \quad \bigtriangledown^{\mathrm{S}}(\{P\}\ S_1\ \{Q\}) \\ S_2 : \{Q\} \\ \quad \bigtriangledown^{\mathrm{S}}(\{Q\}\ S_2\ \{R\}) \\ \quad \{R\} \end{array}$$

The rule is to be read as follows:

if the root of the derivation tree is an application of the "seq"-rule, the the proof tree must have the form

$$\text{seq}\frac{\dfrac{T_1}{\{P\}\ S_1\ \{Q\}} \qquad \dfrac{T_1}{\{Q\}\ S_2\ \{R\}}}{\{P\}\ S_1; S_2\ \{R\}}$$

which we abbreviate as

$$\text{seq}\frac{\{P\}\ S_1\ \{Q\} \qquad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$$

[9] there are some minor limitations given in section 4.5 on page 26

then the translation of this tree is the concatenation of the two verified instruction sequences obtained from $p_1 \equiv \bigtriangledown^{\mathrm{S}}(\{P\}\ S_1\ \{Q\})$ and $p_2 \equiv \bigtriangledown^{\mathrm{S}}(\{Q\}\ S_2\ \{R\})$ where the triple $\{P\}\ S_1\ \{Q\}$ again stands for the whole tree that leads to this conclusion: $\dfrac{T_1}{\{P\}\ S_1\ \{Q\}}$. Note that the well-formedness condition

$$\begin{aligned}\mathrm{precondition}_{\mathtt{p}}(l) &= E_l\\ \mathrm{postcondition}_{\mathtt{p}}(l) &= E_{l+1} = \mathrm{precondition}_{\mathtt{p}}(l+1)\end{aligned}$$

is respected at the boundary between the two sequences. "S_1 :" and "S_2 :" are symbolic labels. They stand for the actual labels at the beginning of the sequences p_1 and p_2 resp. Logical labels are also introduced for newly introduced instructions (e.g., $A : \mathtt{pop}\ x$ in the translation of the assign rules). These are always *fresh* labels that stand for the actual numerical label within the definition.

Example 6. Consider the translation rule for the weak-rule: $\bigtriangledown^{\mathrm{S}}\left(\text{weak}\dfrac{Q \to Q' \quad \{P\}\ S\ \{Q\}}{\{P\}\ S\ \{Q'\}}\right) =$

$$\begin{aligned}&L : \{P\}\\ &\quad\bigtriangledown^{\mathrm{S}}(\{P\}\ S\ \{Q\})\\ &P : \{Q\}\\ &\quad\mathtt{nop}\\ &\quad\{Q'\}\end{aligned}$$

We're introducing a new instruction nop into our result sequence of verified instruction. The problem is that nop is not verified itself. According to the nop-rule

$$\text{nop}\frac{E_l \to E_{l+1}}{\{E_l\}\ l : \mathtt{nop}\ \{E_{l+1}\}}$$

We have to prove $E_l \to E_{l+1}$ in order to make the generated nop a verified instruction as required for the proof of a method implementation.

This is what the sections labeled "Reasons" are here for: they contain proofs for the antecedents of the $\mathrm{VM_K}$ rules that are used in the translation. They have to be adapted to a specific deductive system for the underlying logic to be mechanically checkable.

4.3 Making the Translation Easier to Understand

We sometimes argue about all rules of the $\mathrm{VM_K}$ logic by referring to two functions f, g that we claim to be distributive with respect to the boolean operations $\wedge$, $\vee$ and commutative with respect to substitution of logical variables by Σ-terms. This is to be understood as follows.

Looking at section 3.4 on page 16, we see that "most", the ordinary rules, with the notable exceptions of invokevirtual and invokevar, have a special and simple form. Their antecedents consist of a logical implication "→" between two program assertions on which we apply a function consisting of substitutions, shifts and adding one simple conjunction.

Example 7. The pushc rule

$$\text{pushc}\frac{\mathrm{shift}(E_l) \to E_{l+1}[v/s(0)]}{\{E_l\}\ l : \mathtt{pushc}\ v\ \{E_{l+1}\}}$$

has an antecedent of the form

$$f(E_l) \to g(E_{l+1})$$

where $f(E) = \mathrm{shift}(E)$ and $g(E) = E[v/s(0)]$

For the ordinary instructions has antecedents $f(E_{l'}) \to g(E_{l''})$ where f and g have the form

$$E \mapsto \text{shift}^k(E)[\cdots] \wedge A$$

For a specific function f:

$$f(E) = \underbrace{\text{shift}^{k_f}(E)[R_f]}_{f^-(E)} \wedge A_f$$

It is now easy to check that the ordinary rules follow this pattern: (v ranges over values, x over local variables)

$$\begin{aligned}
f, g \in \{\, & \text{shift}(.), \\
& (.)[v/s(0)], \\
& (.)[x/s(0)], \\
& (.), \\
& (\text{shift}(.))[s(0)/x], \\
& (.)[(\text{op}\, s(0))/s(0)], \\
& (\text{shift}(.))[(s(1)\,\text{op}\, s(0))/s(1)], \\
& (.) \wedge \neg s(0), \\
& \text{shift}(.), \\
& (.) \wedge s(0), \\
& (.) \wedge \ \tau(s(0)) \preceq T, \\
& (.)[\text{new}(\$, T)/s(0), \$\langle T\rangle/\,\$], \\
& (.)[\$(\text{iv}(s(0), S@a))/s(0)] \wedge s(0) \neq \text{ null}, \\
& (\text{shift}^2(.))[\$\langle \text{iv}(s(1), S@a) := s(0)\rangle/\,\$] \wedge s(1) \neq \text{ null} \\
& \}
\end{aligned}$$

And that these functions are indeed distributive w.r.t the boolean operators $\wedge$, $\vee$.

$$(\text{shift}^k(E_1)[\cdots] \wedge A) \wedge (\text{shift}^k(E_2)[\cdots] \wedge A) = (\text{shift}^k(E_1 \wedge E_2)[\cdots] \wedge A)$$

$$(\text{shift}^k(E_1)[\cdots] \wedge A) \vee (\text{shift}^k(E_2)[\cdots] \wedge A) = (\text{shift}^k(E_1 \vee E_2)[\cdots] \wedge A)$$

invokevirtual and invokevar are less regular, their antecedents' form is $E_l \to X \wedge Y \to E_{l+1}$

4.4 `nops` Generated During Translation

`nop` instructions may be generated during translation. They can removed in a second pass (through the VM_K proof) if needed: all rules have the f, g form explained above. Both before and after an instruction, we replace `nop`'s using the knowledge, that the implication is transitive either

1. "after"[10] another instruction if g is the identity (goto, invokevirtual, invokevar, nop, method)

$$(f(A) \to B \wedge B \to C) \to (f(A) \to C)$$

or

[10] all successor instructions

2. before another instruction if f is the identity (pop, unop, binop, goto, checkcast, getfield, putfield, onvokevirtual, invokevar, nop, method):

$$(A \to B \wedge B \to g(C)) \to (A \to g(C))$$

Example 8. Consider the following sequence from section 5.3 on page 44

$$\left[\dfrac{\dfrac{T_1}{\mathtt{x} = x_0 \to (\mathtt{x} = x_0 \wedge 1 = 1)}}{\{\mathtt{x} = x_0\}\ 1 : \mathtt{nop}\ \{(\mathtt{x} = x_0 \wedge 1 = 1)\}}, \dfrac{\dfrac{T_2}{\mathrm{shift}((\mathtt{x} = x_0 \wedge 1 = 1)) \to (\mathtt{x} = x_0 \wedge 1 = s(0))[1/s(0)]}}{\{(\mathtt{x} = x_0 \wedge 1 = 1)\}\ 2 : \mathtt{pushc}\ 1\ \{(\mathtt{x} = x_0 \wedge 1 = s(0))\}}\right]$$

In this context, it seems that `nop`s cannot be removed: Rule 1 is not applicable because the `nop` has no predecessor and rule 2 is not applicable either (because shift(.) is not the identity). This is not a problem however. shift does only *rename* free variables, which does not change the truth-value of a formula:

$$\left[\dfrac{\dfrac{\dfrac{\dfrac{T_1}{\mathtt{x} = x_0 \to (\mathtt{x} = x_0 \wedge 1 = 1)}}{\mathrm{shift}(\mathtt{x} = x_0) \to \mathrm{shift}((\mathtt{x} = x_0 \wedge 1 = 1))} \quad \dfrac{T_2}{\mathrm{shift}((\mathtt{x} = x_0 \wedge 1 = 1)) \to (\mathtt{x} = x_0 \wedge 1 = s(0))[1/s(0)]}}{\mathrm{shift}(\mathtt{x} = x_0) \to (\mathtt{x} = x_0 \wedge 1 = s(0))[1/s(0)]}}{\{\mathtt{x} = x_0\}\ 2 : \mathtt{pushc}\ 1\ \{(\mathtt{x} = x_0 \wedge 1 = s(0))\}}\right]$$

Taking this special case into account, we can apply the extended rule 2 to `pushvar` and `pushc` as well. This is important: `nop` before `pushvar` and `pushc` are generated for every expression that is translated by ∇^{E}.

Closer examinations of the actual proof-sequences that can be produced show that *all* `nop`*s can be removed* from the proofs we generate by just applying the two rules above when appropriate.

4.5 Limitation: $\exists$ Elimination and $\forall$ Introduction

We are convinced that the ex- and the all-rules are not strictly necessary on the bytecode level. We believe that they can *always* be lifted to the outmost level of method body proofs an then pulled down into the proof for the method identifier ($T@m$). Unfortunately, we do know how this is achieved, in general. For now we just assume that ex- and all-rules do not occur in source proofs for method bodies. Thus source proofs should have a special *normal form.*

Other possibilities of handling the ex- and all-rules are more invasive: they may make the mechanical verification of a proof cumbersome or destroy other desirable properties of our current proof rules like having exactly one rule for every instruction[11].

[11] `invokevirtual` is a special case: easy generalization would allow both rules to be combined into one, but it is technically convenient to keep them separate

4.6 Expressions

4.6.1 How Expressions are Translated

The translation of expressions is special: expressions are atomic entities for source logic, but not on for the $\mathrm{VM_K}$.

$\bigtriangledown^{\mathrm{E}}(P, e, Q \equiv \operatorname{shift} P \wedge e = s(0)) =$

$$\begin{aligned}
L :& \{P\} \\
& \texttt{nop} \\
E :& \{P' \equiv P \wedge e = e\} \\
& \bigtriangledown^{\mathrm{X}}(P', e, Q) \\
& \{Q\}
\end{aligned}$$

Reasons

$L : \texttt{nop} \quad P \rightarrow P \wedge e = e$ is trivially true.

4.6.2 $e_1 \operatorname{op} e_2$

This and the next two sections do the actual work for translating expressions.

