HOME

PAGES

About Cog
About this blog
Building a Cog
Development Image
Cog Projects
Collaborators
Compiling the VM
Downloads
Eliot Miranda
On-line Papers and Presentations

CATEGORIES

Cog
Spur

SEARCH

Find

{ 2008 07 22 }

Closures Part II – the Bytecodes

Its been a while since the last post. I've been working on the stack VM and the closure bootstrap. But its high time I posted more of the closure implementation. Apologies for the delay.

In the post before last we saw the closure design in Cog and its prototype implementation above an unmodified VM. This post looks at the opcodes used to implement Closures in the image. The image level implementation is in InsructionStream, ContextPart and MethodContext is used by the debugger and some of the image-level browsing facilities. The next post will probably present the VM implementation.

Before we dive in, let's recap. The closure implementation has one key aim for performance. The activation of a closure should be independent of its enclosing activation. This simplifies stack management meaning that while the closure is running it doesn't need to access its lexically enclosing activation to run. This is achieved by a simple transformation in two parts. Any closed-over variable that does not change after being closed-over is copied into the closure which accesses the copy. Any closed-over variable which does change after being closed over is put in a heap-allocated "indirection vector" (a simple array, one element per closed-over variable) and the indirection vector is copied into the closure. All accesses to the variable are made through the indirection vector. Here's the transformation applied at the source level to inject:into:

Collection methods for enumerating

```
inject: thisValue into: binaryBlock
	| nextValue |
	nextValue := thisValue.
	self do: [:each | nextValue := binaryBlock value: nextValue value: each].
	^nextValue
```

transforms to

```
inject: thisValue into: binaryBlock
	| indirectTemps |
	indirectTemps := Array new: 1.
	indirectTemps at: 1 put: thisValue.
	self do: [:each | indirectTemps at: 1 put: (binaryBlock value: (indirectTemps at: 1) value: each)].
	^(indirectTemps at: 1)
```

So let's implement this with bytecodes, so that we can bootstrap closures in something close to the current VM, something that will run both the existing system and a system recompiled to use closures. Before we do so we need to digress a bit on bytecode set design.

Eventually we'll design a new bytecode set which will be chosen to encode the opcodes we find in the current system as efficiently as we can. That means

– making sure that opcodes with high dynamic frequency have short bytecodes that are quick to interpret
– that opcodes with high static frequency have short bytecodes that represent them compactly
– and that bytecodes exist to ensure all opcodes have large enough ranges to implement the system without restriction.

What do I mean? Well, look at the Push Receiver Variable opcode which has several bytecodes in the current Squeak VM:

0-15 0000iiii Push Receiver Variable #iiii
128 10000000 jjkkkkkk Push (Receiver Variable, Temporary Location, Literal Constant, Literal Variable) [jj] #kkkkkk
132 10000100 iiijjjjj kkkkkkkk (Send, Send Super, Push Receiver Variable,
Push Literal Constant,
Push Literal Variable, Store Receiver Variable,
Store-Pop Receiver Variable,
Store Literal Variable)[iii] #kkkkkkkk jjjjj

Push Receiver Variable has both high dynamic frequency (how often an opcode is evaluated) and static frequency (how often an operation occurs in methods). Most objects have few instance variables. In my Croquet image only 20% of classes (ignoring metaclasses) have more than 15 instance variables. So there is both a high static and high dynamic frequency of pushing instance variables with a small offset. Hence the design above where there are 16 one-byte bytecodes to access the instance variables with offsets 0 through 15, a two byte bytecode that can access instance variables with offsets 0 through 63 and a three byte bytecode that can access instance variables with offsets 0 through 255, the maximum imposed by a class's instance format word (the largest class in my image has 139 instance variables).

Interpreting the short forms 0-15 0000iiii Push Receiver Variable #iiii is a lot quicker than interpreting the longer forms. One piece of code I ran to measure the sizes of bytecoded methods in two images, one which used only the long-forms and one which used the shortest forms available, ran 40% slower in the image which used only the long forms. So the game in designing a bytecode set is to divide up the available bytecode space amongst the opcodes to achieve a compromise between dynamic and static frequency. In practice optimizing on the basis of static frequency is a good heuristic since static frequency is a fair predictor of dynamic frequency.

