

CS652 Smalltalk VM Operational Semantics

Terence Parr

April 11, 2017

Smalltalk	Context stack at ↩
<pre>"Test testEvalReturnBlock" class T [f [x x := 1. ^[x := 5]]] t := T new. t f value ↩</pre>	<p>Start send 0, 'value':</p> $main[-, nil, -] \ f-block0[-, ,]$ <p>Notes: no f on stack during eval of $f-block0$ but enclosing scope of $f-block0$ still points at f's BlockContext.</p>
Smalltalk	Context stack at ↩
<pre>"Test testRemoteMethodCanSetMyLocal" class T [f [x self g:[x := 5 ↩]] g: blk [blk value]] T new f</pre>	<p>@store_local $\Delta=1, i=0$:</p> $main[-, ,] \ \underbrace{f[-, nil,] \ g[-, f^{block_0},] \ f^{block_0}[-, , 5]}_{\leftarrow \text{enclosing context } \Delta=1}$ <p>After block_return:</p> $main[-, ,] \ f[-, 5,] \ g[-, f^{block_0}, 5]$
Smalltalk	Context stack at ↩
<pre>"Test testRemoteReturn" class T [f [self g:[^99]] g: blk [blk value ↩]] t t := T new. t f</pre>	<p>Start send 0, 'value':</p> $main[-, t,] \ f[-, ,] \ g[-, f^{block_0},] \ f^{block_0}[-, ,]$ <p>After return in $[^99]$ block:</p> $main[-, t, 99]$ <p>Notes: Despite eval in g, $[^99]$ unrolls stack to $main$, the caller of f.</p>

$T \bowtie x$	Resolve x in scope T
$o \in X$	o is instance of X
$v \in \text{STObject}$	a single object
$l_i \in \text{STObject}$	the i^{th} argument or local variable object
$o_{\text{class}} \in \text{STMetaClassObject}$	Metaclass (type) of object o
$o_{\text{class}_{\text{class}}} = o_{\text{class}}$	A metaclass object is its own type
$o_{\text{superclass}} \in \text{STMetaClassObject}$	Superclass (type) of object o
o_{field_i}	The i^{th} field of object o
f_{literal_i}	The i^{th} literal of method f
$f_s^{\text{block}_i} \in \text{BlockDescriptor}$	The i^{th} block of method f associated with instance $\text{self}=s$
$f_s^{\text{block}_i}[-, -, -] \in \text{BlockContext}$	The i^{th} block of method f invoked with $\text{self}=s$
$f_s^{\text{block}_i}[-, -, -]^d \in \text{BlockContext}$	The i^{th} block of method f invoked with $\text{self}=s$ and having depth d counting from zero at the method block; e.g., <code>f [x [y]]</code> has a method block at depth 0 with <code>x</code> and a nested block at depth 1 with <code>y</code>
$\gamma \in \text{MethodContext}^*$	Stack of method invocations growing to the right
$\delta \in \text{STObject}^*$	Operand stack of objects growing to the right
\mathbb{S}	The state of the VM system dictionary
(\mathbb{S}, γ)	VM state is the system dictionary and a method invocation stack with zero or more elements
$(\mathbb{S}, \gamma) \Rightarrow (\mathbb{S}', \gamma')$	VM state transition
$(\mathbb{S}, \gamma) \Rightarrow^* (\mathbb{S}', \gamma')$	Zero-or-more state transitions
$f_s[ip, l_0, ..l_{n-1}, \delta]$	Method invocation context that derived from sending message f to receiver s (self); $f \in \text{MethodContext}$; l_i is local variable or argument, indexed from 0 and arguments first; δ is the operand stack; <i>f can also represent a nested code block not just a method</i>
$f[ip, l_0, ..l_{n-1}, \delta]$	Same as previous but the receiver is unknown or irrelevant
$f[ip, -, -]$	A method invitation context with “don’t care” for locals and operand stack