$\bigtriangledown^{\mathrm{X}}(Q_s \equiv Q \wedge \operatorname{unshift}(P[e/s(0)]), e \equiv (e_1 \operatorname{op} e_2), Q_e \equiv \operatorname{shift}(Q) \wedge P) =$

$$\begin{aligned}
E_1 :& \{Q_s\} \\
& \bigtriangledown^{\mathrm{X}}(Q_s, e_1, Q_2 \equiv \operatorname{shift}(Q) \wedge P[(s(0) \operatorname{op} e_2)/s(0)]) \\
E_2 :& \{Q_2\} \\
& \bigtriangledown^{\mathrm{X}}\big(Q_2, e_2, Q_3 \equiv \operatorname{shift}^2(Q) \wedge (\operatorname{shift}(P))[(s(1) \operatorname{op} s(0))/s(1)]\big) \\
P :& \{Q_3\} \\
& \texttt{binop}_{\mathrm{op}} \\
& \{Q_e\}
\end{aligned}$$

Reasons: $\texttt{binop}_{\mathrm{op}}$

$$\begin{aligned}
& Q_3 \rightarrow (\operatorname{shift}(Q_e))[(s(1) \operatorname{op} s(0))/s(1)] \\
= & \operatorname{shift}^2(Q) \wedge (\operatorname{shift}(P))[(s(1) \operatorname{op} s(0))/s(1)] \rightarrow (\operatorname{shift}(\operatorname{shift}(Q) \wedge P))[(s(1) \operatorname{op} s(0))/s(1)] \\
= & \operatorname{shift}^2(Q) \wedge (\operatorname{shift}(P))[(s(1) \operatorname{op} s(0))/s(1)] \\
& \rightarrow \operatorname{shift}^2(Q)[(s(1) \operatorname{op} s(0))/s(1)] \wedge (\operatorname{shift}(P))[(s(1) \operatorname{op} s(0))/s(1)] \\
= & \operatorname{shift}^2(Q) \wedge (\operatorname{shift}(P))[(s(1) \operatorname{op} s(0))/s(1)] \rightarrow \operatorname{shift}^2(Q) \wedge (\operatorname{shift}(P))[(s(1) \operatorname{op} s(0))/s(1)]
\end{aligned}$$

4.6.3 Constants

$\nabla^{\mathrm{X}}(Q_s \equiv Q \wedge \text{ unshift}(P[e/s(0)]), e \equiv c, Q_e \equiv \text{ shift}(Q) \wedge P) =$

$$\begin{aligned} L : &\{Q_s\} \\ &\texttt{pushc}\ c \\ &\{Q_e\} \end{aligned}$$

Reasons: `pushc` c

$$\begin{aligned} &\text{shift}(Q_s) \rightarrow Q_e[c/s(0)] \\ =&\text{shift}(Q \wedge \text{ unshift}(P[e/s(0)])) \rightarrow (\text{shift}(Q) \wedge P))[c/s(0)] \\ =&\text{shift}(Q) \wedge \text{ shift}(\text{unshift}(P[e/s(0)])) \rightarrow \text{shift}(Q)[c/s(0)] \wedge P[c/s(0)] \\ =&\text{shift}(Q) \wedge P[e/s(0)] \rightarrow \text{shift}(Q)[c/s(0)] \wedge P[c/s(0)] \\ =&\text{shift}(Q) \wedge P[e/s(0)] \rightarrow \text{shift}(Q) \wedge P[c/s(0)] \end{aligned}$$

4.6.4 Variables

$\nabla^{\mathrm{X}}(Q_s \equiv Q \wedge \text{ unshift}(P[e/s(0)]), e \equiv x, Q_e \equiv \text{ shift}(Q) \wedge P) =$

$$\begin{aligned} L : &\{Q_s\} \\ &\texttt{pushvar}\ x \\ &\{Q_e\} \end{aligned}$$

Reasons: `pushvar` x

$$\begin{aligned} &\text{shift}(Q_s) \rightarrow Q_e[x/s(0)] \\ =&\text{shift}(Q \wedge \text{ unshift}(P[e/s(0)])) \rightarrow (\text{shift}(Q) \wedge P))[x/s(0)] \\ =&\text{shift}(Q) \wedge \text{ shift}(\text{unshift}(P[e/s(0)])) \rightarrow \text{shift}(Q)[x/s(0)] \wedge P[x/s(0)] \\ =&\text{shift}(Q) \wedge P[e/s(0)] \rightarrow \text{shift}(Q)[x/s(0)] \wedge P[x/s(0)] \\ =&\text{shift}(Q) \wedge P[e/s(0)] \rightarrow \text{shift}(Q) \wedge P[x/s(0)] \end{aligned}$$

4.6.5 Example for Expressions

The evaluation of $\nabla^{\mathrm{E}}(P, x{\cdot}y, Q \equiv P \wedge (x{\cdot}y) = s(0))$ yields the partial proof

1. $\dfrac{(P \rightarrow (P \wedge (\mathrm{x{\cdot}y}) = (\mathrm{x{\cdot}y})))}{\{P\}\ 1 : \texttt{nop}\ \{(P \wedge (\mathrm{x{\cdot}y}) = (\mathrm{x{\cdot}y}))\}}$

2. $\dfrac{((\text{shift } (P \wedge (\mathrm{x{\cdot}y}) = (\mathrm{x{\cdot}y}))) \rightarrow ((\text{shift } P) \wedge (\mathrm{x{\cdot}y}) = s(0)[(s(0){\cdot}\mathrm{y})/s(0)])[\mathrm{x}/s(0)])}{\{(P \wedge (\mathrm{x{\cdot}y}) = (\mathrm{x{\cdot}y}))\}\ 2 : \texttt{pushvar x}\ \{((\text{shift } P) \wedge (\mathrm{x{\cdot}y}) = s(0)[(s(0){\cdot}\mathrm{y})/s(0)])\}}$

3. $\dfrac{((\text{shift } ((\text{shift } P) \wedge (\mathrm{x{\cdot}y}) = s(0)[(s(0){\cdot}\mathrm{y})/s(0)])) \rightarrow ((\text{shift } (\text{shift } P)) \wedge (\text{shift } (\mathrm{x{\cdot}y}) = s(0))[(s(1){\cdot}s(0))/s(1)])[\mathrm{y}/s(0)])}{\{((\text{shift } P) \wedge (\mathrm{x{\cdot}y}) = s(0)[(s(0){\cdot}\mathrm{y})/s(0)])\}\ 3 : \texttt{pushvar y}\ \{((\text{shift } (\text{shift } P)) \wedge (\text{shift } (\mathrm{x{\cdot}y}) = s(0))[(s(1){\cdot}s(0))/s(1)])\}}$

4. $\dfrac{(((\text{shift } (\text{shift } P)) \wedge (\text{shift } (\mathrm{x{\cdot}y}) = s(0))[(s(1){\cdot}s(0))/s(1)]) \rightarrow (\text{shift } ((\text{shift } P) \wedge (\mathrm{x{\cdot}y}) = s(0)))[(s(1){\cdot}s(0))/s(1)])}{\{((\text{shift } (\text{shift } P)) \wedge (\text{shift } (\mathrm{x{\cdot}y}) = s(0))[(s(1){\cdot}s(0))/s(1)])\}\ 4 : \texttt{binop}\ {\cdot}\ \{((\text{shift } P) \wedge (\mathrm{x{\cdot}y}) = s(0))\}}$

or, after some simplifications and assuming that P is a source proof assertion:

1. $\dfrac{(P \to (P \wedge (\mathrm{x}\cdot\mathrm{y}) = (\mathrm{x}\cdot\mathrm{y})))}{\{P\}\ 1 : \mathtt{nop}\ \{(P \wedge (\mathrm{x}\cdot\mathrm{y}) = (\mathrm{x}\cdot\mathrm{y}))\}}$

2. $\dfrac{((P \wedge (\mathrm{x}\cdot\mathrm{y}) = (\mathrm{x}\cdot\mathrm{y})) \to (P \wedge (\mathrm{x}\cdot\mathrm{y}) = (\mathrm{x}\cdot\mathrm{y})))}{\{(P \wedge (\mathrm{x}\cdot\mathrm{y}) = (\mathrm{x}\cdot\mathrm{y}))\}\ 2 : \mathtt{pushvar\ x}\ \{(P \wedge (\mathrm{x}\cdot\mathrm{y}) = (s(0)\cdot\mathrm{y}))\}}$

3. $\dfrac{((P \wedge (\mathrm{x}\cdot\mathrm{y}) = (s(1)\cdot\mathrm{y})) \to (P \wedge (\mathrm{x}\cdot\mathrm{y}) = (s(1)\cdot\mathrm{y})))}{\{(P \wedge (\mathrm{x}\cdot\mathrm{y}) = (s(0)\cdot\mathrm{y}))\}\ 3 : \mathtt{pushvar\ y}\ \{(P \wedge (\mathrm{x}\cdot\mathrm{y}) = (s(1)\cdot s(0)))\}}$

4. $\dfrac{((P \wedge (\mathrm{x}\cdot\mathrm{y}) = (s(1)\cdot s(0))) \to (P \wedge (\mathrm{x}\cdot\mathrm{y}) = (s(1)\cdot s(0))))}{\{(P \wedge (\mathrm{x}\cdot\mathrm{y}) = (s(1)\cdot s(0)))\}\ 4 : \mathtt{binop}\cdot\ \{(P \wedge (\mathrm{x}\cdot\mathrm{y}) = s(0))\}}$

4.7 Base Rules

4.7.1 assign

$$\nabla^{\mathrm{S}}\left(\text{assign}\,\frac{}{\{P[e/x]\}\ \ x = e\ \ \{P\}}\right) =$$

$$\begin{aligned}
V : &\{P[e/x]\} \\
&\quad \bigtriangledown^{\mathrm{E}} (P[e/x], e, \mathrm{shift}(P[e/x]) \wedge s(0) = e) \\
A : &\{\mathrm{shift}(P[e/x]) \wedge s(0) = e\} \\
&\quad \mathtt{pop}\ x \\
&\quad \{P\}
\end{aligned}$$

Reasons

$A : \mathtt{pop}\ x$

P is a source assertion, it does not contain references to the stack

$$\underbrace{\mathrm{shift}(P[e/x])}_{P[e/x]} \wedge s(0) = e \to (\underbrace{\mathrm{shift}(P)}_{P})[s(0)/x]$$

$$= P[s(0)/x] \wedge s(0) = e \to P[s(0)/x]$$

4.7.2 while

$$\bigtriangledown^{\mathrm{S}}\left(\text{while}\frac{\{e=\top\wedge P\}\ S\ \{P\}}{\{P\}\ \texttt{while}(e)\{S\}\ \{e=\bot\wedge P\}}\right)=$$

$$\begin{array}{l}
L:\{P\}\\
\quad\texttt{goto}\ T\\
S:\{e=\top\wedge P\}\\
\quad\bigtriangledown^{\mathrm{S}}(\{e=\top\wedge P\}\ S\ \{P\})\\
T:\{P\}\\
\quad\bigtriangledown^{\mathrm{E}}(P,e,\mathrm{shift}(P)\wedge s(0)=e)\\
B:\{\mathrm{shift}(P)\wedge s(0)=e\}\\
\quad\texttt{brtrue}\ S\\
\quad\{P\wedge e=\bot\}
\end{array}$$

Reasons

$L:\texttt{goto}\ T$ We have to prove: $P\rightarrow P$, trivial

$B:\texttt{brtrue}\ S$ We have to prove:

$$\mathrm{shift}(P)\wedge s(0)=e\wedge\neg s(0)\rightarrow\mathrm{shift}(P\wedge e=\bot)$$

trivial, and

$$\mathrm{shift}(P)\wedge s(0)=e\wedge s(0)\rightarrow\mathrm{shift}(e=\top\wedge P)$$

also trivial

4.7.3 if

In the produced code, the else-part comes before the then-part of the statement.

$$\nabla^{\mathrm{S}}\left(\text{if}\frac{\{e=\top\wedge P\}\ S_1\ \{Q\}\qquad\{e=\bot\wedge P\}\ S_2\ \{Q\}}{\{P\}\ \texttt{if}(e)\{S_1\}\texttt{else}\{S_2\}\ \{Q\}}\right)=$$

$$\begin{array}{rl}
T: & \{P\}\\
 & \quad\nabla^{\mathrm{E}}(P,e,\mathrm{shift}(P)\wedge s(0)=e)\\
B: & \{\mathrm{shift}(P)\wedge s(0)=e\}\\
 & \texttt{brtrue}\ S_1\\
S_2: & \{e=\bot\wedge P\}\\
 & \quad\nabla^{\mathrm{S}}(\{e=\bot\wedge P\}\ S_2\ \{Q\})\\
L: & \{Q\}\\
 & \texttt{goto}\ E\\
S_1: & \{e=\top\wedge P\}\\
 & \quad\nabla^{\mathrm{S}}(\{e=\top\wedge P\}\ S_1\ \{Q\})\\
E: & \{Q\}\\
 & \texttt{nop}\\
 & \{Q\}
\end{array}$$

Reasons

$B:\texttt{brtrue}\ S_1$ We have to prove:

$$\mathrm{shift}(P)\wedge s(0)=e\wedge\neg s(0)\rightarrow\mathrm{shift}(e=\bot\wedge P)$$

trivial, and

$$\mathrm{shift}(P)\wedge s(0)=e\wedge s(0)\rightarrow\mathrm{shift}(e=\top\wedge P)$$

also trivial

4.7.4 seq

$$\nabla^{\mathrm{S}}\left(\text{seq}\frac{\{P\}\ S_1\ \{Q\}\qquad\{Q\}\ S_2\ \{R\}}{\{P\}\ S_1;S_2\ \{R\}}\right)=$$

$$\begin{array}{rl}
S_1: & \{P\}\\
 & \quad\nabla^{\mathrm{S}}(\{P\}\ S_1\ \{Q\})\\
S_2: & \{Q\}\\
 & \quad\nabla^{\mathrm{S}}(\{Q\}\ S_2\ \{R\})\\
 & \{R\}
\end{array}$$

4.7.5 conjunct/disjunct

Conjunction and disjunction are treated identically. We show here only the conjunct-rule.

