Digitalk Bytecodes

Leandro Caniglia, Valeria Murgia & Gerardo Richarte

December 13, 2008

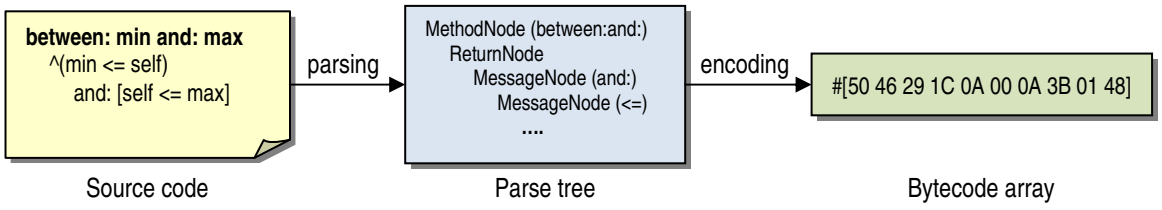
This document provides a broad description of the intermediate language that sits between regular Smalltalk *source code and natively assembled routines. Unlike usual programming languages, this one is read-only in the sense that the programmer can look at it in mnemonic form while its codification is left to the* Smalltalk *compiler. The value of understanding Smalltalk bytecodes is better appreciated by those interested in deepening their understanding of topics normally beyond the scope of standard programming: processes and activation frames. However focused on the specific set of bytecodes of* Digitalk *virtual machines, the concepts and techniques covered here can be applied to other* Smalltalk *dialects.*

Compilation

The *Smalltalk* compiler transforms source code into instances of the class CompiledMethod. In this process several classes collaborate to produce the desired result: CompilerInterface, SmalltalkCompiler, Scanner, ParseNode and Binding hierarchies, and Encoder.

The CompilerInterface creates the SmalltalkCompiler instance, takes care of compilation options and exception handlers (compilation errors). The SmalltalkCompiler parses the source code creating a Scanner that transforms the source code String into a sequence of *tokens*. These tokens are then *analyzed* according to the *Smalltalk grammar* (syntax). The result of the analysis is expressed as a MethodNode, the root of a nested structure of ParseNodes including objects that represent arguments, message sends, variables, assignments, blocks, etc.

Once the parse tree has been successfully created, the objects collaborate to generate the byteCodeArray of the CompiledMethod. The SmalltakCompiler creates and holds an Encoder that *emits* sequences of binary encoded *bytecodes*. Bytecodes may have a size of one or more bytes; they represent ParseNodes as parameterized assembly-like instructions. The byteCodeArray is a ByteArray that holds the sequence of binary encoded bytecodes.



Native code

Bytecodes are an intermediate representation of computable code. In order to make this code run on the host CPU, the Virtual Machine (VM) translates them into *native code*. The part of the VM that *nativizes* bytecodes is called the *Dynamic* or *JIT* (Just in Time) *compiler*. Even though a comprehensive description of the Dynamic compiler is beyond the scope of this document, its conceptualization is simple: native code is dynamically generated by the VM from method bytecodes.

Because the receiver of a message is only known when the message is about to be sent, the binding between the message selector and the actual method may potentially change at every invocation. The lookup, therefore, may result in a method whose native form is not available. When that is the case, the *JIT* has to nativize the method on the fly. For the sake of performance the VM maintains a *code cache* that holds methods that have already been nativized. The class VirtualMachineExe has services that allow the *Smalltalk* programmer to monitor the state of the code cache or change its size.

Bytecodes

Bytecodes are binary specifications of instructions. Every bytecode consists of one or more bytes. The first byte determines the *opcode*. In some cases, the first byte also includes a parameter codified in the last few bits. Additional bytes, if present, contain parameters required by the opcode. A complete list of bytecodes can be examined by inspecting the PoolDictionary named XCByteCodes.

The class ByteCodeReader uses this dictionary to recover the structure of the byteCodeArray and provide human-readable printing services (see ByteCodReader >> #displayMethod: and its variants).