Figure 1: Smalltalk VM Bytecode Specification Notation

Bytecode Instruction	Transition
<i>initial state</i>	$state_0 = (\mathbb{S}[\text{Transcript}], \text{main}_m[0, \epsilon, \epsilon])$ for $m \in \text{MainClass}$; program terminates if $\exists state_0 \Rightarrow^* (\mathbb{S}', \epsilon)$
<code>nil</code>	$(\mathbb{S}, \gamma f[ip, -, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip + 1, -, \delta \text{ nil}])$
<code>self</code>	$(\mathbb{S}, \gamma f_s[ip, -, \delta]) \Rightarrow (\mathbb{S}, \gamma f_s[ip + 1, -, \delta s])$
<code>true</code>	$(\mathbb{S}, \gamma f[ip, -, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip + 1, -, \delta \text{ true}])$
<code>false</code>	$(\mathbb{S}, \gamma f[ip, -, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip + 1, -, \delta \text{ false}])$
<code>push_char c</code>	$(\mathbb{S}, \gamma f[ip, -, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip + 3, -, \delta c])$
<code>push_int i</code>	$(\mathbb{S}, \gamma f[ip, -, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip + 5, -, \delta i])$
<code>push_float i</code>	$(\mathbb{S}, \gamma f[ip, -, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip + 5, -, \delta \text{ intBitsToFloat}(i)])$
<code>push_field i</code>	$(\mathbb{S}, \gamma f_s[ip, -, \delta]) \Rightarrow (\mathbb{S}, \gamma f_s[ip + 3, -, \delta s_{field_i}])$
<code>push_local 0, i</code>	$(\mathbb{S}, \gamma f[ip, \dots l_i \dots, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip + 5, \dots l_i \dots, \delta l_i])$
<code>push_local $n > 0, i$</code>	$(\mathbb{S}, \gamma g^{block}[-, \dots l_i \dots, -]^{d-n} \dots g^{block'}[ip, -, -]^{d-1} \dots g^{block''}[ip, -, \delta]^d) \Rightarrow$ $(\mathbb{S}, \gamma \dots g^{block''}[ip + 5, -, \delta l_i]^d)$
<code>push_literal i</code>	$(\mathbb{S}, \gamma f[ip, -, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip + 3, -, \delta f_{literal_i}])$
<code>push_global i</code>	$(\mathbb{S}, \gamma f[ip, -, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip + 3, -, \delta \mathbb{S}[f_{literal_i}]])$
<code>push_array n</code>	$(\mathbb{S}, \gamma f[ip, -, \delta a_1..a_n]) \Rightarrow (\mathbb{S}, \gamma f[ip + 3, -, \delta A])$ where $A = \text{Array}(a_1..a_n)$
<code>store_field i</code>	$(\mathbb{S}, \gamma f_s[ip, -, \delta \mathbf{v}]) \Rightarrow (\mathbb{S}[s_{field_i} = \mathbf{v}], \gamma f_s[ip + 3, -, \delta \mathbf{v}])$
<code>store_local n, i</code>	$(\mathbb{S}, \gamma f[ip, \dots l_i \dots, \delta \mathbf{v}]) \Rightarrow (\mathbb{S}, \gamma f[ip + 5, \dots l_{i-1} \mathbf{v} l_{i+1} \dots, \delta \mathbf{v}])$
<code>pop</code>	$(\mathbb{S}, \gamma f[ip, -, \delta \mathbf{v}]) \Rightarrow (\mathbb{S}, \gamma f[ip + 1, -, \delta])$
<code>send n, i</code>	$(\mathbb{S}, \gamma f[ip, -, \delta r_{p_1..p_n}]) \Rightarrow (\mathbb{S}, \gamma f[ip + 5, -, \delta] (r_{class} \bowtie f_{literal_i})_r[0, p_1..p_n, \epsilon])$
<code>send_super n, i</code>	$(\mathbb{S}, \gamma f[ip, -, \delta r_{p_1..p_n}]) \Rightarrow (\mathbb{S}, \gamma f[ip + 5, -, \delta] (r_{superclass} \bowtie f_{literal_i})_r[0, p_1..p_n, \epsilon])$
<code>block i</code>	$(\mathbb{S}, \gamma f[ip, -, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip + 3, -, \delta f_s^{block_i}])$
<code>block_return</code>	$(\mathbb{S}, \gamma f[ip, -, \delta] g^{block}[-, -, \delta' \mathbf{v}]) \Rightarrow (\mathbb{S}, \gamma f[ip, -, \delta \mathbf{v}])$
<i>(method local)</i> <code>return</code>	$(\mathbb{S}, \gamma f[ip, -, \delta] g[-, -, \delta' \mathbf{v}]) \Rightarrow (\mathbb{S}, \gamma f[ip, -, \delta \mathbf{v}])$
<i>(method nonlocal)</i> <code>return</code>	$(\mathbb{S}, \gamma f[ip, -, \delta] g_s[-, -, -] \dots h[-, -, -] g_s^{block}[-, -, \delta' \mathbf{v}]) \Rightarrow (\mathbb{S}, \gamma f[ip, -, \delta \mathbf{v}])$
<code>dbg i, loc</code>	$(\mathbb{S}, \gamma f[ip, -, -]) \Rightarrow (\mathbb{S}[file=f_{literal_i}, line=loc[31:8], col=loc[7:0]], \gamma f[ip + 7, -, -])$ Set VM current filename to $f_{literal_i}$ and split loc into char position (indexed from 0) from lower 8 bits and line number from the upper 24 bits.