$$\nabla^{\mathrm{S}}\left(\text{conj}\,\frac{\{P_1\}\ S\ \{Q_1\}\qquad\{P_2\}\ S\ \{Q_2\}}{\{P_1\wedge P_2\}\ S\ \{Q_1\wedge Q_2\}}\right)=$$

create the two proofs

$$\nabla^{\mathrm{S}}\left(\{P_1\}\ S\ \{Q_1\}\right)$$

and

$$\nabla^{\mathrm{S}}\left(\{P_2\}\ S\ \{Q_2\}\right)$$

The embedded instructions are (by construction) the same. With the two proofs, we assemble a third proof (the result)

$$\nabla^{\mathrm{S}}\left(\{P_1\wedge P_2\}\ S\ \{Q_1\wedge Q_2\}\right)$$

by merging, for all instructions, their specifications

$$\left\{A_{(l)}\right\}\ l:\text{instr}\ \left\{A_{(l+1)}\right\}$$

and

$$\left\{B_{(l)}\right\}\ l:\text{instr}\ \left\{B_{(l+1)}\right\}$$

to get

$$\left\{A_{(l)}\wedge B_{(l)}\right\}\ l:\text{instr}\ \left\{A_{(l+1)}\wedge B_{(l+1)}\right\}$$

For the common instructions with that have the f, g form (see section 4.3 on page 24), a proof tree can then be constructed easily:

$$\frac{\text{distributive}\,\dfrac{\dfrac{\dfrac{\cdots}{f(A)\to g(B)}\qquad\dfrac{\cdots}{f(C)\to g(D)}}{f(A)\wedge f(C)\to g(B)\wedge g(D)}}{\cdots f(A\wedge C)\to g(B\wedge D)\cdots}}{\left\{A_{(l)}\wedge B_{(l)}\right\}\ l:\text{instr}\ \left\{A_{(l+1)}\wedge B(l+1)\right\}}$$

This argument is not true for `invokevirtual` (i.e. for the invokevirtual and invokevar rules). In that case, we apply the the rule directly to the method specification that is required by `invokevirtual` if the two rules we're looking at are both invokevirtual:

$$\frac{\dfrac{\dfrac{\cdots}{\{P_1\}\ T:m\ \{Q_1\}}\qquad\dfrac{\cdots}{\{P_2\}\ T:m\ \{Q_2\}}}{\{P_1\wedge P_2\}\ T:m\ \{Q_1\wedge Q_2\}}\qquad\cdots\quad\text{Others are proven as above}}{\left\{A_{(l)}\wedge B_{(l)}\right\}\ l:\texttt{invokevirtual}\ T:m\ \left\{A_{(l+1)}\wedge B_{(l+1)}\right\}}$$

In the case of at least one invokevar rule[12], we propagate the conjunction up the proof tree. As a simplification, we assume that the logical variables used in the two proofs are disjoint (if this is not the case, we could always insert subst-rules)

given one tree

$$\text{invokevar}\,\frac{\{A'_l\}\ l:\texttt{invokevirtual}\ T:m\ \{A'_{l+1}\}\qquad A_l\to A'_l[(\text{shift}(w))/Z]\qquad A'_{l+1}[w/Z]\to A_{l+1}}{\{A_l\}\ l:\texttt{invokevirtual}\ T:m\ \{A_{l+1}\}}$$

and a second tree (represented by its root)

$$\{B_l\}\ l:\texttt{invokevirtual}\ T:m\ \{B_{l+1}\}$$

[12] we have to "merge" one invocation-var rule with another rule (either a invocation-var or a invokevirtual rule)

we produce

$$\text{invokevar}\,\frac{\dfrac{\dots}{\{A'_l \wedge B_l\}\ l:\texttt{invokevirtual}\ T:m\ \left\{A'_{l+1}\wedge B_{l+1}\right\}} \qquad \begin{array}{c}(A_l\wedge B_l)\to(A'_l\wedge B_l)[(\text{shift}(w))/Z] \\ (A'_{l+1}\wedge B_{l+1})[w/Z]\to(A_{l+1}\wedge B_{l+1})\end{array}}{\{A_l\wedge B_l\}\ l:\texttt{invokevirtual}\ T:m\ \{A_{l+1}\wedge B_{l+1}\}}$$

where the rest of the tree (represented by dots) is obtained from merging the two trees

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

and

$$\{B_l\}\ l:\texttt{invokevirtual}\ T:m\ \{B_{l+1}\}$$

Because we know that B_l and B_{l+1} do not contain Z, we know (i.e. we can prove) that the result tree is a valid application of the invokevar rule. Illustration:

$$(A_l\wedge B_l)\to(A'_l\wedge B_l)[(\text{shift}(w))/Z] = (A_l\wedge B_l)\to A'_l[(\text{shift}(w))/Z]\wedge B_l$$

Example 9. Consider the following method fragment:

```
x = x^2;
```

and two proofs for it

$\{x{\cdot}x = y\}$
`x = x*x`
$\{x = y\}$

$\{x{\cdot}x > 0\}$
`x = x*x`
$\{x > 0\}$

together, applying the conjunct rule, they tell us that

$\{x{\cdot}x = y \wedge x{\cdot}x > 0\}$
`x = x*x`
$\{x = y \wedge x > 0\}$

or representing the proof as a tree

$$\frac{\dfrac{}{\{(\mathtt{x{\cdot}x}) = \mathtt{y}\}\ x = (\mathtt{x{\cdot}x})\ \{\mathtt{x} = \mathtt{y}\}} \qquad \dfrac{}{\{((\mathtt{x{\cdot}x}) > 0)\}\ x = (\mathtt{x{\cdot}x})\ \{(\mathtt{x} > 0)\}}}{\{((\mathtt{x{\cdot}x}) = \mathtt{y} \wedge ((\mathtt{x{\cdot}x}) > 0))\}\ x = (\mathtt{x{\cdot}x})\ \{(\mathtt{x} = \mathtt{y} \wedge (\mathtt{x} > 0))\}}$$

Applying the ∇^{S} function on this tree gives us:

1. $\dfrac{(((\mathtt{x{\cdot}x}) = \mathtt{y} \wedge ((\mathtt{x{\cdot}x}) > 0)) \to (((\mathtt{x{\cdot}x}) = \mathtt{y} \wedge (\mathtt{x{\cdot}x}) = (\mathtt{x{\cdot}x})) \wedge (((\mathtt{x{\cdot}x}) > 0) \wedge (\mathtt{x{\cdot}x}) = (\mathtt{x{\cdot}x}))))}{\{((\mathtt{x{\cdot}x}) = \mathtt{y} \wedge ((\mathtt{x{\cdot}x}) > 0))\}\ 1 : \mathtt{nop}\ \{(((\mathtt{x{\cdot}x}) = \mathtt{y} \wedge (\mathtt{x{\cdot}x}) = (\mathtt{x{\cdot}x})) \wedge (((\mathtt{x{\cdot}x}) > 0) \wedge (\mathtt{x{\cdot}x}) = (\mathtt{x{\cdot}x})))\}}$

2. $\dfrac{((((\mathtt{x{\cdot}x}) = \mathtt{y} \wedge (\mathtt{x{\cdot}x}) = (\mathtt{x{\cdot}x})) \wedge (((\mathtt{x{\cdot}x}) > 0) \wedge (\mathtt{x{\cdot}x}) = (\mathtt{x{\cdot}x}))) \to (((\mathtt{x{\cdot}x}) = \mathtt{y} \wedge (\mathtt{x{\cdot}x}) = (\mathtt{x{\cdot}x})) \wedge (((\mathtt{x{\cdot}x}) > 0) \wedge (\mathtt{x{\cdot}x}) = (\mathtt{x{\cdot}x}))))}{\{(((\mathtt{x{\cdot}x}) = \mathtt{y} \wedge (\mathtt{x{\cdot}x}) = (\mathtt{x{\cdot}x})) \wedge (((\mathtt{x{\cdot}x}) > 0) \wedge (\mathtt{x{\cdot}x}) = (\mathtt{x{\cdot}x})))\}\ 2 : \mathtt{pushvar\ x}\ \{(((\mathtt{x{\cdot}x}) = \mathtt{y} \wedge (\mathtt{x{\cdot}x}) = (s(0){\cdot}\mathtt{x})) \wedge (((\mathtt{x{\cdot}x}) > 0) \wedge (\mathtt{x{\cdot}x}) = (s(0){\cdot}\mathtt{x})))\}}$

3. $$\frac{((((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(1)\cdot\mathtt{x})) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(1)\cdot\mathtt{x}))) \rightarrow (((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(1)\cdot\mathtt{x})) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(1)\cdot\mathtt{x}))))}{\{(((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(0)\cdot\mathtt{x})) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(0)\cdot\mathtt{x})))\}\ 3 : \mathtt{pushvar\ x}\ \{(((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(1)\cdot s(0))) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(1)\cdot s(0))))\}}$$

4. $$\frac{((((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(1)\cdot s(0))) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(1)\cdot s(0)))) \rightarrow (((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(1)\cdot s(0))) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(1)\cdot s(0)))))}{\{(((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(1)\cdot s(0))) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(1)\cdot s(0))))\}\ 4 : \mathtt{binop}\cdot\ \{(((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = s(0)) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = s(0)))\}}$$

5. $$\frac{((((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = s(0)) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = s(0))) \rightarrow (\mathtt{x} = \mathtt{y} \wedge (\mathtt{x} > 0)))}{\{(((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = s(0)) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = s(0)))\}\ 5 : \mathtt{pop\ x}\ \{(\mathtt{x} = \mathtt{y} \wedge (\mathtt{x} > 0))\}}$$

or, in linear form

1. nop

   $\{((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge ((\mathtt{x}\cdot\mathtt{x}) > 0))\}$

   $1 : \mathtt{nop}$

   $\{(((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = (\mathtt{x}\cdot\mathtt{x})) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = (\mathtt{x}\cdot\mathtt{x})))\}$

2. pushvar

   $\{(((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = (\mathtt{x}\cdot\mathtt{x})) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = (\mathtt{x}\cdot\mathtt{x})))\}$

   $2 : \mathtt{pushvar\ x}$

   $\{(((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(0)\cdot\mathtt{x})) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(0)\cdot\mathtt{x})))\}$

3. pushvar

   $\{(((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(0)\cdot\mathtt{x})) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(0)\cdot\mathtt{x})))\}$

   $3 : \mathtt{pushvar\ x}$

   $\{(((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(1)\cdot s(0))) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(1)\cdot s(0))))\}$

4. binop

   $\{(((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(1)\cdot s(0))) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = (s(1)\cdot s(0))))\}$

   $4 : \mathtt{binop}\cdot$

   $\{(((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = s(0)) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = s(0)))\}$

5. pop

   $\{(((\mathtt{x}\cdot\mathtt{x}) = \mathtt{y} \wedge (\mathtt{x}\cdot\mathtt{x}) = s(0)) \wedge (((\mathtt{x}\cdot\mathtt{x}) > 0) \wedge (\mathtt{x}\cdot\mathtt{x}) = s(0)))\}$

   $5 : \mathtt{pop\ x}$

   $\{(\mathtt{x} = \mathtt{y} \wedge (\mathtt{x} > 0))\}$

4.7.6 strength

$$\nabla^{\mathrm{S}}\left(\text{strength}\frac{P' \rightarrow P \qquad \{P\}\ S\ \{Q\}}{\{P'\}\ S\ \{Q\}}\right) =$$

$$\begin{array}{l} P : \{P'\} \\ \quad \mathtt{nop} \\ L : \{P\} \\ \quad \nabla^{\mathrm{S}}\left(\{P\}\ S\ \{Q\}\right) \\ \quad \{Q\} \end{array}$$

Reasons

$P:$ `nop`

$$P' \to P$$

but this we know from the source proof. (see section 4 on page 22)

4.7.7 weak

$$\nabla^{\mathrm{S}}\left(\text{weak}\frac{Q \to Q' \qquad \{P\}\ S\ \{Q\}}{\{P\}\ S\ \{Q'\}}\right) =$$

$$\begin{array}{ll}
L: & \{P\} \\
 & \nabla^{\mathrm{S}}\,(\{P\}\ S\ \{Q\}) \\
P: & \{Q\} \\
 & \texttt{nop} \\
 & \{Q'\}
\end{array}$$

Reasons

$P:$ `nop` We have to prove

$$Q \to Q'$$

but this we know from the source proof. (see section 4 on page 22)

4.7.8 inv

$$\nabla^{\mathrm{S}}\left(\text{inv}\frac{\{P\}\ S\ \{Q\}}{\{P \wedge W\}\ S\ \{Q \wedge W\}}\right) =$$

We just add a $\wedge W$ to every specification of the sequence produced by $\nabla^{\mathrm{S}}\,(\{P\}\ S\ \{Q\})$. We again exploit the special form of the rules.

$$\dfrac{\text{distributive}\dfrac{\dfrac{\dfrac{\dfrac{f(A) \to g(B)}{f(A) \wedge W \to g(B) \wedge W}}{(f^{-}(A) \wedge A_f) \wedge (W \wedge A_f) \to (g^{-}(B) \wedge A_g) \wedge (W \wedge A_g)}}{f(A) \wedge f(W) \to g(B) \wedge g(W)}}{\cdots f(A \wedge W) \to g(B \wedge W) \cdots}}{\left\{E_{(l)} \wedge W\right\}\ l : \text{instr}\ \left\{E_{(l+1)} \wedge W\right\}}$$

`invokevirtual` is treated as in section 4.7.5 on page 31. We propagate the $\wedge W$ along the slim proof tree until we reach the method. We then apply the the original Java-K inv rule to the method.