For instance, the method Magnitude >> #between:and: has the following bytecodes:

Bytecode (hex)	**Mnemonic**	**1st argument**	**2nd argument**
50	LoadArgument1		
46 29	SendBinaryPseudo	<=	self
1C 0A 00	TestJumpFalse	10	0
0A	LoadSelf		
3B 01	SendBinaryLessThanEqual	Arg	1
48	Return		

Note that the binary encoding of arguments is not restricted to octets. For instance, SendBinaryPseudo and SendBinaryLessThanEqual encode their two arguments in bit fields of the same byte while TestJumpFalse takes two bytes to encode the jump length. The argument of LoadArgument1 is implicitly encoded in the opcode.

Most bytecodes are grouped in families according to their function. Bytecodes in the same family share the same opcode and use their first byte to specify an implicit argument. The last bytecode of the family assigns an additional byte to specify an argument that exceeds the max implicit value.

For example, the LoadArgument family is:

Bytecode (hex)	**Mnemonic**	**Implicit argument**	**Argument**
50	LoadArgument1	1	
51	LoadArgument2	2	
52	LoadArgument3	3	
53	LoadArgument4	4	
54 xx	LoadArgumentN		xx (>= 5)

The Encoder uses the method #putCode:max:range: to generate bytecodes for the opcode specified in the first argument. The range argument is the index parameter. If the range argument does not surpass the max, it is encoded in the first byte by subtracting max – range + 1 from the opcode. If not, a two-byte bytecode is generated instead, with the opcode in the first byte and the range argument in the second.

In the example above, the opcode associated to LoadArgument is 16r54 and the max is 4. Therefore, LoadArgument2 is generated by sending putCode: 16r54 max: 4 range: 2, which emits the single byte bytecode 16r54 – (4 – 2 + 1) = 16r51.

Storage

Native code accesses data allocated in different places:

a) *Execution Stack*
b) *CPU Registers*
c) *Environment arrays*
d) *Compiled method's literal frame*
e) *Receiver's instance variables*

Execution Stack

The Encoder honors the calling convention that consists in passing method arguments in the stack. To support this convention the Encoder emits the appropriate *push* bytecodes before emitting *send* ones.

Nativized methods also follow the same convention. The *JIT* compiler translates method bytecodes into machine code that pushes arguments on the execution stack. The receiver of the message is loaded in a CPU register as explained below.

Besides passing parameters by *pushing* them into the stack, *temporary variables* are also allocated in the stack. For instance, the method

```
String >> #\ aString
    | slash |
    slash := FileSystemPath directorySeparator.
    ^(self endsWith: slash)
        ifTrue: [self , aString]
        ifFalse: [self , slash, aString]
```

has one temporary named slash. Because of this, the Encoder generates bytecodes to access it in a specific stack location. For instance, the assignment is encoded as

```
<C3>    StoreTemporary1
```

which stores the object held in the R register (see below) into the first (and only) stack-allocated temporary variable.

CPU Registers

The Encoder explicitly refers to register R, which is mapped to a CPU register. As documented in [1] this register must always hold the receiver of the message that is about to be sent. When the message returns, register R holds the result.

Methods are always activated with register R loaded with their receiver (self). Because of this, a method that begins by sending a message to the receiver has no need to load R with self. For instance, the method

```
Symbol >> #arity
    | last |
    last:= self last.
    last = $: ifTrue: [^self occurrencesOf: $:].
    (last isAlphaNumeric or: [last == $_]) ifFalse: [^1].
    ^0
```

has an encoding that begins with:

```
<E2>    SendSelector1    #last
```

because it assumes that the caller of the nativized method will initialize R to self.

Another register known to the Encoder is the *argument* register A. This register is used by the following bytecodes:

Bytecode (hex)	**Mnemonic**	**Description**
09	MoveRegToArg	A ← R
20	SendBinaryRRPlus	R ← R + A
23	SendBinaryRRMinus	R ← R – A
26	SendBinaryRRMultiply	R ← R * A
28	SendBinaryRRDivide	R ← R / A
2B	SendBinaryRREqual	R ← R = A
2F	SendBinaryRRQuotient	R ← R // A
32	SendBinaryRRNotEqual	R ← R ~= A
37	SendBinaryRRLessThan	R ← R < A
3A	SendBinaryRRGreaterThan	R ← R > A