Figure 2: Smalltalk VM State Transition Rules

Bytecode Instruction	Description
<code>nil</code>	Push <code>nil</code> onto the operand stack of the <i>current block or method context</i> .
<code>self</code>	Push <code>self</code> , the current method's receiver object, onto the operand stack
<code>true</code>	Push <code>true</code> onto the operand stack
<code>false</code>	Push <code>false</code> onto the operand stack
<code>push_char c</code>	Push character <i>c</i> onto the operand stack
<code>push_int i</code>	Push integer <i>i</i> onto the operand stack
<code>push_float f</code>	Push floating-point <i>f</i>
<code>push_field i</code>	Push field at index <i>i</i> of <code>self</code>
<code>push_local 0, i</code>	Push the local variable or method argument at index <i>i</i> of the current context onto the operand stack of the current context
<code>push_local n > 0, i</code>	Push the local variable or method argument at index <i>i</i> of the context <i>n</i> callers above (stack growing downwards) onto the operand stack of the current context
<code>push_literal i</code>	Push string literal at literal index <i>i</i> onto the stack
<code>push_global i</code>	Look up string at literal index <i>i</i> in the global scope and push that object onto the operand stack of the current context. This is used to look up class definition objects primarily as in <code>Array new</code> .
<code>push_array n</code>	Pop <i>n</i> items from the operand stack, create an array of them, push the array back to the operand stack
<code>store_field i</code>	Store the top of the operand stack into field <i>i</i> of <code>self</code> ; does <i>not</i> pop the operand stack
<code>store_local n, i</code>	Store the top of the operand stack into local <i>i</i> of the current context; does <i>not</i> pop the operand stack
<code>pop</code>	Pop the top of the operand stack off
<code>send n, i</code>	Send message with <i>n</i> arguments to method identified by string literal <i>i</i> (in current context). The receiver of the message <code>send</code> is pushed on the stack before the arguments so this instruction pops <i>n</i> + 1 items from the operand stack. The method is looked up in the class of the receiver object
<code>send_super n, i</code>	Same as <code>send</code> except that the method is looked up in the superclass of <code>self</code> 's class. The receiver is always <code>self</code>
<code>block i</code>	Push the block descriptor identified by index <i>i</i> onto the operand stack of the current context
<code>block_return</code>	Pop the return value on the top of the operand stack of the current context and push it on the operand stack of the caller's context. Pop the context stack.
<i>(method local)</i> <code>return</code>	Pop the top of the operand stack of the current context and push it on the operand stack of the caller's context. Pop the context stack.
<i>(method nonlocal)</i> <code>return</code>	Pop the top of the operand stack of the current context, unwind the call stack until the *enclosing method* of the current block, pop to the caller, push the return value onto the operand stack
<code>dbg i, loc</code>	Set VM current filename to $f_{literal_i}$ and split <i>loc</i> into char position (indexed from 0) from lower 8 bits and line number from the upper 24 bits. $line=loc[31:8], col=loc[7:0]$

Figure 3: Smalltalk VM Bytecode

Smalltalk	Context stack at ↩
<pre>"Test returnFromNestedCallViaBlock" class Test [f [self g: [^99↩]] g: blk [self h: blk] h: blk [blk value]] Test new f</pre>	<p>Before return:</p> $main[-, ,] \underbrace{f[-, ,] \ g[-, f^{block_0},] \ h[-, f^{block_0},] \ f^{block_0}[-, , 99]}_{\text{enclosing context } \Delta=1}$ <p>After return:</p> $main[-, , 99]$ <p>Notes: Despite eval in g, $[^99]$ unrolls stack to $main$, the caller of f.</p>
Smalltalk	Context stack at ↩
<pre>"Test testSendBlockBackToSameMethod- AndSetLocal" class T [f: blk pass: p [x p=1 ifTrue: [self g: [x:=5↩]] ifFalse: [blk value]. ^x] g: blk [self f: blk pass: 2]] T new f: nil pass: 1</pre>	<p>At store_local $\Delta=2, i=2$:</p> <div style="display: flex; align-items: center;"> <div style="margin-right: 20px;"> <p>1st call</p> <p>2nd call</p> <p>ctx →</p> </div> <div style="border: 1px solid black; padding: 5px; text-align: center;"> $main[-, ,]$ $f:pass: [-, nil\ 1\ 5,]$ $f:pass:^{block_0} [-, ,]$ $g[-, f:pass:^{block_1},]$ $f:pass: [-, f:pass:^{block_1}\ 2\ nil,]$ $f:pass:^{block_2} [-, ,]$ $f:pass:^{block_1} [-, , 5]$ Stack </div> <div style="margin-left: 20px;"> <p>←</p><p>←</p><p>←</p> </div> </div> <p>Notes: The enclosing block of $[x:=5]$ (called $f:pass:^{block_1}$) is $[self\ g: [x:=5]]$ (called $f:pass:^{block_0}$). The enclosing block of $f:pass:^{block_0}$ is the first call, not the second call, to $f:pass:.$</p>