Example 10. Consider this simple proof

$\{x{\cdot}x > 0\}$
`x = x*x`
$\{x > 0\}$

again.

Applying the inv rule with $W \equiv (X = Y)$ yields

$\Downarrow \{x{\cdot}x > 0 \wedge X = Y\}$
$\{x{\cdot}x > 0\}$
`x = x*x`
$\{x > 0\}$
$\Uparrow \{x > 0 \wedge X = Y\}$

and its proof tree

$$\frac{\frac{}{\{((\mathtt{x}{\cdot}\mathtt{x}) > 0)\}\ \ x = (\mathtt{x}{\cdot}\mathtt{x})\ \ \{(\mathtt{x} > 0)\}}}{\{(((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge X = Y)\}\ \ x = (\mathtt{x}{\cdot}\mathtt{x})\ \ \{((\mathtt{x} > 0) \wedge X = Y)\}}$$

Applying the ∇^{S} function on this tree gives us what we expected:

1. $\dfrac{((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge X = Y) \to ((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge (\mathtt{x}{\cdot}\mathtt{x}) = (\mathtt{x}{\cdot}\mathtt{x})) \wedge X = Y))}{\{(((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge X = Y)\}\ 1 : \mathtt{nop}\ \{((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge (\mathtt{x}{\cdot}\mathtt{x}) = (\mathtt{x}{\cdot}\mathtt{x})) \wedge X = Y)\}}$
2. $\dfrac{(((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge (\mathtt{x}{\cdot}\mathtt{x}) = (\mathtt{x}{\cdot}\mathtt{x})) \wedge X = Y) \to ((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge (\mathtt{x}{\cdot}\mathtt{x}) = (\mathtt{x}{\cdot}\mathtt{x})) \wedge X = Y))}{\{((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge (\mathtt{x}{\cdot}\mathtt{x}) = (\mathtt{x}{\cdot}\mathtt{x})) \wedge X = Y)\}\ 2 : \mathtt{pushvar\ x}\ \{((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge (\mathtt{x}{\cdot}\mathtt{x}) = (s(0){\cdot}\mathtt{x})) \wedge X = Y)\}}$
3. $\dfrac{(((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge (\mathtt{x}{\cdot}\mathtt{x}) = (s(1){\cdot}\mathtt{x})) \wedge X = Y) \to ((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge (\mathtt{x}{\cdot}\mathtt{x}) = (s(1){\cdot}\mathtt{x})) \wedge X = Y))}{\{((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge (\mathtt{x}{\cdot}\mathtt{x}) = (s(0){\cdot}\mathtt{x})) \wedge X = Y)\}\ 3 : \mathtt{pushvar\ x}\ \{((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge (\mathtt{x}{\cdot}\mathtt{x}) = (s(1){\cdot}s(0))) \wedge X = Y)\}}$
4. $\dfrac{(((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge (\mathtt{x}{\cdot}\mathtt{x}) = (s(1){\cdot}s(0))) \wedge X = Y) \to ((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge (\mathtt{x}{\cdot}\mathtt{x}) = (s(1){\cdot}s(0))) \wedge X = Y))}{\{((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge (\mathtt{x}{\cdot}\mathtt{x}) = (s(1){\cdot}s(0))) \wedge X = Y)\}\ 4 : \mathtt{binop}\cdot\ \{((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge (\mathtt{x}{\cdot}\mathtt{x}) = s(0)) \wedge X = Y)\}}$
5. $\dfrac{(((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge (\mathtt{x}{\cdot}\mathtt{x}) = s(0)) \wedge X = Y) \to ((\mathtt{x} > 0) \wedge X = Y))}{\{((((\mathtt{x}{\cdot}\mathtt{x}) > 0) \wedge (\mathtt{x}{\cdot}\mathtt{x}) = s(0)) \wedge X = Y)\}\ 5 : \mathtt{pop\ x}\ \{((\mathtt{x} > 0) \wedge X = Y)\}}$

4.7.9 subst

$\nabla^{\mathrm{S}}\left(\text{subst}\dfrac{\{P\}\ S\ \{Q\}}{\{P[t/Z]\}\ S\ \{Q[t/Z]\}}\right) =$ As before, first we generate $\nabla^{\mathrm{S}}(\{P\}\ S\ \{Q\})$ and then we replace Z by t in each specification and in all proofs for assertions (the antecedents normally). We again use the special form of our rules. We do not really need to do the replacement in all proof, we can actually replace the proofs *compositionally*:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\cdots}}{f(A) \to g(B)}}{(f(A) \to g(B))[t/Z]}}{\cdots f(A[t/Z]) \to g(B[t/Z]) \cdots}}{\left\{E^{(l)}[t/Z]\right\}\ l : \text{instr}\ \left\{E^{(l+1)}[t/Z]\right\}}$$

`invokevirtual` is treated as in section 4.7.8 on the preceding page.

4.8 Rules for References, Objects and Dynamic Dispatch

4.8.1 invocation

$$\nabla^{\mathrm{S}}\left(\text{invocation}\frac{\{P\}\ T:m\ \{Q\}}{\{y \neq \text{ null } \wedge P[y/\text{this}, e/\text{p}]\}\ x = y.T:m(e)\ \{Q[x/\text{result}]\}}\right) =$$

$$\begin{aligned}
&S:\{Q_s \equiv y \neq \text{ null } \wedge P[y/\text{this}, e/\text{p}]\}\\
&\quad \texttt{pushvar}\ y\\
&V:\{\text{shift}(Q_s) \wedge s(0) = y\}\\
&\quad \bigtriangledown^{\mathrm{E}}(\text{shift}(Q_s) \wedge s(0) = y, e, Q_i \equiv \text{ shift}(\text{shift}(Q_s) \wedge s(0) = y) \wedge s(0) = e)\\
&I:\{Q_i\}\\
&\quad \texttt{invokevirtual}\ T:m\\
&A:\{Q[s(0)/\text{result}]\}\\
&\quad \texttt{pop}\ x\\
&\quad \{Q[x/\text{result}]\}
\end{aligned}$$

Reasons

$A:\texttt{pop}\ x$ $\quad Q$ does not contain program variables or stack references

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

$S:\texttt{pushvar}\ y$

$$\text{shift}\, Q_s \rightarrow (\text{shift}(Q_s) \wedge s(0) = y)[x/s(0)]$$

$I:\texttt{invokevirtual}\ T:m$

$$\begin{aligned}
&Q_i\\
&\ \rightarrow s(1) \neq \text{ null } \wedge P[s(1)/\text{this}, s(0)/\text{p}]\\
=&\,\text{shift}(\text{shift}(Q_s) \wedge s(0) = y) \wedge s(0) = e\\
&\ \rightarrow s(1) \neq \text{ null } \wedge P[s(1)/\text{this}, s(0)/\text{p}]\\
=&Q_s \wedge s(1) = y \wedge s(0) = e\\
&\ \rightarrow s(1) \neq \text{ null } \wedge P[s(1)/\text{this}, s(0)/\text{p}]\\
=&y \neq \text{ null } \wedge P[y/\text{this}, e/\text{p}] \wedge s(1) = y \wedge s(0) = e\\
&\ \rightarrow s(1) \neq \text{ null } \wedge P[s(1)/\text{this}, s(0)/\text{p}]
\end{aligned}$$

4.8.2 invocation-var

$$\nabla^{\mathrm{S}}\left(\text{invocation-var}\frac{\{P\}\ x = y.T:m(e)\ \{Q\}}{\{P[w/Z]\}\ x = y.T:m(e)\ \{Q[w/Z]\}}\right) =$$

generate the proof $\nabla^{\mathrm{E}}(P, x = y.T : m(e), Q)$ and replace all Z by w in all instruction proofs. The proofs are still going to work, because the instructions `pushvar` , `pushc` and $\texttt{binop}_{\mathrm{op}}$ generated by ∇^{E} and ∇^{X} (for the actual parameter e) do not modify local variables (w in our case) The proof-checker does not need to know that, it just sees that the deductions are still correct, which is what we want to achieve.

In fact, we need not treat invocation-var any different than the subst-rule. The only difference is that we replace a *local variable* instead of a logical variable. f, g are not commutative with respect to replacement of logical variables with local variables in general. But as explained in the previous paragraph, we also know a lot more (see also section 4.6 on page 27 and section 4.8.1 on the preceding page): we know that for the evaluation of expressions, only rules with $f, g \in \{\mathrm{shift}, (.)[\cdots/s(0)], (\mathrm{shift}(.))[(s(1)\,\mathrm{op}\,s(0))/s(1)]\}$ are generated, the variable w we may introduce is not used in the assignment. Taking these facts into account, we can safely treat invocation-var for all rules exactly as the subst-rule.

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{f(A) \to g(B)}}{(f(A) \to g(B))[w/Z]}}{\cdots f(A[w/Z]) \to g(B[w/Z]) \cdots}}{\left\{E^{(l)}[w/Z]\right\}\ l : \mathrm{instr}\ \left\{E^{(l+1)}[w/Z]\right\}}$$

`invokevirtual` is treated similarly as in section 4.7.5 on page 31 by *insertion of the specialized invokevar rule.*

Note that we can restrict our attention to local variables and we do not need to include stack elements in our consideration although that would also be possible with our invokevar rule (section 3.4.3 on page 18).

Example 11. Consider the virtual method invocation:

$$\text{invokevar}\,\dfrac{\text{inv}\,\dfrac{\text{invoke}\,\dfrac{\{\texttt{p} = p_0\}\ \ \mathit{Add2}\ \ \{\texttt{result} = (p_0 + 2)\}}{\{\texttt{p} = p_0[\texttt{x}/\texttt{p}][\texttt{o}/\texttt{this}]\}\ \ x = o.\mathit{Add2}(\texttt{x})\ \ \{\texttt{result} = (p_0 + 2)[\texttt{x}/\texttt{result}]\}}}{\{(\texttt{p} = p_0[\texttt{x}/\texttt{p}][\texttt{o}/\texttt{this}] \wedge Z = 5)\}\ \ x = o.\mathit{Add2}(\texttt{x})\ \ \{(\texttt{result} = (p_0 + 2)[\texttt{x}/\texttt{result}] \wedge Z = 5)\}}}{\{(\texttt{p} = p_0[\texttt{x}/\texttt{p}][\texttt{o}/\texttt{this}] \wedge Z = 5)[\texttt{t}/Z]\}\ \ x = o.\mathit{Add2}(\texttt{x})\ \ \{(\texttt{result} = (p_0 + 2)[\texttt{x}/\texttt{result}] \wedge Z = 5)[\texttt{t}/Z]\}}$$

simplified:

$$\text{invokevar}\,\dfrac{\text{inv}\,\dfrac{\text{invoke}\,\dfrac{\{\texttt{p} = p_0\}\ \ \mathit{Add2}\ \ \{\texttt{result} = (p_0 + 2)\}}{\{\texttt{x} = p_0\}\ \ x = o.\mathit{Add2}(\texttt{x})\ \ \{\texttt{x} = (p_0 + 2)\}}}{\{(\texttt{x} = p_0 \wedge Z = 5)\}\ \ x = o.\mathit{Add2}(\texttt{x})\ \ \{(\texttt{x} = (p_0 + 2) \wedge Z = 5)\}}}{\{(\texttt{x} = p_0 \wedge \texttt{t} = 5)\}\ \ x = o.\mathit{Add2}(\texttt{x})\ \ \{(\texttt{x} = (p_0 + 2) \wedge \texttt{t} = 5)\}}$$