Environment arrays

Attention must be paid not to confuse *Smalltalk* temporaries with *stack* temporaries. Stack temporaries are momentarily allocated in the execution stack. These variables are local to the native routine. A *Smalltalk* temporary is the familiar local variable declared between

pipes '|'. However, not every method or block temporary can be allocated in the stack. The reason is that method temporaries accessed from block closures have to survive as long as the block does. Since the execution stack vanishes as soon as the method returns, chances are that a block survives the method that defined it (for example, a method returning a sortBlock that uses a method temporary to define the order criterion). Therefore, local variables used inside a block cannot be allocated in the stack and must be preserved as long as the block is alive. As an example, consider the following method:

```
Collection >> #select: selectBlock thenCollect: collectBlock
    | answer |
    answer := self species new.
    self do: [:element |
        (selectBlock evaluateWith: element)
            ifTrue: [answer add: (collectBlock evaluateWith: element)]].
    ^answer
```

This method defines the local (temporary) variable answer. However, since this variable is accessed inside the block closure that is provided as an argument to the #do: message, it cannot be allocated in the stack. In fact, if we evaluate (Collection compiledMethodAt: #select:thenDo:) tempCount, the value we get is 0. The local variable answer is instead allocated in a *Smalltalk* Array known as the method's *environment*. The block closure holds a reference to that array at position 4, and can be accessed with the method BlockClosure >> #methodEnvironment.

The same thing happens when a block temporary is used by nested blocks. In general, environment arrays are just the collection of all local variables that are used from within inner blocks.

Literal Frame

Many bytecodes reference their parameters indirectly by their position in the method's literal frame. For instance, the SendSelector pattern includes the following bytecodes:

Bytecode (hex)	**Mnemonic**	**Implicit argument**	**Argument**
E2..F6	SendSelector1..21	1..21	
F7 xx	SendSelectorN		xx (>= 22)

In all cases, an argument k, implicit or not, specifies the actual selector to be sent as the one stored in slot n – k + 1, where n is the size of the literal frame (i.e., indexes to selectors are in reverse order).

The literal frame can also contain associations binding names with *shared*[1] objects. These objects are also referenced in bytecodes by their position (or index).

Bytecode (hex)	**Mnemonic**	**Implicit argument**	**Argument**
5A..61	LoadAssoc1..8	1..8	
62 xx	LoadAssocN		xx (>= 9)

[1] Global and class variables

63..67	PushAssoc1..5	1..5	
68 xx	PushAssocN		xx (x >= 6)
69 xx	StoreAssocN		xx (x >= 1)

For instance, if a method sends a message to a class, its literal frame will have an association whose key is the class symbol and its value is the class.[2]

The literal frame also holds other objects created at compilation time from the source code such as, e.g., 'string', $c, #symbol, #[0 1 255], #(0 1 255), 3.141592, 1073741824 (LargeIntegers[3]), and their combinations like #(#[0 1 255] $c 'string'), etc. The following table lists the bytecodes that operate with entries stored in the literal frame. Arguments of these bytecodes are indexes within literal frame.

Bytecode (hex)	**Mnemonic**	**Implicit argument**	**Argument**
9E..A1	LoadLiteral1..4	1..4	
A2 xx	LoadLiteralN		xx (>= 5)
A3..AD	PushLiteral1..11	1..11	
AE xx	PushLiteralN		xx (>= 12)

In addition, for every block in the method there is one entry in the literal frame that holds its *template*. Block templates are instances of the class BlockClosure, not tied to any method, and serve the purpose of describing the block that will eventually be created when the method is activated.[4]

Instance variables

There are three bytecode patterns to access instance variables. The first pattern loads the appropriate instance variable into register R. The second pattern pushes the instance variable on the stack. The third one copies the contents of R on the instance variable.

Bytecode (hex)	**Mnemonic**	**Implicit argument**	**Argument**
7F..8A	LoadInstance1..12	1..12	
8B xx	LoadInstanceN		xx (>= 13)
8C..94	PushInstance1..9	1..9	
95 xx	PushInstanceN		xx (>= 10)
96	StoreInstance1..7	1..7	
9D xx	StoreInstanceN		xx (>= 8)

The pattern LoadInstance is used to send a message to the corresponding instance variable.

