Specification of the GNU Smalltalk virtual machine

Paolo Bonzini

Abstract

This document exposes the virtual machine's architecture with respect to bytecode execution, and documents how the methods are stored as objects in memory.

The first part of the specification details a subset of the reflection classes that the virtual machine accesses in order to execute the code. The second part details the bytecode set and the invariants that should hold for a method to be valid.

This is a low-level specification. A formalization of the Smalltalk language semantics, on which the virtual machine is based, can be found in [Wol87].

1 Structure of instances of `CompiledCode`

Instances of `CompiledCode` can have byte-sized indexed instance variables which store the bytecodes; in addition, they have three named instance variables:

- `literals` is an `Array` that holds the constants and references to globals that are referenced in the code (the method's *literal pool*).
- `flags` is a `SmallInteger` that stores miscellaneous information about the method (number of locals and arguments, stack size, and so on).

`CompiledCode` has two concrete subclasses, `CompiledMethod` and `CompiledBlock`. These two classes match the two fundamental constructs for organizing program flow in the Smalltalk language—*methods* and *blocks*. Logically, methods represent actions made on an object, or in general requests to an object: by defining a method, an object exposes a contract that hides implementation details; blocks, instead, are objects that access a series of statements, and they can be passed around or stored in an instance variable. For example, the following code

```
myList collect: [ :x | x + 1 ].
```

uses blocks like this Lisp *lambda* function:

```
(mapcar (lambda (x) (+ 1 x)) *my-list*)
```

From this example, one can see that methods like `collect:` live inside a class (the class of `myList`) and can be invoked through their name; instead a block is anonymous.

Blocks are usually passed as actual parameters, so that "generic" contracts can be completed: for example, a *print this object* block can complete the contract *invoke the block for all the elements of an array*, so that all the elements of the array are printed. Blocks are also central to the implementation of *if-then-else* conditionals and loops: in this case, the methods' contract is that of implementing a given control structure, and blocks "fill the hole" of what to do depending on the truth of the condition.

Considering the implementation, a method stays at the outermost level of the static chain. Instead, a block's activation record holds a static link to another block or method's activation: this is used to access the *escaping variables* that come from outer activation records. Also, blocks are *higher-order functions* which can be invoked after the method that created them has finished running: for this reason they need a *closure*, which holds the pointer to the block and the escaping variables or the static link. Invoking particular methods on the closure object starts execution of the block[1].

1.1 Flag definition for `CompiledMethod`

The layout of the flags in a `CompiledMethod` object is as follows:

- bits 0-4 hold the number of arguments that the method receives;
- bits 5-10 hold the number of stack slots that the method needs, divided by 4. For historical reasons, this includes the number of local variables (stored by bits 11-16) in addition to the number of slots needed to store partial results of the calculations.
- bits 11-16 hold the number of local variables used by the method.
- bits 17-29, if greater than zero, attribute special behavior to the method:
  - if bits 27-29 are 1, the method returns the object on which it was invoked, without doing anything else;
  - if bits 27-29 are 2, the method returns the n-th named instance variable, where n is the content of bits 17-26;
  - if bits 27-29 are 3, the method returns the n-th slot of the *literal pool*, where n is the content of bits 17-26;
  - if bits 27-29 are 4 or 5, each method invocation triggers a primitive, whose number is stored again in bits 17-26. By invoking a primitive, Smalltalk code can ask the virtual machine to perform elementary operations like integer arithmetic, or to ask the operating systems for services like file I/O.

    Primitive numbers are not part of the specification. GNU Smalltalk's virtual machine primitives are named, and at startup the virtual machine automatically remaps the primitive numbers if needed.

    Bytecodes are executed only if the primitive fails: unlike Java `native` methods, it is possible to explicitly specify what to do in this case.

[1] Closures are themselves instances of the class `BlockClosure`. These objects' implementation however need not be specified because the virtual machine creates them dynamically at run-time; other objects described in this chapter, instead, are created by the development environment and used by the virtual machine.