But the design of a bytecode set for the closure system is something that can wait until we've bootstraped the closure system and can start to perform some measurements. The first order of business is to bootstrap closures in the existing Squeak VM. To do that we need to find room. Only if we can't find room do we need to make room. Luckily, the existing Squeak VM has 8 unused bytecodes, and that's more than enough.

Here are the unused entries in the method that defines the VM's bytecodes, Interpreter class>>initializeBytecodeTable:

```
…
(126 127 unknownBytecode)
…
(138 143 experimentalBytecode)
…
```

We'll use 5 of the 6 experimental bytecodes. These aren't generated by the compiler (and implement an optimization we hope to outdo). We don't have many to play with so we can't do much space optimization. Because we only have 6 and each opcode needs at least one index we'll have to use multi-byte encodings and include the indices in trailing bytes.

New Closure Bytecodes

Here are all 5 bytecodes. We'll go through each separately three times. First we'll define the bytecodes. Then we'll implement them in the image's execution simulation machinery, which we can use to test the bytecode before we generate a VM, and which is used by the debugger and also by the printing machinery that disassembles methods. lastly we'll go through the VM code.

```
	138	10001010 jkkkkkkk	Push (Array new: kkkkkkk)/Pop kkkkkkk into: (Array new: kkkkkkk)
	140	10001100 kkkkkkkk jjjjjjjj	Push Temp At kkkkkkkk In Temp Vector At: jjjjjjjj
	141	10001101 kkkkkkkk jjjjjjjj	Store Temp At kkkkkkkk In Temp Vector At: jjjjjjjj
	142	10001110 kkkkkkkk jjjjjjjj	Pop and Store Temp At kkkkkkkk In Temp Vector At: jjjjjjjj
	143	10001111 llllkkkk jjjjjjjj iiiiiiii	Push Closure Num Copied llll Num Args kkkk BlockSize jjjjjjjjiiiiiiii
```

The first opcode can be used to create indirect temp vectors. Because Squeak stack frames are limited in size (50-ish slots) we don't need to be able to create an Array with 265 elements. We can make our bytecode more useful if it can either create a new Array initialized with nils or initialize a new Array with elements popped off the stack.

```
	138	10001010 jkkkkkkk Push (Array new: kkkkkkk)/Pop kkkkkkk into: (Array new: kkkkkkk)
```

So we can use it to compile brace expressions. In Squeak

```
	{ expr1. expr2. expr3. }
```

is convenient short-hand for

```
	Array with: expr1 with: expr2 with: expr3
```

and more flexible because brace expressions can have any number of elements. The existing Squeak compiler compiles brace expressions with up to 4 elements as sends of braceWith:[with:[with:[with:]]]. e.g.

Bezier3Segment methods for vector functions
controlPoints

```
	^{start. via1. via2. end}
```

is compiled to

```
	21 <41> pushLit: Array
	22 <00> pushRcvr: 0
	23 <02> pushRcvr: 2
	24 <03> pushRcvr: 3
	25 <01> pushRcvr: 1
	26 <83 80> send: braceWith:with:with:with:
	28 <7C> returnTop
```

Question, why does the compiler limit itself to only 4 elements max?

With the new bytecode this compiles to

```
	13 <00> pushRcvr: 0
	14 <02> pushRcvr: 2
	15 <03> pushRcvr: 3
```

16 <01> pushRcvr: 1
17 <8A 84> pop 4 into (Array new: 4)
19 <7C> returnTop

The top bit of the following byte (bit j in the specification) is set and indicates popping 4 elements off the stack into the new array. The VM already knows the class Array because it is used e.g. in doesNotUnderstand: processing. We save 8 bytes by not having to have the literals for the class Array and the message selector #braceWith:with:with:with: in the compiled method. We save a further byte avoiding the pushLit: instruction. And of course the bytecode is much quicker to interpret than the old code. We're saving time and space. Way to go.