[2] Note however that no restriction exists on the contents of the literal frame. Its slots can contain any object and this possibility is sometimes exploited to implement non-standard features, see e.g. [2].

[3] SmallIntegers are codified inside bytecodes rather than in the literal frame.

[4] The method BlockClosure class >> #initStandardTemplates initializes the most common block templates used by compiled methods. Note however, that many compiled methods use non-standard block templates.

Instance variables are also referenced by bytecodes associated to some binary selectors that use them as arguments[5]. The following is a list of such binary selectors:

BitAnd	Equal	Divide
BitOr	GreaterThan	Minus
BitShift	GreaterThanEqual	Multiply
BitXor	Identical	Plus
Or	LessThan	Quotient
	LessThanEqual	Remainder
	NotEqual	

Stack Allocation

As we have seen, the stack plays a central role in the *Smalltalk* execution model. Arguments are passed by pushing them on the stack, and the stack is also used for temporary storage. A good understanding of the execution model involves a good understanding of how the stack is allocated.

Traditionally, the *Smalltalk-80* specification used the stack to push not only message arguments but also the receiver and the answer. As explained in [1], modern VMs use a register for the receiver and, sometimes, another register for the answer. What did not change, however, is that the called method has the responsibility to remove the arguments from the stack before returning.

In this section we will see how the Encoder supports the execution model by looking more closely at how bytecodes make use of registers and the stack.

The expression

```
CompiledMethod byteCodeReaderClass displayMethod: aCompiledMethod on: aStream
```

prints in mnemonic form the byteCodeArray of aCompiledMethod on aStream. For instance, the method

```
Magnitude >> #between: min and: max
    ^(min <= self) and: [self <= max]
```

is printed as:

```
Magnitude>>between:and:
    type: 0
    blocks: 0
    args: 2
    stk temps: 0
 1  <50>  LoadArgument1
 2  <46>  SendBinaryPseudo  <= self
 4  <1C>  TestJumpFalse   10
 7  <0A>  LoadSelf
 8  <3B>  SendBinaryLessThanEqual    Arg,   2
```

[5] These bytecodes belong to the SendBinary family and they can also reference arguments of binary type listed in Encoder subclasses first binaryTypes

```
10 <48>  Return
```

It is interesting to note how easy it is to make sense of the mnemonic form:

Index	Bytecode (hex)	Mnemonic	1st argument
1	50	LoadArgument1	R ← arg1 (min)
2	46 29	SendBinaryPseudo <= self	R ← R <= self
4	1C 1A 00	TestJumpFalse	if R == false goto 10
7	0A	LoadSelf	R ← self
8	3B 01	SendBinaryLessThanEqual Arg,2	R ← R <= arg2 (max)
10	48	Return	^R

With practice it is very easy to translate the mnemonic representation to the *Smalltalk* syntax. This example also shows how #and: is *inlined* rather than sent as a message. Other examples of inlined selectors are ifTrue:, ifFalse:, whileTrue:, etc.

What this example does not show is the utilization of the stack for storage local to the method. Let's consider instead:

```
Character >> #between: min and: max
    ^(min asciiValue <= asciiInteger) and: [asciiInteger <= max asciiValue]
```

The mnemonic representation of this method is as follows:

```
Character>>between:and:
    type: 0
    blocks: 0
    args: 2
    stk temps: 0
    literals:
    asciiValue
 1  <50>  LoadArgument1
 2  <E2>  SendSelector1   #asciiValue
 3  <3B>  SendBinaryLessThanEqual    Inst,   1
 5  <1C>  TestJumpFalse   15
 8  <8C>  PushInstance1
 9  <51>  LoadArgument2
10 <E2>  SendSelector1   #asciiValue
11 <09>  MoveRegToArg
12 <07>  PopR
13 <3B>  SendBinaryLessThanEqual    A-reg,1[6]
15 <48>  Return
```

The interesting thing here is the PushInstance1 bytecode in line 8. This instruction is used to *preserve* in the stack the value of the first (and sole) instance variable asciiInteger. This ivar

[6] In this case the number 1 in A-reg, 1 is superfluous because there is only one register A. The mnemonic displays it anyway because the same routine is used for numbered arguments like instance variables, temporaries, literals, etc.

is recovered in line 12 with bytecode PopR that moves it from the stack to register R, where it is required as the receiver of the binary message <=.