Applying ∇^{S} yields:

1. $\dfrac{}{\{(\texttt{x} = p_0 \wedge \texttt{t} = 5)\}\ 1 : \texttt{nop}\ \{((\texttt{x} = p_0 \wedge \texttt{o} = \texttt{o}) \wedge \texttt{t} = 5)\}}$
2. $\dfrac{}{\{((\texttt{x} = p_0 \wedge \texttt{o} = \texttt{o}) \wedge \texttt{t} = 5)\}\ 2 : \texttt{pushvar o}\ \{((\texttt{x} = p_0 \wedge \texttt{o} = s(0)) \wedge \texttt{t} = 5)\}}$
3. $\dfrac{}{\{((\texttt{x} = p_0 \wedge \texttt{o} = s(0)) \wedge \texttt{t} = 5)\}\ 3 : \texttt{nop}\ \{(((\texttt{x} = p_0 \wedge \texttt{o} = s(0)) \wedge \texttt{x} = \texttt{x}) \wedge \texttt{t} = 5)\}}$
4. $\dfrac{}{\{(((\texttt{x} = p_0 \wedge \texttt{o} = s(0)) \wedge \texttt{x} = \texttt{x}) \wedge \texttt{t} = 5)\}\ 4 : \texttt{pushvar x}\ \{(((\texttt{x} = p_0 \wedge \texttt{o} = s(1)) \wedge \texttt{x} = s(0)) \wedge \texttt{t} = 5)\}}$

5. $$\dfrac{\dfrac{\dfrac{\{\mathtt{p} = p_0\}\ \mathit{Add2}\ \{\mathtt{result} = (p_0 + 2)\}}{\{(\mathtt{p} = p_0 \wedge Z = 5)\}\ \mathit{Add2}\ \{(\mathtt{result} = (p_0 + 2) \wedge Z = 5)\}}}{\{(((\mathtt{x} = p_0 \wedge \mathtt{o} = s(1)) \wedge \mathtt{x} = s(0)) \wedge Z = 5)\}\ 5 : \mathtt{invokevirtual}\ \mathit{Add2}\ \{(s(0) = (p_0 + 2) \wedge Z = 5)\}}}{\{(((\mathtt{x} = p_0 \wedge \mathtt{o} = s(1)) \wedge \mathtt{x} = s(0)) \wedge \mathtt{t} = 5)\}\ 5 : \mathtt{invokevirtual}\ \mathit{Add2}\ \{(s(0) = (p_0 + 2) \wedge \mathtt{t} = 5)\}}$$

6. $$\dfrac{}{\{(s(0) = (p_0 + 2) \wedge \mathtt{t} = 5)\}\ 6 : \mathtt{pop\ x}\ \{(\mathtt{x} = (p_0 + 2) \wedge \mathtt{t} = 5)\}}$$

or in linear form:

1. $$\boxed{\begin{array}{l} \quad\text{nop} \\ \{(\mathtt{x} = p_0 \wedge \mathtt{t} = 5)\} \\ 1 : \mathtt{nop} \\ \{((\mathtt{x} = p_0 \wedge \mathtt{o} = \mathtt{o}) \wedge \mathtt{t} = 5)\} \end{array}}$$

2. $$\boxed{\begin{array}{l} \quad\text{pushvar} \\ \{((\mathtt{x} = p_0 \wedge \mathtt{o} = \mathtt{o}) \wedge \mathtt{t} = 5)\} \\ 2 : \mathtt{pushvar\ o} \\ \{((\mathtt{x} = p_0 \wedge \mathtt{o} = s(0)) \wedge \mathtt{t} = 5)\} \end{array}}$$

3. $$\boxed{\begin{array}{l} \quad\text{nop} \\ \{((\mathtt{x} = p_0 \wedge \mathtt{o} = s(0)) \wedge \mathtt{t} = 5)\} \\ 3 : \mathtt{nop} \\ \{(((\mathtt{x} = p_0 \wedge \mathtt{o} = s(0)) \wedge \mathtt{x} = \mathtt{x}) \wedge \mathtt{t} = 5)\} \end{array}}$$

4. $$\boxed{\begin{array}{l} \quad\text{pushvar} \\ \{(((\mathtt{x} = p_0 \wedge \mathtt{o} = s(0)) \wedge \mathtt{x} = \mathtt{x}) \wedge \mathtt{t} = 5)\} \\ 4 : \mathtt{pushvar\ x} \\ \{(((\mathtt{x} = p_0 \wedge \mathtt{o} = s(1)) \wedge \mathtt{x} = s(0)) \wedge \mathtt{t} = 5)\} \end{array}}$$

5. $$\boxed{\begin{array}{l} \quad\text{invokevar} \\ \{(((\mathtt{x} = p_0 \wedge \mathtt{o} = s(1)) \wedge \mathtt{x} = s(0)) \wedge \mathtt{t} = 5)\} \\ 5 : \boxed{\begin{array}{l} \quad\text{invokevirt} \\ \{(((\mathtt{x} = p_0 \wedge \mathtt{o} = s(1)) \wedge \mathtt{x} = s(0)) \wedge Z = 5)\} \\ 5 : \mathtt{invokevirtual}\ \mathit{Add2} \\ \{(s(0) = (p_0 + 2) \wedge Z = 5)\} \end{array}} \\ \{(s(0) = (p_0 + 2) \wedge \mathtt{t} = 5)\} \end{array}}$$

6. $$\boxed{\begin{array}{l} \quad\text{pop} \\ \{(s(0) = (p_0 + 2) \wedge \mathtt{t} = 5)\} \\ 6 : \mathtt{pop\ x} \\ \{(\mathtt{x} = (p_0 + 2) \wedge \mathtt{t} = 5)\} \end{array}}$$

4.8.3 cast

$$\bigtriangledown^{\mathrm{S}}\left(\text{cast}\frac{}{\{\tau(e)\preceq T\wedge P[e/x]\}\ \ x=(T)e\ \ \{P\}}\right)=$$

$$\begin{aligned}
V:&\{Q_s\equiv\ \tau(e)\preceq T\wedge P[e/x]\}\\
&\bigtriangledown^{\mathrm{E}}(Q_s,e,\mathrm{shift}(Q_s)\wedge s(0)=e)\\
C:&\{\mathrm{shift}(Q_s)\wedge s(0)=e\}\\
&\texttt{checkcast}\\
A:&\{\mathrm{shift}(Q_s)\wedge s(0)=e\}\\
&\texttt{pop}\ x\\
&\{P\}
\end{aligned}$$

Reasons

C : `checkcast`

$$\begin{aligned}
&\mathrm{shift}(Q_s)\wedge s(0)=e\rightarrow\mathrm{shift}(Q_s)\wedge s(0)=e\wedge\ \tau(s(0))\preceq T\\
=&\,\tau(e)\preceq T\wedge P[e/x]\wedge s(0)=e\rightarrow\tau(e)\preceq T\wedge P[e/x]\wedge s(0)=e\wedge\ \tau(s(0))\preceq T
\end{aligned}$$

A : `pop` x

$$\begin{aligned}
&\mathrm{shift}(Q_s)\wedge s(0)=e\rightarrow(\mathrm{shift}(P))[s(0)/x]\\
=&\,\tau(e)\preceq T\wedge P[e/x]\wedge s(0)=e\rightarrow P[s(0)/x]
\end{aligned}$$

4.8.4 construct

$$\bigtriangledown^{\mathrm{S}}\left(\text{construct}\frac{}{\{P[\mathit{new}(\$,T)/x,\$\langle T\rangle/\,\$]\}\ \ x=\mathrm{new}\,T()\ \ \{P\}}\right)=$$

$$\begin{aligned}
V:&\{Q_s\equiv P[\mathrm{new}(\$,T)/x,\$\langle T\rangle/\,\$]\}\\
&\texttt{newobj}\ T\\
A:&\{(\mathrm{shift}(P))[s(0)/x]\}\\
&\texttt{pop}\ x\\
&\{P\}
\end{aligned}$$

Reasons

V : `newobj` T

$$\begin{aligned}
&\mathrm{shift}(Q_s)\\
&\quad\rightarrow ((\mathrm{shift}(P))[s(0)/x])[\mathrm{new}(\$, T)/s(0), \$\langle T\rangle/\,\$]\\
=&Q_s\\
&\quad\rightarrow P[s(0)/x][\mathrm{new}(\$, T)/s(0), \$\langle T\rangle/\,\$]\\
=&Q_s\\
&\quad\rightarrow P[\mathrm{new}(\$, T)/x, \mathrm{new}(\$, T)/s(0), \$\langle T\rangle/\,\$]\\
=&Q_s\\
&\quad\rightarrow \underbrace{P[\mathrm{new}(\$, T)/x, \$\langle T\rangle/\,\$]}_{=Q_s}
\end{aligned}$$

A : `pop` x

$$\begin{aligned}
&(\mathrm{shift}(P))[s(0)/x]\\
&\rightarrow (\mathrm{shift}(P))[s(0)/x]
\end{aligned}$$

4.8.5 read

$$\nabla^{\mathrm{S}}\left(\mathrm{read}\frac{}{\{y \neq \text{ null } \wedge P[\$(\mathrm{iv}(y, S@a))/x]\}\ x = y.S@a\ \{P\}}\right) =$$

$$\begin{aligned}
S :& \{Q_s \equiv y \neq \text{ null } \wedge P[\$(\mathrm{iv}(y, S@a))/x]\}\\
&\quad \texttt{pushvar}\ y\\
V :& \{\mathrm{shift}(Q_s) \wedge s(0) = y\}\\
&\quad \texttt{getfield}\ S@a\\
A :& \{\mathrm{shift}(Q_s) \wedge s(0) = \$(\mathrm{iv}(y, S@a))\}\\
&\quad \texttt{pop}\ x\\
&\quad \{P\}
\end{aligned}$$

Reasons

S : `pushvar` y

$$\begin{aligned}
&\mathrm{shift}(Q_s) \rightarrow (\mathrm{shift}(Q_s) \wedge s(0) = y)[y/s(0)]\\
=&\mathrm{shift}(Q_s) \rightarrow \mathrm{shift}(Q_s) \wedge y = y
\end{aligned}$$

V : `getfield` $S@a$

$$\begin{aligned}
&\text{shift}(Q_s) \wedge s(0) = y\\
&\quad\rightarrow (\text{shift}(Q_s) \wedge s(0) = \$(\text{iv}(y, S@a)))[\$(\text{iv}(s(0), S@a))/s(0)] \wedge s(0) \neq \text{ null}\\
=&\,\text{shift}(Q_s) \wedge s(0) = y\\
&\quad\rightarrow (\text{shift}(Q_s) \wedge\ \$(\text{iv}(s(0), S@a)) = \$(\text{iv}(y, S@a)))[\$(\text{iv}(s(0), S@a))/s(0)] \wedge s(0) \neq \text{ null}\\
=&\,\text{shift}(Q_s) \wedge s(0) = y\\
&\quad\rightarrow (\text{shift}(Q_s))[\$(\text{iv}(s(0), S@a))/s(0)] \wedge s(0) \neq \text{ null}\\
=&Q_s \wedge s(0) = y\\
&\quad\rightarrow Q_s \wedge s(0) \neq \text{ null}\\
=&\,\underbrace{(Q_s \wedge y \neq \text{ null})}_{Q_s} \wedge s(0) = y\\
&\quad\rightarrow Q_s \wedge s(0) \neq \text{ null}
\end{aligned}$$

A : `pop` x

$$\begin{aligned}
&\text{shift}(Q_s) \wedge s(0) = \$(\text{iv}(y, S@a)) \rightarrow (\text{shift}(P))[s(0)/x]\\
=&Q_s \wedge s(0) = \$(\text{iv}(y, S@a))\\
&\quad\rightarrow P[s(0)/x]\\
=&y \neq \text{ null} \wedge P[\$(\text{iv}(y, S@a))/x] \wedge s(0) = \$(\text{iv}(y, S@a))\\
&\quad\rightarrow P[s(0)/x]
\end{aligned}$$