So in compiling inject:into: the first two bytecodes look like

17 <8A 01> push: (Array new: 1)
19 <6A> popIntoTemp: 2

We could have designed the bytecode to include the offset of the local into which we want to store the new array since in our compiler Push (Array new: kkkkkkk) is always followed by a popIntoTemp: n. But since there can currently be up to 32 locals we'd either be limited in the size of the array or use a three byte bytecode. Neither of these is a win. A bytecode that does both Push (Array new: kkkkkkk) and Pop kkkkkkk into: (Array new: kkkkkkk) is more flexible and just as compact because we frequently get to use the short-form for the following popIntoTemp.

Answer, if the compiler used the same approach for very large brace expressions then for one thing we'd need a lot of braceWith:……with: methods, but more importantly we'd soon get stack overflow given the small finite stack size of Smalltalk's contexts. So large brace expressions are compiled to stream code that is equivalent to

(Array braceStream: N) nextPut: expr1; nextPut: expr2; nextPut: expr3; … nextPut: exprN
and the closure compiler limits itself to 8 elements before using the stream implementation.

Now the Push Temp At m In Temp Vector At: n opcode. In a moment of weakness I decided to go for a short-form here.

139 10001011 jjjjkkkk Push Temp At kkkk In Temp Vector At: jjjj
140 10001100 kkkkkkkk jjjjjjjj Push Temp At kkkkkkkk In Temp Vector At: jjjjjjjj

But then I recovered my sanity. There were only 2700 occurrences of the short form in my image, which is about a 0.01% saving. The unused bytecode is probably much more valuable. But this explains the mysterious gap in the new bytecodes at 139. So the opcode's bytecode is

140 10001100 kkkkkkkk jjjjjjjj Push Temp At kkkkkkkk In Temp Vector At: jjjjjjjj

This fetches the local at offset j..j on the stack and push the k..k'th element in it.

The next two store into the k..k'th element of the local at j..j, one version popping the result off the stack. This is a general convention in the Smalltalk-80 compiler. These are store and store-pop forms of almost every store opcode. The store form is used in the stores into vat and vax in things like

var := vat := vax := expr

whereas the pop form gets used in the store into var.

141 10001101 kkkkkkkk jjjjjjjj Store Temp At kkkkkkkk In Temp Vector At: jjjjjjjj
142 10001110 kkkkkkkk jjjjjjjj Pop and Store Temp At kkkkkkkk In Temp Vector At: jjjjjjjj

The final bytecode is more interesting.

143 10001111 llllkkkk jjjjjjjj iiiiiiii Push Closure Num Copied llll Num Args kkkk BlockSize jjjjjjjjiiiiiiii

This creates a closure, initializing it from the arguments and the current context. Because I'm doing The Simplest Thing That Could Possibly Work this is very closely modelled on the old BlockContext scheme. We don't use a separate method for the block's code. Instead we embed the code for a block within the code for its enclosing block or method. This has pros and cons. Its arguably more compact because blocks and methods get to share literals if they access the same ones, and because there is only one CompiledMethod object, but this is offset by the fact that we will need larger ranges and hence more long-form bytecodes to access the literals because the CompiledMethod's literal frame contains the literals for the method and its blocks.

In the old scheme a block is created by sending blockCopy: to the active context with the block's argument count as its argument and then jumping around the bytecodes for the block. The primitive for blockCopy: takes into account the size of the jump and the compiler always generates a 2 byte jump. Its a hack but it works:

20 <89> pushThisContext
21 <76> pushConstant: 1
22 <C8> send: blockCopy:
23 <A4 08> jumpTo: 33
… the bytecodes for the block …
33 … the continuation of the method …

The bytecode set has short forms for push -1, 0, 1 and 2 where you see <76> being used above but if a block has more than 2 arguments then we need 4 bytes for the literal as well as the push. But we'll say that creating a block costs 5 bytecodes. One problem is that the long jump only has a 10-bit range so we can't have blocks with more than 1023 bytes of bytecode in them. Collapsing these 4 separate bytecodes down into 1 multibyte code saves time and space, and easily allows a 16-bit jump size.

So now the same block creation code compiles to the following bytecode:
27 <8F 21 00 09> closureCopyNumCopied: 2 numArgs: 1 bytes 31 to 40
… the bytecodes for the block …
40 … the continuation of the method …

We've gained a byte. The above looks worse (its our old friend Collection>>inject:into: which uses those bulky indirect temp bytecodes). In the common case we save, but any savings are offset by the bytecodes to push copied temps. We'll gather full measurements in the next post.