Smalltalk-80 relied on this feature to implement a basic form of exception handling: various methods invoked the same primitive, with different ways to react to its failure. Nowadays, this is not needed anymore, yet it remains useful to handle only the most common cases in the primitive (which is written in C), and leave failure handling to Smalltalk code. In most cases, the triggered behavior is simply to raise an appropriate exception.

- bits 27-29 are 5, bits 17-26 still identify a primitive if they are different than zero. In addition, this value of bits 27-29 identifies that special *annotations* were attached to the method. For example, the exception handling code finds handlers by looking for a method annotation. The virtual machine usually does not look at annotations; a special encoding of bits 27-29 is reserved for this case so that annotated methods are easily distinguished[2].
- if bits 27-29 are 6, the virtual machine does not invoke the method in the standard way. Instead, it invokes a special method on the `CompiledMethod` object, the `valueWithReceiver:withArguments:` method; the two parameters are the object on which the method has just been invoked, and an `Array` holding the parameters. This setting is useful to intercept the calls to a method, for example in a code coverage tool or to insert a breakpoint; it only works for subclasses of `CompiledMethod` which redefine this special method, since the default implementation raises an exception.

1.2 Flag definition for of `CompiledBlock`

The layout of the flags in a `CompiledBlock` object is as follows:

- bits 0-5 describe how the block uses the activation records on the static chain.
  - if they are 0, the block is entirely self-contained and does not refer to any variable on the static chain; the block found in section 1 is of this kind. These blocks do not need a *closure* that links them to the environment in which they were created.
  - if they are 1, the sole referenced variable in the static chain is `self`, the object on which the enclosing method was invoked. Though they need a closure, these blocks can have stack-allocated activation records.
  - if they are 31 (all bits are set), the blocks includes a `RETURN_METHOD_STACK_TOP` bytecode, so the block needs a closure *and* heap-allocated activation records.
  - other values are reserved. The implementation should expect that the block uses the static chain to access variables, but does not use the aforementioned bytecode. It is easiest to use heap-allocated activation records in this cases, but other strategies are possible (see for example chapter 15 of [App98]).

[2] In the Smalltalk language grammar, a primitive is attached to a method with an annotation, but this is specially encoded in the `CompiledMethod` object for speed.

- bits 14-19 hold the number of stack slots that the method needs, divided by 4. For historical reasons, this includes the number of local variables (stored by bits 20-24) in addition to the number of slots needed to store partial results of the calculations.
- bits 20-24 hold the number of local variables used by the block.
- bits 25-29 hold the number of arguments that the block expects.

2 Run-time behavior of the virtual machine

As in the JVM and CLR, the virtual machine is stack-based. The bytecodes that are defined by this specification are closely related to the original Smalltalk-80 bytecodes described by [GR83], but the encoding of the instructions was redesigned in order to:

- simplify the decoding of the instructions and enable the interpreter to prefetch their arguments aggressively;
- add flexibility, removing the limit of 64 objects in a method's *literal pool* and of 64 named instance variables.
- make space for introducing *superoperators*, that is, single instructions that replace common sequences of bytecodes in order to save both space and interpretation overhead.

In addition, the compilation of blocks was changed to support their behavior as *closures* and in order to classify them according to the effective need of creating a *closure* as described in section 1.2. References to the availability of activation records as objects were removed[3].

2.1 Registers

The virtual machine has six registers:

- `ip` is the *instruction pointer*, and points to the next bytecode to be executed;
- `sp`, the *stack pointer*, points to the memory cell that currently holds the stack top;
- `thisMethod` holds a reference to the `CompiledCode` object whose bytecode is being executed;
- `self` holds a reference to the object on which the currently executing method was invoked (or, if this is a block, to the value of `self` at the time the block's closure was created).