Note that the ivar is preserved to prevent a possible side effect of the #asciiValue message. The semantic of asciiInteger <= max asciiValue requires the value held by asciiInteger to be "protected" of any change that the message #asciiValue could potentially inflict on it. In this way the programmer can intuitively think of *Smalltalk* expressions as being evaluated from left to right.

This example shows that even though the method defines no stack temporary, the execution will dynamically allocate an *implicit* stack temporary between lines 8 and 12. We can illustrate this by adding a new column on the left showing the depth of the stack *after*[7] executing the bytecode in that line.

Character>>between:and:

```
    type: 0
    blocks: 0
    args: 2
    stk temps: 0
    literals:
    asciiValue
(0)  1  <50>  LoadArgument1
(0)  2  <E2>  SendSelector1   #asciiValue
(0)  3  <3B>  SendBinaryLessThanEqual   Inst,   1
(0)  5  <1C>  TestJumpFalse   15
(1)  8  <8C>  PushInstance1
(1)  9  <51>  LoadArgument2
(1) 10 <E2>  SendSelector1   #asciiValue
(1) 11 <09>  MoveRegToArg
(0) 12 <07>  PopR
(0) 13 <3B>  SendBinaryLessThanEqual   A-reg,1
(0) 15 <48>  Return
```

As we can see there are two kinds of stack allocations: *static* and *dynamic* ones. Static allocations make room for method arguments and stack temporaries. In addition, dynamic allocations are required to preserve objects from possible side effects inflicted by intermediate messages that compute arguments.[8]

Note that pushing arguments according to the calling convention is not considered as a stack allocation in the context of this discussion. The point here is to show that the stack might, at some point in time, hold more temporaries than the number of stack temporaries of the method as specified by CompiledMethod >> #tempCount (which equals the stk temps: header section of the mnemonic display).

---

[7] The fact that the depth shown on the *left* corresponds to the value obtained *once* the bytecode has been executed is admittedly counterintuitive. We choose this format simply because it was simpler to derive from the original one.

[8] Static temporaries are allocated for as long as the method is in the stack. Dynamic ones are allocated and de-allocated during the execution of the method. In both cases, however, stack temporaries are accessible through bytecodes of the LoadTemporary family.

Example

Another example where an implicit stack variable is dynamically allocated to preserve its value is the following:

```
method
    self unary1 keyword: self unary2
```

A naïve encoding would look like this:

```
1  <E3>   SendSelector2    #unary2
2  <06>   PushR
3  <0A>   LoadSelf
4  <E2>   SendSelector1    #unary1
5  <E4>   SendSelector3    #keyword:
```

However, after some analysis it is easy to understand that this encoding could give puzzling results (i.e., results not expected by the *Smalltalk* programmer who is used to left-to-right evaluation).

Since this encoding sends #unary2 before #unary1, confusion would raise if #unary2 had some side effect on the result of #unary1. For instance, #unary1 answers an instance variable, while #unary2 changes its value. Before the presence of such side effect, the *Smalltalk* programmer expects #unary2 to be sent after #unary1 (in our example, the original ivar to be the receiver of #keyword:).

The correct encoding (which is the actual one) avoids this problem by preserving the result of #unary1 dynamically as an implicit stack temporary:

```
(0) 1  <E2> SendSelector1    #unary1
(1) 2  <06> PushR
(1) 3  <0A> LoadSelf
(1) 4  <E3> SendSelector2    #unary2
(2) 5  <06> PushR
(2) 6  <AF> LoadTemporary1
(1) 7  <E4> SendSelector3    #keyword:
(0) 8  <03> DropTos1
(0) 9  <49> ReturnSelf
```

As before, we have included the stack depth after executing each bytecode. In line 2: PushR dynamically creates the implicit temporary that saves the result of #unary1. This value is recovered in line 6: LoadTemporary1, which sets register R to that value for using it as the receiver of #keyword:. Note that the bytecode in line 6 treats the access to the dynamic allocation as a regular stack temporary. The stack is balanced back to 0 in line 8: DropTos1, a bytecode that simply discards the top of the stack.