Smalltalk fragment	Visitor method result	Side-effects
ϵ	ϵ (object Code.None)	
class T : S []	ϵ	
main	main	
	self	
	return	
f <primitive:#primitive-name>	ϵ	
f []	ϵ	f _{code} = self
		return
f [body]	ϵ	f _{code} = body
		pop
		self
		return
operator [body]	ϵ	operator _{code} = body
		pop
		self
		return
a:x b:y c:z [body]	ϵ	a:b:c:code = body
		pop
		self
		return
$\underbrace{[args locals]}_{f^{block_i}}$	block i	f _{block_i} = nil
$\underbrace{[body]}_{f^{block_i}}$	block i	block_return
		f _{block_i} = body
		block_return
expr ₁ .expr ₂ expr _n	expr ₁	
	pop	
	expr ₂	
	pop	
	...	
	expr _n	

Figure 4: Smalltalk Class/Method/Block Compilation Rules

Smalltalk fragment	Visitor method result	Side-effects
class T [x ₀ x ₁ ..x _n]...f [... x _i :=expr	expr	
a:x ₀ b:x ₁ [x ₂ ..x _n]... x _i :=expr	store_field i expr	
f [x ₀ ..x _n]... x _i :=expr	store_local 0, i expr	
f [... [x ₀ ..x _n]... x _i :=expr	store_local 0, i expr	
f:x [... [... x _i :=expr	store_local 0, i store_local Δ, 0	
$\Delta = \#scopes$ f [... [x] ... [... x _i :=expr	store_local Δ, 0 expr	
Δ class T [x ₀ x ₁ ..x _n]...f [... x _i	push_field i	
a:x ₀ b:x ₁ [x ₂ ..x _n]... x _i	push_local 0, i	
f:x [... [... x	push_local Δ, 0	
$\Delta = \#scopes$ f [... [x] ... [... x	push_local Δ, 0	
Δ		
99	push_int 99	
\$a	push_char ASCII('a')	
1.2	push_float asIntBits(1.2)	
'a string'	push_literal i	$f_{literal_i}^{block_j} = \text{"a string"}$
nil	nil	
self	self	
true	true	
false	false	
{ expr ₁ .expr ₂ expr _n }	expr ₁ expr ₂ ... expr _n push_array n	

Figure 5: Smalltalk Expression Compilation Rules

Smalltalk fragment	Visitor results	Side-effects
(unary msg) f [... <i>expr</i> <i>w</i>	<i>expr</i> send 0, <i>i</i>	$\mathbf{f}_{literal_i}^{block_j} = "w"$
(binary msg) f [... <i>expr</i> ₁ <i>op</i> <i>expr</i> ₂	<i>expr</i> ₁ <i>expr</i> ₂ send 1, <i>i</i>	$\mathbf{f}_{literal_i}^{block_j} = "op"$
f [... <i>expr</i> <i>w</i> ₁ : <i>e</i> ₁ <i>w</i> ₂ : <i>e</i> ₂ ... <i>w</i> _{<i>n</i>} : <i>e</i> _{<i>n</i>}	<i>expr</i> <i>e</i> ₁ <i>e</i> ₂ ... <i>e</i> _{<i>n</i>} send <i>n</i> , <i>i</i>	$\mathbf{f}_{literal_i}^{block_j} = "w_1:w_2:\dots w_n:"$
f [... super <i>w</i>	self send_super 0, <i>i</i>	$\mathbf{f}_{literal_i}^{block_j} = "w"$
f [... super <i>w</i> ₁ : <i>e</i> ₁ <i>w</i> ₂ : <i>e</i> ₂ ... <i>w</i> _{<i>n</i>} : <i>e</i> _{<i>n</i>}	self <i>e</i> ₁ <i>e</i> ₂ ... <i>e</i> _{<i>n</i>} send_super <i>n</i> , <i>i</i>	$\mathbf{f}_{literal_i}^{block_j} = "w_1:w_2:\dots w_n:"$
^ <i>expr</i>	<i>expr</i> return	

Figure 6: Smalltalk Message Expression Compilation Rules