[3]Many Smalltalk environments, especially those that are more closely related to Smalltalk-80 or to the sample implementation found in [GR83], allow the programmer to access the information in activation records through the instances of `ContextPart` (in particular, through its concrete subclasses, `BlockContext` and `MethodContext`). This feature is useful in particular cases, like for implementing exception handling; however, it is an undesirable constraint for the virtual machine implementor, who might decide to realize their functionality in a different way (for example with special primitives). For this reason, this will not be formally specified.

- `fp`, the *frame pointer*, points to the address on the stack of the first argument passed by the caller.
- `arg`, the *argument register*, is an integer value that holds the argument to the currently executing bytecode.

`thisMethod` and `self` are part of the garbage collector's root set. Other roots are the objects between `fp` and `sp`, and the content of the activation records above the currently executing one.

2.2 Opcodes and operands

The virtual machine bytecodes can be classified in:

- *stack operations* (section 2.4), that manipulate the current activation record, creating and destroying references to the objects used by the program.
- *method invocation and termination* (section 2.5), which transfer control to another activation record, potentially changing the content of all the virtual machine registers.
- *branches* (section 2.6), either conditional or unconditional, which transfer control to another instruction of the same method, without switching to a different activation record.
- *superoperators*. Superoperators represent a sequence of bytecodes, all but one of which have a fixed argument: the argument of the one with variable arguments is contained as usual in the `arg` register.

The operands can come from different sources:

- from the virtual machine's `self` and `arg` registers;
- from the instance variables of `self`;
- from the current activation record, or from the activation records that can be reached through the static chain;
- from the *literal pool* of the currently executing method of block;
- from global variables, through `VariableBinding` objects that are stored in the *literal pool*;
- they can be `true`, `false` and `nil`.

Most of the bytecodes have zero or one operand, which is contained in the `arg` register. A few ones have two operands: the second is held in bits 0-7 of `arg`, the first is held in higher bits.

The interpretation loop is as follows:

```
arg = (arg << 8) | ip[1];
... execute the routine for the bytecode to *ip ...
if (*ip != 55)
  arg = 0;
ip += 2
```

In other words, all bytecodes are two byte long and are formed by an opcode byte and an argument byte. `arg` is cleared by almost all bytecodes, but when

the *extension bytecode* 55 is found, it acts like a shift register: this allows one to encode two-operand instructions or instructions whose argument does not fit a byte.

For example, the instruction `PUSH_INTEGER 1000` is encoded by these four bytes:

- *55* is the opcode for the extension bytecode
- *3* is the most significant byte of the argument 1000
- *44* is the opcode for the `PUSH_INTEGER` bytecode
- *232* is the least significant byte of the argument 1000

This encoding is substantially variable length, but it is specially designed to allow easy decoding of the arguments and to make small arguments cheaper. While it does make two-operand bytecodes more expensive, these are only four in this bytecode set and are relatively rare; moreover, all four of them have at least one very predictable argument, so that most of the time there is a one-operand superoperator that encodes the two-operand bytecodes in a compact way.

Extension bytes can only precede a superoperator *if the variable-argument bytecode is the first in the superoperator* (recall that all but one of the operations in the superoperator have a fixed argument). This constraint simplifies and speeds up the virtual machine, because otherwise the most significant bytes of `arg` would refer to a different bytecode than the least significant byte.

2.3 Limits

Despite the flexibility allowed by variable length encoding, the instruction set does have some limits:

- because of how two-operand bytecodes are encoded, it is impossible to pass more than 256 arguments to a method or block. Actually, the block and method flags described in section 1 have a more restrictive limitation of 32 arguments.
- for the same reason, it is impossible to access variables that require 256 or more hops along the static chain.
- in addition, conditional jumps can only be forward.

2.4 Stack manipulation bytecodes

`PUSH_INSTANCE_VAR` (opcode: 35)
: This bytecode puts on the stack the `arg`-th named instance variable of the object pointed to by `self`. The `sp` register is incremented.