Implementing the Closure Bytecodes at the Image Level

To implement a new bytecode one needs to alter a few parts of the system. One place is the VM. We'll do that at the end of this post. There are two main places in the image. One is the compiler, to generate the bytecode. I'm going to defer discussing the compiler changes for now because there are enough changes to justify a post on its own. Another is

in the execution simulation machinery which is used for several things
– implementing the debugger
– implementing the decompiler
– implementing method disassembling
– implementing browser support

This all revolves around InstructionStream which is the class that manages the interpretation of bytecoded methods at the image level, the image's dual of the VM itself.

```
Object subclass: #InstructionStream
    instanceVariableNames: 'sender pc'
    classVariableNames: 'SpecialConstants'
    poolDictionaries: ''
    category: 'Kernel-Methods'
```

InstructionStream interprets bytecodes in two methods

```
    interpretNextInstructionFor: client
        "Send to the argument, client, a message that specifies the type of the
        next instruction."
    interpretExtension: offset in: method for: client
```

the latter takes care of the decoding of bytecodes 128 through 143 which are complex enough to need their own method. The interpretation causes the sending of messages to the client, one per opcode. Here's the entire opcode set for Smalltalk-80. As you can see its a small set, and more or less each opcode corresponds to a single token in the source language. InstructionClient is an abstract superclass for classes that implement these opcodes:

```
InstructionStream methods for instruction decoding
blockReturnTop
    "Return Top Of Stack bytecode."

doDup
    "Duplicate Top Of Stack bytecode."

doPop
    "Remove Top Of Stack bytecode."

jump: offset
    "Unconditional Jump bytecode."

jump: offset
    "Unconditional Jump bytecode."

methodReturnConstant: value
    "Return Constant bytecode."

methodReturnReceiver
    "Return Self bytecode."

methodReturnTop
    "Return Top Of Stack bytecode."

popIntoLiteralVariable: anAssociation
    "Remove Top Of Stack And Store Into Literal Variable bytecode."

popIntoReceiverVariable: offset
    "Remove Top Of Stack And Store Into Instance Variable bytecode."

popIntoTemporaryVariable: offset
    "Remove Top Of Stack And Store Into Temporary Variable
```

```
bytecode."

pushActiveContext
	"Push Active Context On Top Of Its Own Stack bytecode."

pushConstant: value
	"Push Constant, value, on Top Of Stack bytecode."

pushLiteralVariable: anAssociation
	"Push Contents Of anAssociation On Top Of Stack bytecode."

pushReceiver
	"Push Active Context's Receiver on Top Of Stack bytecode."

pushReceiverVariable: offset
	"Push Contents Of the Receiver's Instance Variable Whose Index
	 is the argument, offset, On Top Of Stack bytecode."

pushTemporaryVariable: offset
	"Push Contents Of Temporary Variable Whose Index Is the
	 argument, offset, On Top Of Stack bytecode."

send: selector super: supered numArgs: numberArguments
	"Send Message With Selector, selector, bytecode. The argument,
	 supered, indicates whether the receiver of the message is specified
with
	 'super' in the source method. The arguments of the message are
found in
	 the top numArguments locations on the stack and the receiver just
	 below them."

storeIntoLiteralVariable: anAssociation
	"Store Top Of Stack Into Literal Variable Of Method bytecode."

storeIntoReceiverVariable: offset
	"Store Top Of Stack Into Instance Variable Of Method bytecode."

storeIntoTemporaryVariable: offset
	"Store Top Of Stack Into Temporary Variable Of Method bytecode."
```

I'm shoe-horning the closure bytecodes into the unused extensions, so I need to modify the **interpretExtension:in:for:** method, Because we now have a four byte bytecode for the Push Closure Num Args M BlockSize N opcode we need a new temp at the beginning of the method:

