CS652 Smalltalk VM Operational Semantics

Terence Parr

April 12, 2017

Smalltalk	Context stack at ↩

`"Test testEvalReturnBlock"`
`class T [`
`  f [|x| x := 1.^[x := 5] ]`
`]`
`t := T new.`
`t f value` ↩
	Start `send 0,'value'`:
	$$main[\_, nil, \_]\ f\text{-}block0[\_, ,]$$
	Notes: no f on stack during eval of f-$block0$ but enclosing scope of f-$block0$ still points at f's `BlockContext`.

Smalltalk	Context stack at ↩

`"Test testRemoteMethodCanSetMyLocal"`
`class T [`
`  f [|x| self g:[x := 5` ↩`]]`
`  g:  blk [ blk value]`
`]`
`T new f`
	`@store_local` Δ=1,i=0:
	$$main[\_, ,]\ \underbrace{f[\_, nil,]\ g[\_, f^{block_0},]\ f}_{\overleftarrow{\text{enclosing context}}\ \Delta=1}\ ^{block_0}[\_, , \mathbf{5}]$$
	After `block_return`:
	$$main[\_, ,]\ f[\_, \mathbf{5},]\ g[\_, f^{block_0}, \mathbf{5}]$$

Smalltalk	Context stack at ↩

`"Test testRemoteReturn"`
`class T [`
`  f [ self g:[^99] ]`
`  g:blk [ blk value` ↩`]`
`]`
`|t|`
`t := T new.`
`t f`
	Start `send 0,'value'`:
	$$main[\_, t,]\ f[\_, ,]\ g[\_, f^{block_0},]\ f^{block_0}[\_, ,]$$
	After `return` in `[^99]` block:
	$$main[\_, t, 99]$$
	Notes: Despite eval in g, `[^99]` unrolls stack to $main$, the caller of f.