The reader should note that the purpose of line 5: PushR is different from that of line 2 as the second push is required by the calling convention to pass the argument to the message #keyword:.

Stack Tracing

The tracing of the stack depth explained above can be automatically computed for any method. The idea is to keep track of bytecodes that push on or pop from the stack. There are some few special cases to consider:

Explicit Push

There are several bytecode patterns that explicitly push objects on the stack:

Bytecode (hex)	**Mnemonic**	**Depth change**
06	PushR	+1
0B	PushSelf	+1
0F	PushTrue	+1
11	PushFalse	+1
13	PushSmallInteger0	+1
15	PushSmallInteger1	+1
17	PushSmallInteger2	+1
19	PushSmallInteger	+1
4F 03	LoadBlockContextN	+1
55..59	PushArgument1..N	+1
63..68	PushAssoc1..N	+1
72..77	PushContextTemporary1..N	+1
8C..95	PushInstance1..N	+1
A3..AE	PushLiteral1..N	+1
BC..C2	PushTemporary1..N	+1
FC 05..08	PushIndirect (IndirectEscape)	+1

Explicit Pop

The following bytecode patterns explicitly pop objects from the stack:

Bytecode (hex)	**Mnemonic**	**Depth change**
03	DropTos1	-1
04	DropTos2	-2
05 b	DropTosN	-b
07	PopR	-1
CB..E0	SendSpecial1..22	-arity
E1	SendSuperSpecialN	-arity
E2..F7	SendSelector1..N	-arity
F8..F9	SendSuper1..N	-arity

In this table arity stands for the number of arguments of the implied selector. Patterns SendSelector and SendSuper refer to selectors found in the literal frame. Their implicit or explicit argument determines the reverse index in that array, 1 being the last one. The

special selectors are those answered by the method ByteCodeReader class >> #specialSelectors. An implementation of Symbol >> #arity was included above in this document.

Note that the table above does not include any bytecodes of the SendBinary family. There are two reasons for this. First, this family includes the RR subfamily that operated directly with registers R and A, as already explained in this document. Secondly, the remaining bytecodes of the family use the binary encoding within the bytecode, rather than the stack, to define the argument of the binary operation.

Block closures

The third group that deserves special attention is that of block patterns. These bytecodes mark the beginning and end of block closures. Bytecodes inside a block do not alter the depth of the current stack because their execution will happen when the block (rather than the home method) is activated. Therefore, inside the block, the stack depth must be reset to 0, and the original value reestablished as soon as the block ends. For instance:

```
method
    self unary1 do: [self unary2]
```

has the following encoding:

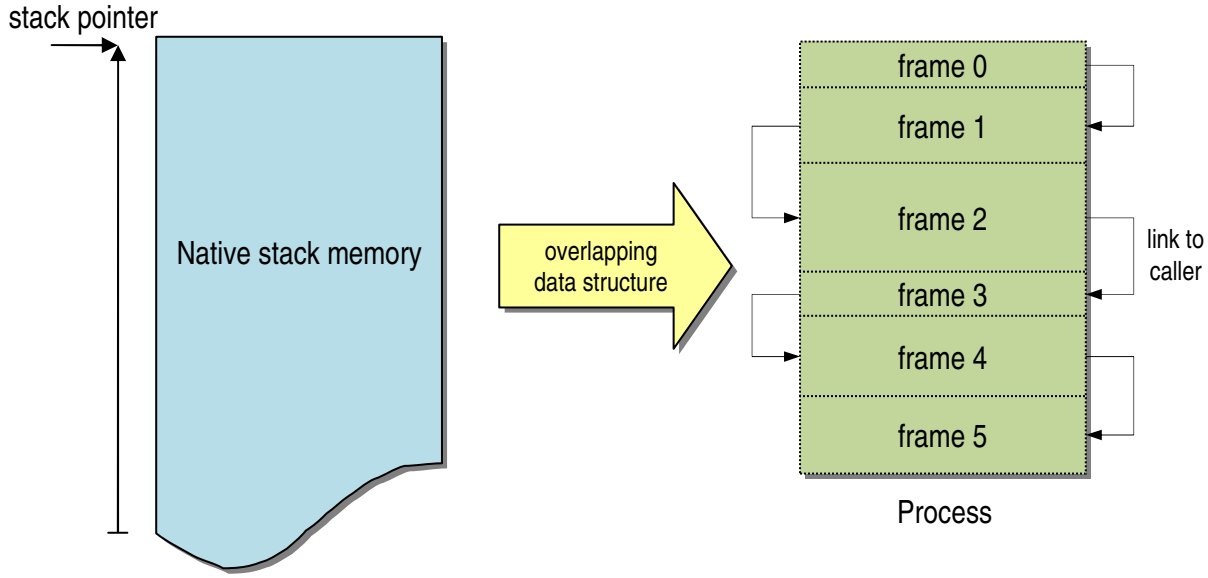
```
(0)  1  <4F>  PushBlockclosure    1, 9
(0)  6  <0A>  LoadSelf
(0)  7  <E2>  SendSelector1    #unary2
(1)  8  <4A>  ReturnFallOutBlock
(1)  9  <0A>  LoadSelf
(1) 10 <E3>  SendSelector2    #unary1
(0) 11 <D3>  SendSpecial9     #do:
(0) 12 <49>  ReturnSelf
```