`PUSH_LOCAL` (opcode: 32)
: This bytecode puts on the stack the `arg`-th object in the current activation record (which is pointed to by `fp`). The `sp` register is incremented.

`PUSH_OUTER_LOCAL` (opcode: 33)

This is a two-operand bytecode: the operands are n, the number of the local variable in the activation record, and s, the number of hops in the static chain. s resides in the least significant byte of `arg`, while n resides in higher bits, as described in section 2.2. The interpreter should first find the s-th activation record on the static chain, and then put on the stack the n-th object in that activation record. The `sp` register is incremented.

The case where s is 0 is reserved for future use.

`PUSH_CONST` (opcode: 46)

This bytecode puts on the stack the `arg`-th object in the literal pool. The `sp` register is incremented.

`PUSH_GLOBAL` (opcode: 34)

This bytecode fetches the `arg`-th object from the literal pool, pushes it (incrementing the `sp` register) and sends `value` to it.

`PUSH_SELF` (opcode: 56)

This bytecode puts on the stack the object pointed to by the `self` virtual machine register. The `sp` register is incremented.

`PUSH_SPECIAL` (opcode: 45)

This bytecode puts on the stack one of the three objects `nil`, `true` o `false`, depending on the `arg` register being respectively 0, 1 or 2. The `sp` register is incremented.

`PUSH_INTEGER` (opcode: 44)

Questo bytecode puts on the stack a `SmallInteger` object whose value lies between 0 and $2^{29} - 1$. The `sp` register is incremented.

`POP_INTO_NEW_STACKTOP` (opcode: 47)

This bytecode fetches the stack top and decrements the `sp` register. Afterwards, the object is stored into the `arg`-th instance variable of the object that has become the new stack top. This bytecode is used to initialize arrays.

`STORE_INSTANCE_VAR` (opcode: 39)

This bytecode moves the stack top into the `arg`-th named instance variable of the object pointed to by `self`, without modifying `sp`.

`STORE_LOCAL` (opcode: 36)

This bytecode moves the stack top into the `arg`-th variable of the current activation record, without modifying `sp`.

`STORE_OUTER_LOCAL` (opcode: 37)

This is a two-operand bytecode: the operands are n, the number of the local variable in the activation record, and s, the number of hops in the static chain. s resides in the least significant byte of `arg`, while n resides in higher bits, as described in section 2.2. The interpreter should first find the s-th activation record on the static chain, and then move the current stack top (pointed to by `sp`) to the n-th slot of that activation record. The `sp` register is not modified.

The case where s is 0 is reserved for future use.

`STORE_GLOBAL` (opcode: 38)

This bytecode fetches the `arg`-th object from the literal pool, which shall be an instance of `VariableBinding`; afterwards, it sends `value:` to it, the parameter being the current stack top. The stack top is destroyed and replaced with an undefined object. `sp` is not modified.

`POP_STACK_TOP` (opcode: 48)

This bytecode decrements the `sp` register.

`DUP_STACK_TOP` (opcode: 52)

This bytecode puts on the stack another reference to the object that is currently the stack top, incrementing the `sp` register.

2.5 Bytecode to invoke methods and to return values

`SEND` (opcode: 28)

This bytecode is the most common way to invoke a method. It is a two-operand bytecode: as described in section 2.2, one operand (the number of arguments accepted by the callee) is found in the least significant byte of `arg`, and the other (the index in the literal pool of the method name) is found in the higher bytes.

The receiver and arguments are found on the stack. The virtual machine saves its registers in the current activation record, pops the arguments into a new activation record, and pops the receiver (which is found before the arguments) into the `self` register. Then, the new activation record becomes current and is linked to the previously active one through the dynamic chain.