$T \bowtie x$	Resolve x in scope T
$o \in \texttt{X}$	o is instance of X
$\mathbf{v} \in \texttt{STObject}$	a single object
$\boldsymbol{l}_i \in \texttt{STObject}$	the i^{th}argument or local variable object
$o_{class} \in \texttt{STMetaClassObject}$	Metaclass (type) of object o
$o_{class_{class}} = o_{class}$	A metaclass object is its own type
$o_{superclass} \in \texttt{STMetaClassObject}$	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
$f_s^{block_i} \in \texttt{BlockDescriptor}$	The i^{th}block of method f associated with instance self=s
$f_s^{block_i}[\_, \_, \_] \in \texttt{BlockContext}$	The i^{th}block of method f invoked with self=s
$f_s^{block_i}[\_, \_, \_]^d \in \texttt{BlockContext}$	The i^{th}block of method f invoked with self=s and having depth d counting from zero at the method block; e.g., `f [|x| [|y|]]` has a method block at depth 0 with `x` and a nested block at depth 1 with `y`
$\gamma \in \texttt{MethodContext}^*$	Stack of method invocations growing to the right
$\delta \in \texttt{STObject}^*$	Operand stack of objects growing to the right
$\mathbb{S}$	The state of the VM system dictionary
$(\mathbb{S}, \gamma)$	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); $f \in$ `MethodContext`; l_i is local variable or argument, indexed from 0 and arguments first; δ is the operand stack; *f can also represent a nested code block not just a method*
$f[ip, l_0, ..l_{n-1}, \delta]$	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**
initial state	$state_0 = (\mathbb{S}[\texttt{Transcript}], \texttt{main}_m[0, \epsilon, \epsilon])$ for $m \in \texttt{MainClass}$; program terminates if $\exists\ state_0 \Rightarrow^* (\mathbb{S}', \epsilon)$
`nil`	$(\mathbb{S}, \gamma f[ip, \_, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip+1, \_, \delta\, \texttt{nil}])$
`self`	$(\mathbb{S}, \gamma f_s[ip, \_, \delta]) \Rightarrow (\mathbb{S}, \gamma f_s[ip+1, \_, \delta\, s])$
`true`	$(\mathbb{S}, \gamma f[ip, \_, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip+1, \_, \delta\, \texttt{true}])$
`false`	$(\mathbb{S}, \gamma f[ip, \_, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip+1, \_, \delta\, \texttt{false}])$
`push_char` c	$(\mathbb{S}, \gamma f[ip, \_, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip+3, \_, \delta\, c])]$
`push_int` i	$(\mathbb{S}, \gamma f[ip, \_, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip+5, \_, \delta\, i])$
`push_float` i	$(\mathbb{S}, \gamma f[ip, \_, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip+5, \_, \delta\, intBitsToFloat(i)])$
`push_field` i	$(\mathbb{S}, \gamma f_s[ip, \_, \delta]) \Rightarrow (\mathbb{S}, \gamma f_s[ip+3, \_, \delta\, s_{field_i}])$
`push_local` $0, i$	$(\mathbb{S}, \gamma f[ip, \cdots l_i \cdots, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip+5, \cdots l_i \cdots, \delta\, l_i])$
`push_local` $n > 0, i$	$(\mathbb{S}, \gamma g^{block}[\_, \cdots \boldsymbol{l}_i \cdots, \_]^{d-n} \cdots g^{block'}[ip, \_, \_]^{d-1} \cdots g^{block''}[ip, \_, \delta]^d) \Rightarrow$ $(\mathbb{S}, \gamma \cdots g^{block''}[ip+5, \_, \delta \boldsymbol{l}_i]^d)$
`push_literal` i	$(\mathbb{S}, \gamma f[ip, \_, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip+3, \_, \delta\, f_{literal_i}])$
`push_global` i	$(\mathbb{S}, \gamma f[ip, \_, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip+3, \_, \delta\, \mathbb{S}[f_{literal_i}]])$
`push_array` n	$(\mathbb{S}, \gamma f[ip, \_, \delta\, a_1..a_n]) \Rightarrow (\mathbb{S}, \gamma f[ip+3, \_, \delta A])$ where $A = Array(a_1..a_n)$
`store_field` i	$(\mathbb{S}, \gamma f_s[ip, \_, \delta\, \mathbf{v}]) \Rightarrow (\mathbb{S}[s_{field_i} = \mathbf{v}], \gamma f_s[ip+3, \_, \delta\, \mathbf{v}])$
`store_local` n, i	$(\mathbb{S}, \gamma f[ip, \cdots l_i \cdots, \delta\, \mathbf{v}]) \Rightarrow (\mathbb{S}, \gamma f[ip+5, \cdots l_{i-1} \mathbf{v}\, l_{i+1} \cdots, \delta\, \mathbf{v}])$
`pop`	$(\mathbb{S}, \gamma f[ip, \_, \delta\, \mathbf{v}]) \Rightarrow (\mathbb{S}, \gamma f[ip+1, \_, \delta])$
`send` n, i	$(\mathbb{S}, \gamma f[ip, \_, \delta\, r\, p_1..p_n]) \Rightarrow (\mathbb{S}, \gamma f[ip+5, \_, \delta]\ (r_{class} \bowtie f_{literal_i})_r[0, p_1..p_n, \epsilon])$
`send_super` n, i	$(\mathbb{S}, \gamma f[ip, \_, \delta\, r\, p_1..p_n]) \Rightarrow (\mathbb{S}, \gamma f[ip+5, \_, \delta]\ (r_{superclass} \bowtie f_{literal_i})_r[0, p_1..p_n, \epsilon])$
`block` i	$(\mathbb{S}, \gamma f[ip, \_, \delta]) \Rightarrow (\mathbb{S}, \gamma f[ip+3, \_, \delta\, f_s^{block_i}])$
`block_return`	$(\mathbb{S}, \gamma f[ip, \_, \delta]\ g^{block}[\_, \_, \delta'\, \mathbf{v}]) \Rightarrow (\mathbb{S}, \gamma f[ip, \_, \delta\, \mathbf{v}])$
(*method local*) `return`	$(\mathbb{S}, \gamma f[ip, \_, \delta]\ g[\_, \_, \delta'\, \mathbf{v}]) \Rightarrow (\mathbb{S}, \gamma f[ip, \_, \delta\, \mathbf{v}])$
(*method nonlocal*) `return`	$(\mathbb{S}, \gamma f[ip, \_, \delta]\ g_s[\_, \_, \_] \cdots h[\_, \_, \_]\ g_s^{block}[\_, \_, \delta'\, \mathbf{v}]) \Rightarrow (\mathbb{S}, \gamma f[ip, \_, \delta\, \mathbf{v}])$
`dbg` i, loc	$(\mathbb{S}, \gamma f[ip, \_, \_]) \Rightarrow (\mathbb{S}[file{=}f_{literal_i}, line{=}loc[31{:}8], col{=}loc[7{:}0]], \gamma f[ip+7, \_, \_])$ 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