4.8.6 write

$$\bigtriangledown^{\text{S}}\left(\text{write}\frac{}{\{y \neq \text{ null} \wedge P[\$\langle\text{iv}(y, S@a) := e\rangle/\ \$]\}\ \ y.S@a = e\ \ \{P\}}\right) =$$

$$\begin{aligned}
S :\, & \{Q_s \equiv y \neq \text{ null} \wedge P[\$\langle\text{iv}(y, S@a) := e\rangle/\ \$]\}\\
& \texttt{pushvar}\ y\\
V :\, & \{\text{shift}(Q_s) \wedge s(0) = y\}\\
& \bigtriangledown^{\text{E}}(\text{shift}(Q_s) \wedge s(0) = y, e, Q_x \equiv\ \text{shift}(\text{shift}(Q_s) \wedge s(0) = y) \wedge s(0) = e)\\
A :\, & \{Q_x\}\\
& \texttt{putfield}\ S@a\\
& \{P\}
\end{aligned}$$

Reasons

S : `pushvar` y

$$\begin{aligned}
&\text{shift}(Q_s) \rightarrow (\text{shift}(Q_s) \wedge s(0) = y)[y/s(0)]\\
=&\,\text{shift}(Q_s) \rightarrow \text{shift}(Q_s) \wedge y = y
\end{aligned}$$

$A :$ `putfield` $S@a$

$$\begin{aligned}
Q_x \to\ & \underbrace{(\text{shift}^2(P))[\$\langle \text{iv}(s(1), S@a) := s(0)\rangle / \$] \wedge s(1) \neq \text{null}}_{G} \\
=\ & \text{shift}(\text{shift}(Q_s) \wedge s(0) = y) \wedge s(0) = e \to G \\
=\ & \text{shift}^2(Q_s) \wedge s(1) = y \wedge s(0) = e \to G \\
=\ & y \neq \text{null} \wedge \underbrace{\text{shift}^2(P[\$\langle \text{iv}(y, S@a) := e\rangle / \$])}_{\text{does not contain references to the stack}} \wedge s(1) = y \wedge s(0) = e \to G \\
=\ & y \neq \text{null} \wedge P[\$\langle \text{iv}(y, S@a) := e\rangle / \$] \wedge s(1) = y \wedge s(0) = e \to G \\
=\ & P[\$\langle \text{iv}(y, S@a) := e\rangle / \$] \wedge y \neq \text{null} \wedge s(1) = y \wedge s(0) = e \\
& \to P[\$\langle \text{iv}(s(1), S@a) := s(0)\rangle / \$] \wedge s(1) \neq \text{null}
\end{aligned}$$

4.9 Methods

$$\bigtriangledown^{\text{S}} \left(\frac{\{P\}\ \mathit{body}(T@m)\ \{Q\}}{\{P\}\ T@m\ \{Q\}} \right) =$$

$$\text{VM}_\text{K}\; \frac{\mathfrak{p} = \left[\begin{array}{l} S : \{P' = P \wedge \text{this} \neq \text{null}\} \\ \quad \bigtriangledown^{\text{S}}(\{P'\}\ \mathit{body}(T@m)\ \{Q\}) \\ T : \{Q\} \\ \quad \texttt{end\_method} \\ \quad \{Q\} \end{array}\right] \qquad \begin{array}{r} I_{|\mathfrak{p}|} = \texttt{end\_method} \\ \forall i < |\mathfrak{p}| : \mathfrak{p}(i) \neq \texttt{end\_method} \\ \text{precondition}_{\mathfrak{p}}(1) = P \wedge \text{this} \neq \text{null} \\ \text{postcondition}_{\mathfrak{p}}(|\mathfrak{p}|) = Q \end{array}}{\{P\}\ T@m\ \{Q\}}$$

5 Application

In this section, we present a Java-K proof and its automatic translation in order to give a feeling – together with implementation in section A on page 49 – how the translation rules are used. The example is necessarily trivial. That's why it does not use some of the more complex instruction rules. It calculates the factorial of a number.

We first give a linear form of the proof, then the actual proof tree and its translation into a VM$_\text{K}$ proof.

5.1 Java-K, linear

```
public class Trivial {
        public int f(int x) {
               $\{x = x_0\}$
               result = 1;
               $\{x = x_0 \wedge \text{result} = 1\}$
               $\Longrightarrow$
               $\{\text{result} \cdot x! = x_0!\}$
               while(x != 0){
                       $\{x \neq 0 \wedge \text{result} \cdot x! = x_0!\}$
                       $\Longrightarrow$
```

```
                    $\{\text{result}\cdot x\cdot(x-1)! = x_0!\}$
                    result = result*x;
                    $\{\text{result}\cdot(x-1)! = x_0!\}$
                    x = x-1;
                    $\{\text{result}\cdot x! = x_0!\}$
                }
                $\{\text{result}\cdot x! = x_0! \wedge x = 0\}$
                $\Longrightarrow$
                $\{\text{result} = x_0!\}$
        }
}
```

5.2 Java-K, Tree

$$T_0 = \dfrac{\dfrac{\dfrac{\text{id}\dfrac{T_1}{\{P\}\ \mathit{body}_{\mathrm{VM}_\mathrm{K}}(\texttt{Trivial@f})\ \{Q\}}}{\{P\}\ \texttt{Trivial@f}\ \{Q\}}}{\cdots\ \text{see [PHM99]}}}{\{P \equiv x = x_0\}\ \texttt{Trivial}:\texttt{f}\ \{Q \equiv \text{ result} = x_0!\}}$$

$$T_1 = \text{weak}\dfrac{\text{seq}\dfrac{T_2 \quad T_3}{\{x = x_0\}\ \texttt{result} = \texttt{1}; \texttt{while}(...)...\ \{\text{result}\cdot x! = x_0! \wedge x = 0\}}}{\{x = x_0\}\ \texttt{result} = \texttt{1}; \texttt{while}(...)...\ \{\text{result} = x_0!\}}$$

$$T_2 = \text{weak}\dfrac{\text{assign}\dfrac{}{\{x = x_0\}\ \texttt{result} = \texttt{1}\ \{x = x_0 \wedge \text{ result} = 1\}}}{\{x = x_0\}\ \texttt{result} = \texttt{1}\ \{\text{result}\cdot x! = x_0!\}}$$

$$T_3 = \text{while}\dfrac{\text{seq}\dfrac{T_4 \quad T_5}{\{x \neq 0 \wedge \text{ result}\cdot x! = x_0!\}\ ...; ...\ \{\text{result}\cdot x! = x_0!\}}}{\{\text{result}\cdot x! = x_0!\}\ \texttt{while}(...)...\ \{\text{result}\cdot x! = x_0! \wedge x = 0\}}$$

$$T_4 = \text{strength}\dfrac{\text{assign}\dfrac{}{\{\text{result}\cdot x\cdot(x-1)! = x_0!\}\ \texttt{result} = \texttt{result}\cdot\texttt{x};\ \{\text{result}\cdot(x-1)! = x_0!\}}}{\{x \neq 0 \wedge \text{ result}\cdot x! = x_0!\}\ \texttt{result} = \texttt{result}\cdot\texttt{x};\ \{\text{result}\cdot(x-1)! = x_0!\}}$$

$$T_5 = \text{assign}\dfrac{}{\{\text{result}\cdot(x-1)! = x_0!\}\ \texttt{x} = \texttt{x} - \texttt{1};\ \{\text{result}\cdot x! = x_0!\}}$$

5.3 The Translation

The translation has been automatically generated from the source proof. result is abbreviated by **r** (to save space). "Reasons" for the antecedents are not given but they are trivial to check. The assumption this ≠ null is left implicit in our assertions.

$$\frac{\left[\begin{array}{l}
\overline{\{\mathtt{x}=x_0\}\ 1:\mathtt{nop}\ \{(\mathtt{x}=x_0\wedge 1=1)\}} \\
\overline{\{(\mathtt{x}=x_0\wedge 1=1)\}\ 2:\mathtt{pushc\ 1}\ \{(\mathtt{x}=x_0\wedge 1=s(0))\}} \\
\overline{\{(\mathtt{x}=x_0\wedge 1=s(0))\}\ 3:\mathtt{pop\ r}\ \{(\mathtt{x}=x_0\wedge \mathtt{r}=1)\}} \\
\overline{\{(\mathtt{x}=x_0\wedge \mathtt{r}=1)\}\ 4:\mathtt{nop}\ \{(\mathtt{r}\cdot(\mathtt{x}!))=x_0!\}} \\
\overline{\{(\mathtt{r}\cdot(\mathtt{x}!))=x_0!\}\ 5:\mathtt{goto\ 17}\ \{((\mathtt{r}\cdot(\mathtt{x}!))=x_0!\wedge \mathtt{x}\neq 0)\}} \\
\overline{\{((\mathtt{r}\cdot(\mathtt{x}!))=x_0!\wedge \mathtt{x}\neq 0)\}\ 6:\mathtt{nop}\ \{((\mathtt{r}\cdot\mathtt{x})\cdot((\mathtt{x}-1)!))=x_0!\}} \\
\overline{\{((\mathtt{r}\cdot\mathtt{x})\cdot((\mathtt{x}-1)!))=x_0!\}\ 7:\mathtt{nop}\ \{(((\mathtt{r}\cdot\mathtt{x})\cdot((\mathtt{x}-1)!))=x_0!\wedge(\mathtt{r}\cdot\mathtt{x})=(\mathtt{r}\cdot\mathtt{x}))\}} \\
\overline{\{(((\mathtt{r}\cdot\mathtt{x})\cdot((\mathtt{x}-1)!))=x_0!\wedge(\mathtt{r}\cdot\mathtt{x})=(\mathtt{r}\cdot\mathtt{x}))\}\ 8:\mathtt{pushvar\ r}\ \{(((\mathtt{r}\cdot\mathtt{x})\cdot((\mathtt{x}-1)!))=x_0!\wedge(\mathtt{r}\cdot\mathtt{x})=(s(0)\cdot\mathtt{x}))\}} \\
\overline{\{(((\mathtt{r}\cdot\mathtt{x})\cdot((\mathtt{x}-1)!))=x_0!\wedge(\mathtt{r}\cdot\mathtt{x})=(s(0)\cdot\mathtt{x}))\}\ 9:\mathtt{pushvar\ x}\ \{(((\mathtt{r}\cdot\mathtt{x})\cdot((\mathtt{x}-1)!))=x_0!\wedge(\mathtt{r}\cdot\mathtt{x})=(s(1)\cdot s(0)))\}} \\
\overline{\{(((\mathtt{r}\cdot\mathtt{x})\cdot((\mathtt{x}-1)!))=x_0!\wedge(\mathtt{r}\cdot\mathtt{x})=(s(1)\cdot s(0)))\}\ 10:\mathtt{binop}\cdot\ \{(((\mathtt{r}\cdot\mathtt{x})\cdot((\mathtt{x}-1)!))=x_0!\wedge(\mathtt{r}\cdot\mathtt{x})=s(0))\}} \\
\overline{\{(((\mathtt{r}\cdot\mathtt{x})\cdot((\mathtt{x}-1)!))=x_0!\wedge(\mathtt{r}\cdot\mathtt{x})=s(0))\}\ 11:\mathtt{pop\ r}\ \{(\mathtt{r}\cdot((\mathtt{x}-1)!))=x_0!\}} \\
\overline{\{(\mathtt{r}\cdot((\mathtt{x}-1)!))=x_0!\}\ 12:\mathtt{nop}\ \{((\mathtt{r}\cdot((\mathtt{x}-1)!))=x_0!\wedge(\mathtt{x}-1)=(\mathtt{x}-1))\}} \\
\overline{\{((\mathtt{r}\cdot((\mathtt{x}-1)!))=x_0!\wedge(\mathtt{x}-1)=(\mathtt{x}-1))\}\ 13:\mathtt{pushvar\ x}\ \{((\mathtt{r}\cdot((\mathtt{x}-1)!))=x_0!\wedge(\mathtt{x}-1)=(s(0)-1))\}} \\
\overline{\{((\mathtt{r}\cdot((\mathtt{x}-1)!))=x_0!\wedge(\mathtt{x}-1)=(s(0)-1))\}\ 14:\mathtt{pushc\ 1}\ \{((\mathtt{r}\cdot((\mathtt{x}-1)!))=x_0!\wedge(\mathtt{x}-1)=(s(1)-s(0)))\}} \\
\overline{\{((\mathtt{r}\cdot((\mathtt{x}-1)!))=x_0!\wedge(\mathtt{x}-1)=(s(1)-s(0)))\}\ 15:\mathtt{binop}-\ \{((\mathtt{r}\cdot((\mathtt{x}-1)!))=x_0!\wedge(\mathtt{x}-1)=s(0))\}} \\
\overline{\{((\mathtt{r}\cdot((\mathtt{x}-1)!))=x_0!\wedge(\mathtt{x}-1)=s(0))\}\ 16:\mathtt{pop\ x}\ \{(\mathtt{r}\cdot(\mathtt{x}!))=x_0!\}} \\
\overline{\{(\mathtt{r}\cdot(\mathtt{x}!))=x_0!\}\ 17:\mathtt{nop}\ \{((\mathtt{r}\cdot(\mathtt{x}!))=x_0!\wedge(\mathtt{x}\neq 0)=(\mathtt{x}\neq 0))\}} \\
\overline{\{((\mathtt{r}\cdot(\mathtt{x}!))=x_0!\wedge(\mathtt{x}\neq 0)=(\mathtt{x}\neq 0))\}\ 18:\mathtt{pushvar\ x}\ \{((\mathtt{r}\cdot(\mathtt{x}!))=x_0!\wedge(\mathtt{x}\neq 0)=(s(0)\neq 0))\}} \\
\overline{\{((\mathtt{r}\cdot(\mathtt{x}!))=x_0!\wedge(\mathtt{x}\neq 0)=(s(0)\neq 0))\}\ 19:\mathtt{pushc\ 0}\ \{((\mathtt{r}\cdot(\mathtt{x}!))=x_0!\wedge(\mathtt{x}\neq 0)=(s(1)\neq s(0)))\}} \\
\overline{\{((\mathtt{r}\cdot(\mathtt{x}!))=x_0!\wedge(\mathtt{x}\neq 0)=(s(1)\neq s(0)))\}\ 20:\mathtt{binop}\neq\ \{((\mathtt{r}\cdot(\mathtt{x}!))=x_0!\wedge(\mathtt{x}\neq 0)=s(0))\}} \\
\overline{\{((\mathtt{r}\cdot(\mathtt{x}!))=x_0!\wedge(\mathtt{x}\neq 0)=s(0))\}\ 21:\mathtt{brtrue\ 6}\ \{((\mathtt{r}\cdot(\mathtt{x}!))=x_0!\wedge \mathtt{x}=0)\}} \\
\overline{\{((\mathtt{r}\cdot(\mathtt{x}!))=x_0!\wedge \mathtt{x}=0)\}\ 22:\mathtt{nop}\ \{\mathtt{r}=x_0!\}} \\
\overline{\{\mathtt{r}=x_0!\}\ 23:\mathtt{end\_method}\ \{\mathtt{r}=x_0!\}}
\end{array}\right]}{\{\mathtt{x}=x_0\}\ \mathit{Trivial}@f\ \{\mathtt{r}=x_0!\}}$$