In the leftmost column we have annotated all changes to the stack depth as before. The point here is that lines 1 to 8, corresponding to the block [self unary2], reset the depth to 0 and recover it as soon as the block exits (ReturnFallOutBlock). Even though PushBlockclosure increments the method's stack depth in 1, its effect is not seen inside the block. After executing line 8: ReturnFallOutBlock, the method's stack recovers its depth.

Note also that the PushBlockclosure bytecode includes as an argument the index to the first bytecode following the block (9 in this case). By using this parameter it is easy to know when the original depth counter must be recovered.

Stack Frames

Digitalk VMs allocate the current *Smalltalk* process directly in the CPU execution stack (a.k.a the *native* stack). The class Process is a subclass of OrderedCollection that adds several instance variables whose meaning is beyond the scope of this document. Entries (a.k.a. *slots*) of this collection are objects[9] consecutively allocated in the (native) stack. These slots are logically grouped in data structures known as (process) *frames*.



As shown in the illustration the execution stack is divided into consecutive frames, which are data structures linking the activation of a method with that of its *caller* and *callee*. Stack frames are described in [1]. The structure of stack frames in a *Digitalk* VM can be understood by browsing the class Process.

The size of every frame depends on both static and dynamic information. Statically defined slots depend on the number of arguments and the number of stack temporaries. All other slots are common to all frames: receiver (self), environment, return offset, etc. Dynamically defined slots are those that depend on the number of implicit stack temporaries as explained above.

Even though a complete description of stack frames is beyond the scope of this document, the following table shows the structure of a typical method frame:

Name	Description	Static?
Implicit N	Dynamically allocated temporary N (if any present)	No
…	…	No
Implicit 1	Dynamically allocated temporary 1 (if any present)	No
Temporary N	Statically allocated stack temporary N (if any present)	Yes
…		
Temporary 1	Statically allocated stack temporary 1 (if any present)	Yes

[9] Actually *object pointers* (a.k.a. oops). These are pointers to *Smalltalk* objects, with the exception of SmallIntegers which are codified inside their oop.

Environment	Array, Block, nil or missing	Yes
Home	Array, Block, nil or missing	Yes
CompiledMethod	The compiled method	Yes
Receiver	self	Yes
Frame pointer	Distance in bytes to the caller frame (=0 if no caller)	No[10]
Return offset	Caller's nativized program counter	Yes
Argument 1	Argument 1 (if any present)	Yes
…		
Argument N	Argument N (if any present)	Yes

As we have seen in the preceding paragraphs, the number of dynamically allocated stack slots Implicit1..ImplicitN can be determined by tracing the stack depth in the mnemonic representation of the bytecode array. In the next section we show how this issue can be addressed.

Dynamically Allocated Temporary Count

An interesting problem that we will consider now is how to compute the number of dynamically allocated temporaries in a frame.[11] As we have seen these temporaries are implicit in the sense that they are not declared by the programmer. Instead, stack temporaries are generated by bytecodes that push on the stack objects that need to be preserved until they can be used as receivers or arguments of forthcoming messages.