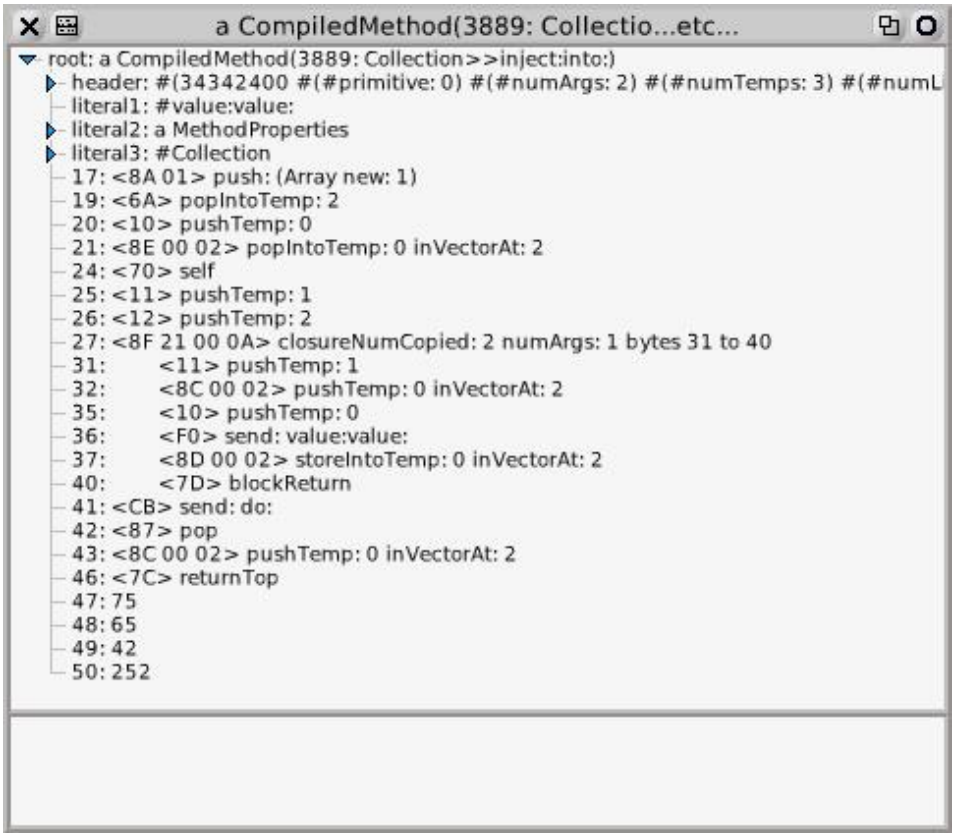
```
InstructionStream methods for private
interpretExtension: offset in: method for: client
	| type offset2 byte2 byte3 byte4 |
	… code for offset 0 through 6 omitted …
	offset = 7 ifTrue: [^client doPop].
	offset = 8 ifTrue: [^client doDup].
	offset = 9 ifTrue: [^client pushActiveContext].
	byte2 := method at: pc. pc := pc + 1.
	offset = 10 ifTrue:
		[^byte2 < 128
			ifTrue: [client pushNewArrayOfSize: byte2]
			ifFalse: [client pushConsArrayWithElements: byte2 –
128]].
	offset = 11 ifTrue: [^self error: 'unusedBytecode'].
	byte3 := method at: pc. pc := pc + 1.
	offset = 12 ifTrue: [^client pushRemoteTemp: byte2 inVectorAt:
byte3].
	offset = 13 ifTrue: [^client storeIntoRemoteTemp: byte2 inVectorAt:
```

```
byte3].
	offset = 14 ifTrue: [^client popIntoRemoteTemp: byte2 inVectorAt:
byte3].
	"offset = 15"
	byte4 := method at: pc. pc := pc + 1.
	^client
		pushClosureCopyNumCopiedValues: (byte2 bitShift: -4)
		numArgs: (byte2 bitAnd: 16rF)
		blockSize: (byte3 * 256) + byte4
```