6 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 CLR, but the results are the same. The formalism of our logic is based on [Ben04].

7 Conclusion

We have presented the operational semantics and a programming logic for the bytecode of $\mathrm{VM_K}$, a virtual machine similar to the JVM or the CLR. Possible uses of this approach include language interoperability of specifications and trusted components that are not available in source form.

To show complex properties of a program, we usually have to produce a proof by hand. This is done on the source level and not on the bytecode level. In order to make applications of a bytecode logic possible, we therefore need an automatic translation procedure from source proofs. We have presented such a translation procedure together with a simple prototype implementation.

The bytecode logic is a simple 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 flows. Checking a proof is purely local: we can check one instruction at a time.

The simplicity of our proof translation is fundamentally tied to the special shape[13] of the deductive rules we introduced. It allowed us to omit instruction independent rules like the weak, inv, subst rules from our bytecode logic. Unfortunately, we were not able to translate the ex- and all-rules to our logic.

References

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

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

[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.

[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.

[13] we called it f, g-form

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

[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.

A Implementation

This appendix contains an implementation of the translation in SML. It is meant to demonstrate that the translation is easy.

The "reasons" sections did not find their way into this translation because we did not want to restrict ourselves to specific underlying logic and a deductive system for that logic. We do not include the trivial invariant this $\neq$ null in our proofs, either.

The actual compilation does not begin before line 217 with the function `CompileX`. Three functions are defined that correspond to the three translation functions defined in section 4 on page 22:

SML function name	Symbol
`CompileX`	$\bigtriangledown^{\mathrm{X}}$
`CompileE`	$\bigtriangledown^{\mathrm{E}}$
`CompileS`	$\bigtriangledown^{\mathrm{S}}$

The translation from the definitions in the text to the runnable SML program is quite direct. Still, the definition of the more difficult translations (line 243) may be interesting when an existing verified instruction sequence is modified (for the substitution rule for instance).

The translation function `CompileImpl` (line 396) is just a distinct name for $\bigtriangledown^{\mathrm{S}}$ when method implementations should be translated, which is our ultimate goal.

The generation of labels is quite mundane: the `genuniqueid` (line 115) is a helper function to produce fresh variables and labels. It works by updating a counter. In order to get the numbers right, we take care to do the recursive invocation of the translation functions in the order in which their results will appear in the final instruction sequence.

The functions at line 154 are used for the simplification of expressions. They are not strictly necessary, but help when producing printable output.

Because there is only one rule for most instructions, we declare (line 199) the function that does the mapping between instructions and rules.

The rest of the code is straight-forward and should not need any additional explanation.

- The datatype `Expr` (line 3) is used to represent all assertions/predicates as well as expressions in the program. Operations that are strictly required to handle the proofs are included (e.g. conjunction, disjunction) plus a few others. `prg, lvr, stk` represent program, logical and stack variables. There is one operation for the replacement of each kind of variables.