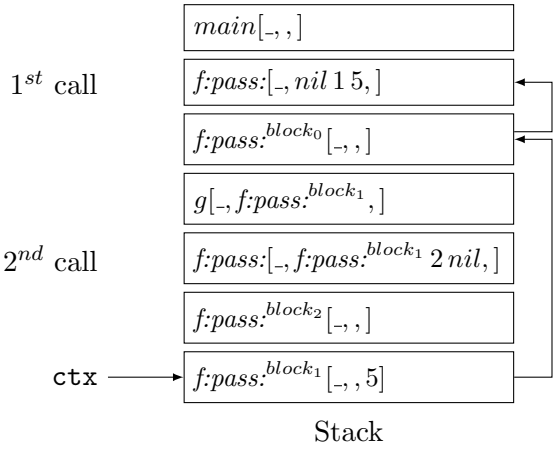
Bytecode Instruction	Description
`nil`	Push `nil` onto the operand stack of the *current block or method context.*
`self`	Push `self`, the current method's receiver object, onto the operand stack
`true`	Push `true` onto the operand stack
`false`	Push `false` onto the operand stack
`push_char` c	Push character c onto the operand stack
`push_int` i	Push integer i onto the operand stack
`push_float` f	Push floating-point f
`push_field` i	Push field at index i of `self`
`push_local` $0, i$	Push the local variable or method argument at index i of the current context onto the operand stack of the current context
`push_local` $n > 0, i$	Push the local variable or method argument at index i of the context n callers above (stack growing downwards) onto the operand stack of the current context
`push_literal` i	Push string literal at literal index i onto the stack
`push_global` i	Look up string at literal index 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 `Array new`.
`push_array` n	Pop n items from the operand stack, create an array of them, push the array back to the operand stack
`store_field` i	Store the top of the operand stack into field i of `self`; does *not* pop the operand stack
`store_local` n, i	Store the top of the operand stack into local i of the current context; does *not* pop the operand stack
`pop`	Pop the top of the operand stack off
`send` n, i	Send message with n arguments to method identified by string literal i (in current context). The receiver of the message send is pushed on the stack before the arguments so this instruction pops $n+1$ items from the operand stack. The method is looked up in the class of the receiver object
`send_super` n, i	Same as `send` except that the method is looked up in the superclass of `self`'s class. The receiver is always `self`
`block` i	Push the block descriptor identified by index i onto the operand stack of the current context
`block_return`	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.
(*method local*) `return`	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.
(*method nonlocal*) `return`	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
`dbg` i, loc	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. *line*=*loc*[31:8], *col*=*loc*[7:0]]

Figure 3: Smalltalk VM Bytecode

Smalltalk	Context stack at ↩

`"Test returnFromNestedCallViaBlock"`
`class Test [`
`  f [ self g: [^99↩] ]`
`  g: blk [ self h: blk ]`
`  h: blk [ blk value ]`
`]`
`Test new f`
	Before `return`:
	$main[_, ,]\ \underbrace{f[_, ,]\ g[_, f^{block_0},]\ h[_, f^{block_0},]\ f}_{\text{enclosing context } \Delta=1}\ ^{block_0}[_, , 99]$
	After `return`:
	$main[_, , 99]$
	Notes: Despite eval in g, `[^99]` unrolls stack to $main$, the caller of f.

Smalltalk	Context stack at ↩

`"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`
	At `store_local` Δ=2,i=2:
	 
	
	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:*.