The new opcodes need to be added to InstructionClient. The next place is to InstructionPrinter, which does disassembly of CompiledMethods:

```
popIntoRemoteTemp: remoteTempIndex inVectorAt: tempVectorIndex
	"Remove Top Of Stack And Store Into Offset of Temp Vector
bytecode."
	self print: 'popIntoTemp: ', remoteTempIndex printString, ' inVectorAt: ',
tempVectorIndex printString

pushClosureCopyNumCopiedValues: numCopied numArgs: numArgs
blockSize: blockSize
	"Push Closure bytecode."
	self print: 'closureNumCopied: ', numCopied printString
			, ' numArgs: ', numArgs printString
			, ' bytes ', scanner pc printString
			, ' to ', (scanner pc + blockSize – 1) printString.
	innerIndents
		atAll: (scanner pc to: scanner pc + blockSize – 1)
		put: (innerIndents at: scanner pc – 1) + 1

pushConsArrayWithElements: numElements
	"Push Cons Array of size numElements popping numElements items
from the stack into the array bytecode."
	self print: 'pop ', numElements printString, ' into (Array new: ',
numElements printString, ')'

pushNewArrayOfSize: numElements
	"Push New Array of size numElements bytecode."
	self print: 'push: (Array new: ', numElements printString, ')'

pushRemoteTemp: remoteTempIndex inVectorAt: tempVectorIndex
	"Push Contents at Offset in Temp Vector bytecode."
	self print: 'pushTemp: ', remoteTempIndex printString, ' inVectorAt: ',
tempVectorIndex printString

storeIntoRemoteTemp: remoteTempIndex inVectorAt: tempVectorIndex
	self print: 'storeIntoTemp: ', remoteTempIndex printString, ' inVectorAt: ',
tempVectorIndex printString
```

and now I can see what Collection>>inject:into: looks like:



Then we have to implement them in ContextPart, MethodContext and BlockContext, the three classes that define the image-level interpreter. The debugger uses these definitions to simulate execution.