- We treat the object-store $ (line 28) just like an ordinary program variable.
- `Stmt` (line 30) represents arbitrary computations both for Java-K and VM_K.
- `Rule` (line 60) is an enumeration of all possible rules. Again, there is no separation between Java-K and VM_K. The antecedents are not stored with the individual rules in order to avoid lengthy repetitions of similar declarations.
- `Proof` (line 102) stores an actual proof. There are two kinds of proofs: those constructed using `DT`, representing proofs for Hoare-triples and those that prove assertions (constructed with `DE`). As before, it may be safer to treat Hoare-triple proofs and assertion proofs as different datatypes, but the unification chosen here aims at making the program smaller.
- Starting at line 104, some accessors for the parts of the proofs are declared, we use them because the DT constructor is quite complicated.

```
(* Transformation of SJ-Proofs to SVM-Proofs *)
(* Predicates/Assertions for SJ/SVM *)
datatype Expr = falseP | trueP
              | implies of Expr * Expr
              | conj of Expr * Expr
              | disj of Expr * Expr
                (* for lvr's *)
              | allP of string * Expr
              | exP of string * Expr
              | shiftP of Expr
              | unshiftP of Expr
              (* relational operators *)
              | eqP of Expr * Expr
              | neP of Expr * Expr
              | binop of string * Expr * Expr
              | unop of string * Expr
              | prg of string
              | lvr of string
              | stk of int
              | genop of string * Expr list
              | replacelvr of string * Expr * Expr
              | replaceprg of string * Expr * Expr
              | replacestk of int * Expr * Expr
              | literal of string
                        (* OS *)(*obj*)(*field*)
              | instvar of Expr * Expr * string
              | instvarupdate of Expr * Expr * string * Expr
val OS = prg "$"
(* SJ Stmts and Methods *)
datatype Stmt = cast of string * string * Expr
              | ctor of string * string
              | assign of string * Expr
              | readf of string * string * string
              | writef of string * string * Expr
              | invoke of string * string * string * Expr
              | swhile of Expr * Stmt
              | sif of Expr * Stmt * Stmt
              | seq of Stmt * Stmt
              | skipstmt
              | kmeth of string (* T@m *)
              | vmeth of string
              | pushc of string
              | pushvar of string
              | pop of string
              | iunop of string
              | ibinop of string
              | goto of string (* labels are strings *)
              | brtrue of string
              | checkcast of string
              | newobj of string
              | invokevirtual of string
              | getfield of string
              | putfield of string
              | methodend
              | nop
(*
 at the moment, no reaons are given, trees are
 trees are just for hoare-triples
*)
datatype Rule = assumed
              | omitted
              | rpushc
              | rpushvar
              | rpop
              | runop
              | rbinop
              | rgoto
              | rbrtrue
              | rcheckcast
              | rnewobj
              | rgetfield
              | rputfield
              | rinvokevirt (* one antecedent *)
              | rinvokevar (* one antecedent *)
              | rnop
              | rmethod
              | rimplementation (* antecedent is proof-list *)
              (* SJ *)
              | xcast
              | xassign
              | xctor
              | xreadf
              | xwritef
              | xinvoke (* one antecedent *)
              | xinvokevar (* one antecedent pre must be *exactely" replacelvr(Z
                 ,w, P) *)
              | ximpl (* one antecedent *)
              (* class- and subtype- rule are not modelled here *)
              | xwhile (* one antecedent *)
              | xif   (* two antecedents *)
              | xseq  (* two antecedents *)
              | xfalse
              | xconj  (* two antecedents *)
              | xdisj  (* two antecedents *)
              | xstren (* one antecedent *)
              | xweak  (* one antecedent *)
              | xinv (* one antecedent, pre must be *exactely* conj(P,R) *)
              | xsub (* one antecedent, pre must be *exactely" replacelvr(Z,t, P
                 ) *)
              | xall (* one antecedent *)
              | xex (* one antecedent *)
and
             (* label, instr, precond, post, rule,  antecedents *)
 Proof = DT of string * Stmt * Expr * Expr * Rule * Proof list
       | DE of Expr (* result *)
(**** PROOF TRANSFORMATION BEGIN ***********)
fun antecedents(DT(label, instr, pre, post, rule, antes)) = antes
fun precondition(DT(label, instr, pre, post, rule, antes)) = pre
fun postcondition(DT(label, instr, pre, post, rule, antes)) = post
fun instruction(DT(label, instr, pre, post, rule, antes)) = instr
fun label(DT(label, instr, pre, post, rule, antes)) = label
fun first L = hd L
fun last L = List.last L
fun strictzip (a::x) (b::z) = (a,b)::(strictzip x z)
  | strictzip [] [] = []
(* generating fresh variables *)
```

```
fun genuniqueid name counter =
    (fn() => let
           val id = "{"^name^"{"^Int.toString(!counter)^"}}";
       in (counter := !counter + 1; id) end, fn() => (counter := 1))
val (freshLvr, resetLvr) = genuniqueid "Z_" (ref 1)
val (freshPrg, resetPrg) = genuniqueid "t_" (ref 1)
val (freshLab, resetLab) = genuniqueid "" (ref 1)
(* future work, for now, we assume, there are no ex/all rules *)
fun liftExRule(proof) = proof
fun liftAllRule(proof) = proof
(* maps the expressions of a proof *)
fun mapProofExprs f (proof as DT(label, instr, pre, post, rule, antes)) =
    DT(label, instr,
       f pre, f post,
       rule, map (mapProofExprs f) antes)
  | mapProofExprs f (DE(E)) = DE(f E)
(* maps the sub-expressions in an expression *)
fun mapSubExprs f expr =
    case expr of
        implies (e1, e2) => implies(f(e1), f(e2))
      | conj (e1, e2) => conj(f(e1), f(e2))
      | allP (v, e) => allP(v, f(e))
      | exP (v, e) => exP(v, f(e))
      | shiftP (e) => shiftP(f(e))
      | unshiftP (e) => unshiftP(f(e))
      | eqP (e1, e2) => eqP(f(e1), f(e2))
      | neP (e1, e2) => neP(f(e1), f(e2))
      | unop (s, e) => unop(s, f(e))
      | binop (s, e1, e2) => binop(s, f(e1), f(e2))
      | replacelvr (s, e1, e2) => replacelvr(s, f(e1), f(e2))
      | replaceprg (s, e1, e2) => replaceprg(s, f(e1), f(e2))
      | replacestk (s, e1, e2) => replacestk(s, f(e1), f(e2))
      | genop (s, L) => genop(s, map f L)
      | e => e
fun mapExprRec f expr = f(mapSubExprs (mapExprRec f) expr)
fun mapExprRecCond P f expr =
    if P(expr) then f(mapSubExprs (mapExprRecCond P f) expr)
    else expr
exception UnshiftOnTOS
fun doShift e = case e of stk i => stk (i+1) | _ => e
fun doUnshift (stk i) = if i = 0 then raise UnshiftOnTOS
                        else stk (i-1)
  | doUnshift e = e
fun doreplacelvr s By In =
    mapExprRecCond
        (fn(e) => case e of
                      allP (x,_) => s <> x
                    | exP (x,_) => s <> x
                    | _ => true)
        (fn(e) =>
           case e of lvr x => if s=x then By else e
                   |_=>e)
        In
fun doreplaceprg s By In =
    mapExprRec
        (fn(e) =>
           case e of prg x => if s=x then By else e
                   |_=>e)
        In
fun doreplacestk i By In =
    mapExprRec
        (fn(e) =>
           case e of stk k => if k=i then By else e
                   |_=>e)
        In
(* tries to evaluate as much as possible from an expression *)
fun simplifyExpr0 e' =
    case e' of
        conj(trueP, E) => E
      | disj(falseP, E) => E
      | conj(E,trueP) => E
      | disj(E,falseP) => E
      | conj(falseP, E) => falseP
      | disj(trueP, E) => trueP
      | conj(E,falseP) => falseP
      | disj(E,trueP) => trueP
      | shiftP e => mapExprRec doShift e
      | unshiftP e => mapExprRec doUnshift e
      | replacelvr (s, By, In) => doreplacelvr s By In
      | replaceprg (s, By, In) => doreplaceprg s By In
      | replacestk (i, By, In) => doreplacestk i By In
      | _ => e'
fun simplifyExpr expr = mapExprRec simplifyExpr0 expr
fun simplifyProof proof = mapProofExprs simplifyExpr proof
fun correspondingVMRule Instr =
    case Instr of
        pushc _ => rpushc
      | pushvar _ => rpushvar
      | pop _ => rpop
      | ibinop _ => rbinop
      | iunop _ => runop
      | goto _ => rgoto
      | brtrue _ => rbrtrue
      | checkcast _ => rcheckcast
      | newobj _ => rnewobj
      | invokevirtual _ => rinvokevirt
      | getfield _ => rgetfield
      | putfield _ => rputfield
      | methodend => rmethod
      | nop => rnop
fun instr(P, L, Instr, Q) = DT(L, Instr, P, Q, correspondingVMRule Instr, [])
fun instrx(P, L, Instr, Q, X) = DT(L, Instr, P, Q, correspondingVMRule Instr, X)
(** Translation of Expressions **)
fun CompileX(Qs as conj(Q, P0), e as binop(opname, e1, e2), Qe as conj(Q0, P)) =
    let val Q2 = conj(shiftP(Q), replacestk(0, binop(opname, stk(0), e2), P))
        val Q3 = conj(shiftP(shiftP(Q)),
                      replacestk(1, binop(opname, stk(1), stk(0)), shiftP(P)))
    in CompileX(Qs, e1, Q2) @ CompileX(Q2, e2, Q3)
       @ [instr(Q3, freshLab(), ibinop opname, Qe)]
    end
  | CompileX(Qs as conj(Q, P0), e as literal c, Qe as conj(Q0, P)) =
    let val L = freshLab()
    in [instr(Qs, L, pushc c, Qe)] end
  | CompileX(Qs as conj(Q, P0), e as prg x, Qe as conj(Q0, P)) =
    let val L = freshLab()
    in [instr(Qs, L, pushvar x, Qe)] end
```

```
fun CompileE(P, e, Q (* shift P ++ e = s(0) *)) =
    let
        val L = freshLab()
        val P' = conj(P, eqP(e,e))
in
    [instr(P, L, nop, P')]@ CompileX(P',e,Q)
end
fun CompileEPost(P,e) = conj(shiftP(P), eqP(e, stk 0))
(* Translation of statements *)
exception InvalidShape of string;
(* Compiling the conjunct/disjunct rule, see paper for requirements
   to call this function *)
fun CompileSConjDisj(proof as DT(_, Comp, P, Q, rule, [pS1, pS2])) = let
    val S1 = CompileS(pS1)
    val S2 = CompileS(pS2)
    fun co(x,y) = case rule of (* abstracting the rule *)
                      xconj => conj(x,y)
                    | xdisj => disj(x,y)
    (* merging two invokevirtual rules *)
    fun mergeivirt([p1 as DT(L, M1, P1, Q1, R1, _)],
                   [p2 as DT(_, M2, P2, Q2, R2, _)]) =
        [DT(L, M1, co(P1,P2), co(Q1,Q2), rule, [p1,p2])]
    (* merging one invokevar with some other rule *)
    fun mergeivar(p1 as DT(L, M1, P1, Q1, rinvokevar, [MA]), p2) =
        DT(L, M1, co(P1,precondition(p2)), co(Q1,postcondition(p2)),
           rule, [merge(MA, p2)])
    (* merging the "common" rules *)
    and merge(p1 as DT(label1, stmt1, pre1, post1, rule1, antes1),
              p2 as DT(label2, stmt2, pre2, post2, rule2, antes2)) =
        case (rule1,rule2) of
            (rinvokevar, _) => mergeivar(p1,p2)
          | (_, rinvokevar) => mergeivar(p2,p1)
          | _ => DT(label1, stmt1, co(pre1,pre2), co(post1,post2),
                    rule1, case rule1 of
                               rinvokevirt => mergeivirt(antes1,antes2)
                             | _ => map merge (strictzip antes1 antes2))
in map merge (strictzip S1 S2) end
and CompileSInv(proof as DT(_, Comp, P' as conj(P,R), Q', xinv, [pS])) =
let val S = CompileS(pS)
    fun mergeivirt([(p as DT(L, M, P, Q, _, _))]) =
        [DT(L, M, conj(P,R), conj(Q,R),
            xinv, [p])]
      | mergeivirt ([]) = (print "CompileSInv:empty"; raise InvalidShape("
         CompileSInv:empty"))
      | mergeivirt (X) = (raise InvalidShape("CompileSInv:mergeivirt"))
    fun mergeivar([p as DT(L, M, P, Q, rinvokevar, [pS])]) =
        [DT(L, M, conj(P,R), conj(Q,R),
            rinvokevar, [merge pS])]
    and merge(DT(L, St, pre, post, rule, antes)) =
        DT(L, St, conj(pre,R), conj(post,R), rule,
           case rule of rinvokevirt => mergeivirt antes
                      | rinvokevar => mergeivar antes
                      | _ => map merge antes
                             )
in map merge S end
and CompileSSub(proof as DT(_, Comp, P' as replacelvr(Z,t, P),
                            Q', xsub, [pS])) =
let val S = CompileS(pS)
```

```
    fun mergeivirt([p as DT(L, M, P, Q, R, _)]) =
        [DT(L, M, replacelvr(Z,t,P), replacelvr(Z,t,Q),
            xsub, [p])]
    fun mergeivar([p as DT(L, M, P, Q, rinvokevar, [pS])]) =
        [DT(L, M, replacelvr(Z,t,P), replacelvr(Z,t,Q),
            rinvokevar, [merge pS])]
    and merge(DT(L, St, pre, post, rule, antes)) =
        DT(L, St, replacelvr(Z,t,pre), replacelvr(Z,t,post), rule,
           case rule of
               rinvokevar => mergeivar antes
             | rinvokevirt => mergeivirt antes
             | _ => map merge antes)
in map merge S end
and CompileSInvokeVar
        (proof as DT(_, invoke(x,y,M,e),
                     P' as replacelvr(Z, prg w, P),
                     Qe, xinvokevar, [pI])) =
    let val I = CompileS(pI)
        fun merge(p as DT(L, St, pre, post, xinvokevirt, [M])) =
            DT(L, St, replacelvr(Z,prg w,pre), replacelvr(Z,prg w,post),
               rinvokevar, [p])
          | merge(DT(L, St, pre, post, rule, antes)) =
            DT(L, St, replacelvr(Z,prg w,pre), replacelvr(Z,prg w,post), rule,
               map merge antes)
    in map merge I end
and CompileS(DT(_, assign(x, e), P0, P, xassign, [])) =
    let
        val P0' = CompileEPost(P0,e)
    in
        CompileE(P0,e,P0') @ [instr(P0', freshLab(), pop x, P)]
    end
  | CompileS(DT(_, swhile(e, St), P, Q, xwhile, [pS])) =
    let val P' = precondition(pS)
        val E' = CompileEPost(P, e)
        val L = freshLab()
        val SB = CompileS(pS)
        val TB = CompileE(P, e, E')
        val B = freshLab()
        val T = label(first(TB))
        val S = label(first(SB))
    in [instr(P, L, goto T, P')] @ SB @ TB @ [instr(E', B, brtrue S, Q)] end
  | CompileS(DT(_, sif(e, St1, St2), P, Q, xif, [pS1,pS2])) =
    let val P1 = precondition(pS1)
        val P2 = precondition(pS2)
        val P' = CompileEPost(P, e)
        val AL = freshLab()
        val B2 = CompileS(pS2)
        val BL = freshLab()
        val B1 = CompileS(pS1)
        val EL = freshLab()
        val S1 = label(first(B1))
        val S2 = label(first(B2))
    in CompileE(P,e,P') @ [instr(P', AL, brtrue S1, P2)]
       @ B2 @ [instr(P, BL, goto EL, P1)]
       @ B1 @ [instr(Q, EL, nop, Q)]
    end
  | CompileS(DT(_, seq(St1, St2), P, Q, xseq, [pS1, pS2])) =
       CompileS(pS1) @ CompileS(pS2)
  | CompileS(proof as DT(_, Comp, P, Q, xconj, [pS1, pS2])) =
    CompileSConjDisj(proof)
  | CompileS(proof as DT(_, Comp, P, Q, xdisj, [pS1, pS2])) =
    CompileSConjDisj(proof)
  | CompileS(proof as DT(_, Comp, P, Q, xinv, [pS])) =
    CompileSInv(proof)
  | CompileS(proof as DT(_, Comp, P, Q, xsub, [pS])) =
    CompileSSub(proof)
  | CompileS(DT(_, St, P', Q, xstren, [pSt])) = let
        val P = precondition(pSt)
    in [instr(P', freshLab(), nop, P)] @ CompileS(pSt) end
  | CompileS(DT(_, St, P, Q', xweak, [pSt])) = let
        val Q = postcondition(pSt)
    in CompileS(pSt) @ [instr(Q, freshLab(), nop, Q')] end
  | CompileS(DT(_, invoke(x,y,M,e), Qs, Qe, xinvoke, [pM])) = let
        val Q1 = CompileEPost(Qs, prg y)
        val Qi = CompileEPost(Q1, e)
        val P = precondition pM
        val Q = postcondition pM
        val Qi' = replaceprg("result", stk 0, Q)
    in CompileE(Qs, prg y, Q1) @ CompileE(Q1, e, Qi)
       (* it is absolutely essential that we do add antecedents
          where they must be *)
       @ [instrx(Qi, freshLab(), invokevirtual M, Qi', [pM])]
       @ [instr(Qi', freshLab(), pop x, Qe)]
    end
  | CompileS(proof as DT(_, invoke(x,y,M,e), Qs, Qe, xinvokevar, [pI])) =
        CompileSInvokeVar(proof)
  | CompileS(DT(_, cast(x, T, e), P0, P, xcast, [])) =
    let val P0' = CompileEPost(P0,e)
    in CompileE(P0,e,P0')
       @ [instr(P0', freshLab(), checkcast T, P0'),
          instr(P0', freshLab(), pop x, P)]
    end
  | CompileS(DT(_, ctor(x, T), Qs, P, xctor, [])) = let
        val P' = replaceprg(x, stk 0, shiftP(P))
    in [instr(Qs, freshLab(), newobj T, P'),
        instr(P', freshLab(), pop x, P)] end
  | CompileS(DT(_, readf(x, y, F), Qs, P, xreadf, [])) = let
        val Qs' = CompileEPost(Qs, prg y)
        val P0 = conj(shiftP(Qs), eqP(instvar(OS, prg y, F), stk 0))
    in CompileE(Qs, prg y, Qs')
       @ [instr(Qs', freshLab(), getfield F, P0),
          instr(P0, freshLab(), pop x, P)] end
  | CompileS(DT(_, writef(y,F,e), Qs, P, xwritef, [])) = let
        val Qs' = CompileEPost(Qs, prg y)
        val Qx = CompileEPost(Qs', e)
    in CompileE(Qs, prg y, Qs') @ CompileE(Qs',e,Qx)
       @ [instr(Qx, freshLab(), putfield F, P)] end
  | CompileS(proof) = raise InvalidShape("CompileS")
fun CompileImpl(DT(L, M, P, Q, ximpl, [Body])) =
let val _ = resetLab()
    val S = CompileS(Body)
in DT(L,M,P,Q,rimplementation,
      S@[instr(Q, freshLab(), methodend, Q)]) end
(**** PROOF TRANSFORMATION END ***********)
```

Listing 5: Implementation