Starting from the class of `self`, the virtual machine looks for an implementation of the requested method in the method dictionaries of all the classes in the hierarchy. If the method is not found, the arguments are popped again from the new activation record and used to create a `Message` object, which contains the method name and the arguments[4] Then, the search restarts, this time searching for a method named `doesNotUnderstand:`. This method *must* be found (because every root class in the hierarchy must implement it), so the virtual machine can start executing it.

The method that is found can be linked to a primitive, no matter if it is the requested one or `doesNotUnderstand:`. If so, the primitive is executed, and if it succeeds, the virtual machine immediately removes the new activation record and pushes the return value of the primitive, continuing the execution of the caller.

If there is no primitive, or if it fails, `ip` is initialized to point to the first bytecode in the callee, which then starts executing.

`SEND_SUPER` (opcode: 29)

This bytecode is equivalent to the previous one, but it searches the callee starting from the superclass *of the class in which the caller was defined*.

[4]The layout of `Message` objects is not described because they are entirely handled by the virtual machine at run-time, like `BlockClosure`s.

This class can be found in the `MethodInfo` object referred to by `thisMethod` (either directly or, when `thisMethod` is a `CompiledBlock`, indirectly through the `CompiledMethod` that encloses the block).

These semantics extend to the case when an implementation of `doesNotUnderstand:` is searched during the execution of this bytecode. To avoid infinite loops, it is a compile-time error to send messages to `super` from within a root class.

`SEND_IMMEDIATE` (opcode: 30)
: This bytecode acts as a shortcut to limit memory usage: it uses `arg` to encode the names and argument counts for over 200 frequently used messages, in order to avoid wasting slots in the literal pool.

`SEND_SUPER_IMMEDIATE` (opcode: 31)
: This bytecode shares the encoding of `SEND_IMMEDIATE` but the semantics of `SEND_SUPER`.

`SEND_FAST` (opcode: 0...24)
: These 25 bytecodes have the same functionality as the `SEND_IMMEDIATE` bytecode, with an argument between 0 and 24. Their purpose is to allow the implementor to apply special optimization to the most frequently used messages, most of which are responsible for implementing control structures and arithmetics.

  In particular, the virtual machine can directly execute a primitive for the `==`, `isNil` and `notNil` methods (bytecodes 24, 20, 21 respectively), without looking up a `CompiledMethod` in the method dictionaries.

  For bytecodes 0...15, the virtual machine can examine the classes of the receiver and arguments, and execute a primitive without searching the method dictionaries if they belong to the `SmallInteger` class, or to a subclass of `Float`[5]. These correspond to the following selectors:

```
+  -  <   >        <=         >=  =        ~=
*  /  \\  bitXor:  bitShift:  //  bitAnd:  bitOr:
```

  For bytecodes 22 and 23, corresponding to selectors `value` and `value:` the virtual machine can execute a primitive without searching the method dictionaries if the receiver is an instance of `BlockClosure`.

  Bytecodes 16...19, corresponding to selectors `at:`, `at:put:`, `size`, and `class`, cannot be optimized this way; however, it often pays to perform more aggressive caching for these selectors: GNU Smalltalk, for example, memoizes the receiver's class whenever the method is connected to a primitive, and does not search the method dictionary when the receiver's class matches the memoized one. This special kind of cache has an hit rate around 80%.

`RETURN_STACK_TOP` (opcode: 51)
: This bytecode returns the stack top to the caller. The current activation record is removed, and the returned object is pushed on the caller's stack.

[5] [Kra98] defines three concrete subclasses of `Float`, named `FloatE`, `FloatD` and `FloatQ`, which represent different precision levels.

`METHOD_RETURN_STACK_TOP` (opcode: 50)

This bytecode can only be included in `CompiledBlock`s. It returns the stack top to *the enclosing method's caller*: all the activation records below the current one are unwound until the virtual machine finds that of the enclosing method; the returned object is pushed on its caller's activation record.