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	$\texttt{f}_{code}$ =
		`self`
		`return`

`f [` *body* `]`	ϵ	$\texttt{f}_{code}$ =
		body
		`pop`
		`self`
		`return`

operator `[` *body* `]`	ϵ	$operator_{code}$ =
		body
		`pop`
		`self`
		`return`

`a: x b: y c: z [` *body* `]`	ϵ	$\texttt{a:b:c:}_{code}$ =
		body
		`pop`
		`self`
		`return`

$\underbrace{\texttt{[}args\texttt{| |}locals\texttt{|]}}_{\texttt{f}^{block_i}}$	`block` i	$\texttt{f}_{block_i}$ =
		`nil`
		`block_return`

$\underbrace{\texttt{[}body\texttt{]}}_{\texttt{f}^{block_i}}$	`block` i	$\texttt{f}_{block_i}$ = *body* `block_return`
$expr_1 . expr_2 . \cdots expr_n$	$expr_1$ `pop` $expr_2$ `pop` $\cdots$ $expr_n$	

Figure 4: Smalltalk Class/Method/Block Compilation Rules

Smalltalk fragment	Visitor method result	Side-effects
`class T` $[\|x_0 x_1 .. x_n\| \cdots$ `f` $[\cdots\ x_i := expr$	$expr$	
	`store_field` i	
`a:` x_0 `b:` x_1 $[\|x_2 .. x_n\| \cdots\ x_i := expr$	$expr$	
	`store_local` $0, i$	
`f` $[\|x_0 .. x_n\| \cdots\ x_i := expr$	$expr$	
	`store_local` $0, i$	
`f` $[\cdots\ [\|x_0 .. x_n\| \cdots\ x_i := expr$	$expr$	
	`store_local` $0, i$	
$\underbrace{\texttt{f:x}\ [\cdots[}_{\Delta = \#scopes}\ \cdots\ x_i := expr$	`store_local` $\Delta, 0$	
`f [`$\cdots \underbrace{[\|\texttt{x}\|\cdots[}_{\Delta}\ \cdots\ x_i := expr$	$expr$	
	`store_local` $\Delta, 0$	
`class T` $[\|x_0 x_1 .. x_n\| \cdots$ `f` $[\cdots\ x_i$	`push_field` i	
`a:` x_0 `b:` x_1 $[\|x_2 .. x_n\| \cdots\ x_i$	`push_local` $0, i$	
$\underbrace{\texttt{f:x}\ [\cdots[}_{\Delta = \#scopes}\ \cdots$ `x`	`push_local` $\Delta, 0$	
`f [`$\cdots \underbrace{[\|\texttt{x}\|\cdots[}_{\Delta}\ \cdots$ `x`	`push_local` $\Delta, 0$	
`99`	`push_int 99`	
`$a`	`push_char` *ASCII('a')*	
`1.2`	`push_float` *asIntBits(1.2)*	
`class T` $[\cdots$ `'a string'`	`push_literal` i	$\texttt{T}_{literal_i}$ = "`a string`"
`nil`	`nil`	
`self`	`self`	
`true`	`true`	
`false`	`false`	
$\{\ expr_1 . expr_2 . \cdots expr_n\ \}$	$expr_1$	
	$expr_2$	
	$\cdots$	
	$expr_n$	
	`push_array` n	

Figure 5: Smalltalk Expression Compilation Rules

	Smalltalk fragment	**Visitor results**	**Side-effects**
(unary msg)	`class T [`$\cdots$ `f [`$\cdots$ *expr w*	*expr* `send` $0, i$	$\texttt{T}_{literal_i}$ = "*w*"
(binary msg)	`class T [`$\cdots$ `f [`$\cdots$ *expr*$_1$ *op expr*$_2$	*expr*$_1$ *expr*$_2$ `send` $1, i$	$\texttt{T}_{literal_i}$ = "*op*"
	`class T [`$\cdots$ `f [`$\cdots$ *expr* w_1:e_1 w_2:e_2 $\cdots$ w_n:e_n	*expr* e_1 e_2 $\cdots$ e_n `send` n, i	$\texttt{T}_{literal_i}$ = "w_1:w_2:$\cdots$ w_n:"
	`class T [`$\cdots$ `f [`$\cdots$ `super` *w*	`self` `send_super` $0, i$	$\texttt{T}_{literal_i}$ = "*w*"
	`class T [`$\cdots$ `f [`$\cdots$ `super` w_1:e_1 w_2:e_2 $\cdots$ w_n:e_n	`self` e_1 e_2 $\cdots$ e_n `send_super` n, i	$\texttt{T}_{literal_i}$ = "w_1:w_2:$\cdots$ w_n:"
	^*expr*	*expr* `return`	

Figure 6: Smalltalk Message Expression Compilation Rules