As we have already seen, by scanning the bytecodes it is possible to trace all changes in the stack depth. The idea of such tracing consists in keeping track of pushes and pops. It also requires saving the current depth counter when entering a block, and recovering it once the block is left. Inside blocks the depth counter has to be reset to 0.[12]

Once the stack tracing is available, we can determine the number of implicit temporaries as the *current* stack depth computed by the tracing. All we need to know is the index of the *current* bytecode, i.e., the index in the byteCodeArray of the bytecode corresponding to the frame under analysis. Let's call that index the frame *instruction counter*.

The frame itself does not know its instruction counter. However, its callee keeps in its *return offset* field, the native program counter where the callee must return. This value is the offset within the caller of the next native assembly instruction.

[10] Since the size of every frame depends on both static and dynamic allocations, the link from one frame to its callee cannot be considered static.

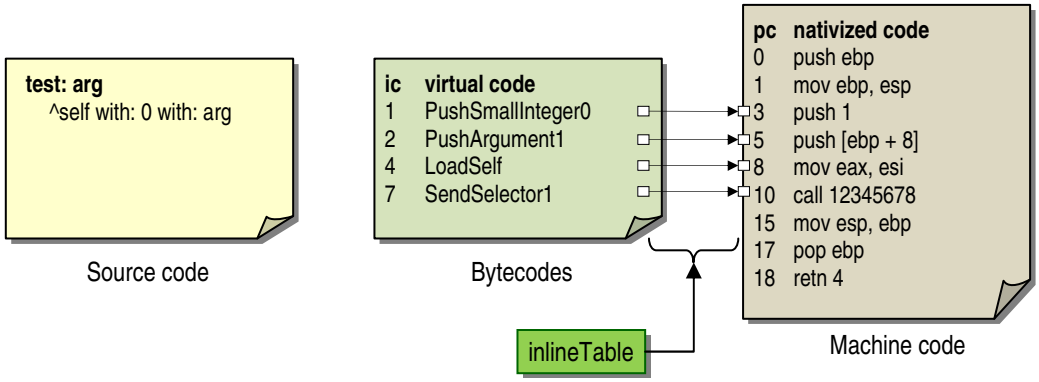
[11] Of course, instances of Process know this information. However, the purpose of this section is to show how to deduce it in order to create new processes or modify existing ones in a way that is valid and coherent with native code.

[12] Since blocks can be nested, this can be accomplished by using a stack-like structure. When scanning a bytecode that enters a block, the current depth can be pushed and a new one started with 0. The old value can be recovered by popping it from the stacked structure. All of this can be easily implemented in a subclass of ByteCodeReader.

Therefore, all we need is a mapping relating the bytecode instruction counter to the native program counter. Fortunately, <primitve 14> provides exactly that mapping. The method

```
CompiledMethod >> #inlineTable
    ^CompilationInfo new compiledMethod: self; inlineInfo
```

answers an array whose values are native program counters (actually offsets), and its indexes are instruction counters in the byteCodeArray.[13]



Actual machine code might differ from the one in the illustration, which is irrelevant for this discussion. Similarly, the translation from bytecodes to machine code is not, in general, one-to-one because most bytecodes spawn to several assembler instructions. What is important here is how the inlineTable can be used to associate to every frame with its instruction counter, from there with the stack depth change that corresponds to it, and finally with the number of implicit temporaries in the frame.

Conclusions

The study of bytecodes and the intermediate language they define are crucial to understand the underlying architecture of the *Smalltalk* computational model. These notes are nothing but an attempt to fill documentation gaps and enable deeper analyses of method activations and *Smalltalk* processes.

References

[1] A SMALLTALK VIRTUAL MACHINE ARCHITECTURE MODEL – Allen Wirfs-Brock, Paul Caudill. 1999.

[2] SIMULATING METHOD WRAPPERS IN VISUAL SMALLTALK – L. Caniglia & V. Murgia. July 21, 2007.

[13] CompilationInfo is the only subclass of CompilationResult. Therefore it can be referenced as CompilationResult subclasses first.