Since a block is an higher-order function, it may happen that a block executes this bytecode after the enclosing method has terminated its execution. This is a run-time error because blocks are not continuations like those found in Scheme: in this case, this bytecode only unwinds one stack frame (thus behaving like `RETURN_STACK_TOP`) and sends the `badReturnError` to the returned value.

2.6 Jump bytecodes

`JUMP_BACK` (opcode: 40)

This bytecode decrements by `arg` the value of the `ip` register.

`JUMP` (opcode: 41)

This bytecode increments by `arg` the value of the `ip` register.

`POP_JUMP_TRUE` (opcode: 42)

This bytecode first looks at the object which is the top of the stack. If this is `true`, the value of the `ip` register is incremented by `arg`. If this is not a valid boolean value (that is, it is neither `true` nor `false`), the value of the `ip` register is incremented by `arg`, *and* the `mustBeBoolean` method is invoked on that object (as described in the description of the `SEND` bytecode, see section 2.5); that method will usually throw an appropriate exception.

`POP_JUMP_FALSE` (opcode: 43)

This bytecode first looks at the object which is the top of the stack. If this is `false`, the value of the `ip` register is incremented by `arg`. If this is not a valid boolean value (that is, it is neither `true` nor `false`), the value of the `ip` register is incremented by `arg`, *and* the `mustBeBoolean` method is invoked on that object (as described in the description of the `SEND` bytecode, see section 2.5); that method will usually throw an appropriate exception.

All these bytecodes operate *after* `ip` has been updated to point to the following bytecode. For example, in this sequence of bytecodes,

```
 0:  PUSH_SELF
 2:  POP_JUMP_FALSE     8
 4:  PUSH_INTEGER       255
 6:  RETURN_STACK_TOP
 8:  PUSH_INTEGER       0
10:  RETURN_STACK_TOP
```

the value of `arg` that is used in the encoding of the `POP_JUMP_FALSE` bytecode is 4, computed as *the difference between the destination bytecode and the first bytecode after the jump* (in this case, $8 - 4$).

2.7 Miscellaneous bytecodes

`LINE_NUMBER_BYTECODE` (opcode: 54)
: This bytecode does no meaningful operation. Its argument is reserved to be used by the development environment.

`MAKE_BLOCK_CLOSURE` (opcode: 49)
: The virtual machine reads from the stack a `CompiledBlock` object and transforms it into a *closure*. The shape of the returned object is not specified, but should conform to the protocols described by [Kra98]. Its argument is currently unused and reserved for future use[6].

`EXIT_THREAD` (opcode: 53)
: This special bytecode is only found in a single method in the whole system. This method is referenced by the final activation record of every thread in the system. When it is executed, the thread is terminated and the current stack top is saved as the thread's return value.

`EXT_BYTE` (opcode: 55)
: This bytecode is used to encode multi-byte arguments in the instruction stream. A description of its usage can be found in section 2.2.

3 Bytecode verification

Bytecode verification is crucial in an architecture that allows downloading of untrusted code. Actually, even if one could do without it if the code was always made into bytecodes by a trusted compiler, a bytecode verifier is also important as a tool to easily discover bugs in the compiler.

The security risks of downloadable code are obvious both in terms of data integrity (the possibility that important data is modified) and of data confidentiality (the possibility that sensitive information is leaked). Downloaded code makes no insurance of polite behavior, and a virtual machine needs to take every possible measure to avoid bypassing of data hiding and stack inspection: verification allows one to replace run-time checks on the bytecode's behavior with static checks on its well-formedness, that can be run just once per program run.

Examples of the tasks performed by a bytecode verifier are:

- checking that branches do not reach the middle of an instruction encoding, and do not fall outside the method;
- checking that the stack does not underflow or overflow;
- checking for properties that must be the same for all the incoming edges of each basic block (such as stack height or the type of each stack slot);
- checking that local variables are initialized before being used;
- checking that the visibility of instance variables is respected.

[6] For example, it could be the number of stack slots to be popped and placed into indexed instance variables of the closure object.

Depending on the architecture of the virtual machine, some or all of these checks can be performed at execution time instead. For example, Smalltalk's uniform treatment of all types makes its virtual machine strongly typed: out-of-bounds errors need to be checked with a verifier, but type errors will always be discovered by the virtual machine at run-time[7].

The absence of primitive types makes the design of a verifier for Smalltalk bytecodes easier than for Java or CLR bytecodes. A Java bytecode verifier must track the type of each stack slot (either a primitive type, or an array and its items' type, or an object with a particular class) and check it for every bytecode. Instead, the type information in a Smalltalk bytecode verifier is limited to the class and must only be checked for a few specific bytecodes (such as `MAKE_DIRTY_BLOCK`). The simple encoding of bytecodes, described in section 2.2, also contributes to the simplification of the verifier.

Verification will be split between *instruction encoding* invariants that can be checked (almost) independently for each bytecode, and *data-flow* invariants that need a limited amount of data-flow analysis.

3.1 Instruction encoding invariants

The instruction encoding invariants, also called *static constraints* in [LY99], are as follows:

- no references to an instance variable nor to activation records on the static chain may appear in a clean block;
- reads from an instance variable must be in bounds, that is must refer a valid index for a named instance variable (as fetched from the `Behavior` object, whose format is not described in this document);
- writes to an instance variable, additionally, must refer to an instance variable defined by an untrusted class, if the method or block is itself untrusted;
- reads and writes to temporary variables must be in bounds, that is they must not refer more temporaries than the number indicated in the header of the `CompiledMethod` or `CompiledBlock` object;
- writes to a global variable must refer to an untrusted `VariableBinding` object, if the method or block is itself untrusted;
- branches must not land in the middle of a bytecode (the offset may not be odd), and cannot land on a bytecode preceded by an `EXT_BYTE` bytecode;
- a `MAKE_DIRTY_BLOCK` bytecode should be immediately preceded by a push of a `CompiledBlock` object;
- the argument to the `PUSH_SPECIAL` bytecode must be 0, 1, or 2;
- the `EXIT_INTERPRETER` may only appear in a method together with a `RETURN_CONTEXT_STACK_TOP` bytecode (i.e. in a method that has exactly these two bytecodes, in this order);

[7] Note that the JVM is not completely exempt from such checks, since it has to check at run-time for `null` receivers or invalid casts.

- undefined bytecodes may not appear in the method.

3.2 Data-flow invariants

The data-flow invariants, also called *structural constraints* in [LY99], are as follows:

- the stack height must be the same for all the incoming edges of a basic block;
- the stack height must never fall below zero (stack underflow) or above the number of stack slots indicated in the `CompiledMethod` or `CompiledBlock` objects;
- the `POP_INTO_NEW_STACKTOP` may only pop into an `Array` object, created with a sequence like

  ```
  push global variable Array
  push an integer with a PUSH_INTEGER bytecode
  send #new:
  ```

  and must be in bounds (according to the argument of the `PUSH_INTEGER` bytecode in the array creation sequence).

References

[App98] Andrew W. Appel, *Modern Compiler Implementation in Java*, Cambridge University Press, 1998.

[GR83] Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, 2nd ed., Addison-Wesley, 1983.

[Kra98] Glenn Krasner, *American National Standard for Information Technology Programming Languages Smalltalk*, Tech. Report ANSI NCITS 319-1998, ANSI Technical Committee J20, 1998.

[LY99] Tim Lindholm and Frank Yellin, *The Java(tm) Virtual Machine Specification*, 2nd ed., Addison-Wesley, 1999.

[Wol87] Mario Wolczko, *Semantics of Smalltalk-80*, in Proceedings of the European Conference on Object-Oriented Programming, Springer-Verlag, 1987.