ContextPart methods for instruction decoding

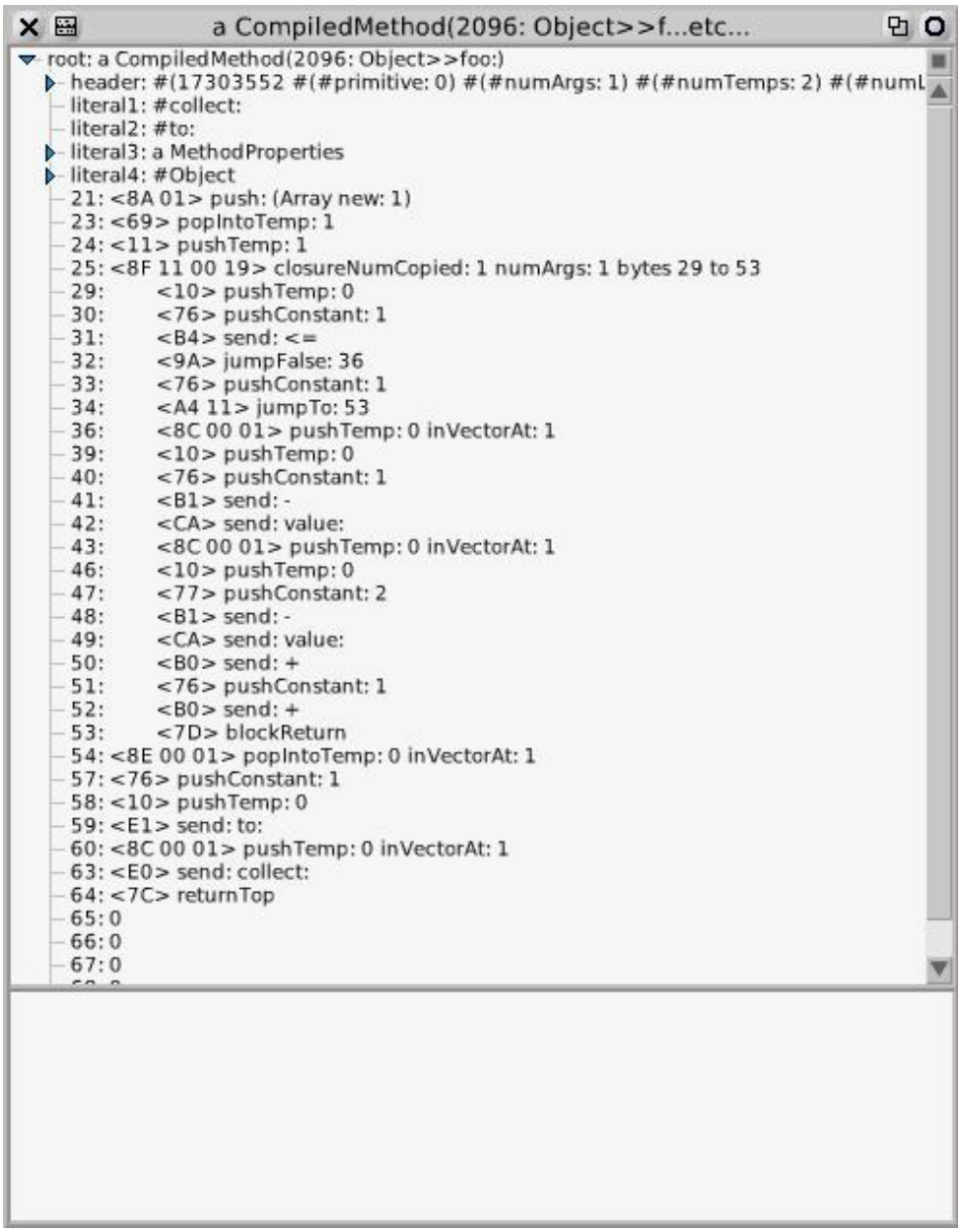
```
popIntoRemoteTemp: remoteTempIndex inVectorAt: tempVectorIndex
	"Simulate the action of bytecode that removes the top of the stack and stores
	 it into an offset in one of my local variables being used as a remote temp vector."

	(self at: tempVectorIndex + 1) at: remoteTempIndex + 1 put: self pop

pushClosureCopyNumCopiedValues: numCopied numArgs: numArgs blockSize: blockSize
	"Simulate the action of a 'closure copy' bytecode whose result is the
	 new BlockClosure for the following code"
	| copiedValues |
	numCopied > 0
		ifTrue:
			[copiedValues := Array new: numCopied.
			 numCopied to: 1 by: -1 do:
				[:i|
				copiedValues at: i put: self pop]]
		ifFalse:
			[copiedValues := nil].
	self push: (BlockClosure new
					outerContext: self
					startpc: pc
					numArgs: numArgs
					copiedValues: copiedValues).
	self jump: blockSize

pushConsArrayWithElements: numElements
	| array |
	array := Array new: numElements.
```

```
        numElements to: 1 by: -1 do:
                [:i|
                array at: i put: self pop].
        self push: array

pushNewArrayOfSize: arraySize
        self push: (Array new: arraySize)

pushRemoteTemp: remoteTempIndex inVectorAt: tempVectorIndex
        "Simulate the action of bytecode that pushes the value at
remoteTempIndex
         in one of my local variables being used as a remote temp vector."
        self push: ((self at: tempVectorIndex + 1) at: remoteTempIndex + 1)

storeIntoRemoteTemp: remoteTempIndex inVectorAt: tempVectorIndex
        "Simulate the action of bytecode that stores the top of the stack at
         an offset in one of my local variables being used as a remote temp
vector."

        (self at: tempVectorIndex + 1) at: remoteTempIndex + 1 put: self top
```

This level is suitable for prototyping bytecodes. We can test real methods using the above definitions before we attempt to build a VM. Here's an example I used to do just that. runSimulated: is the entry-point into the execution simulation machinery.

```
        | m |
        m := ((Parser new
                                encoderClass: EncoderForV3PlusClosures;
                                parse: 'foo: n | nfib |
                                                        nfib := [:i| i <= 1 ifTrue: [1] ifFalse:
[(nfib value: i – 1) + (nfib value: i – 2) + 1]].
                                                        ^(1 to: n) collect: nfib'
                                class: Object)
                        generate: #(0 0 0 0)).
        ContextPart runSimulated: [#receiver withArgs: #(10)
executeMethod: m] #(1 3 5 9 15 25 41 67 109 177)
```

which looks like



Using this one can test a new set of bytecodes entirely at the image level using the Smalltak debugger to fix problems; a little easier than descending into the VM. Once InstructionStream, ContextPart, MethodContext and BlockContext changes are in place one has a good specification for the VM implementation proper in Interpreter.

Implementing the Closure Bytecodes in the Interpreter VM

The Squeak VM is implemented in an unusual way. It is written in a subset of Smalltalk called Slang which is subsequently translated to C via the VMMaker. The C code is then compiled to create the actual VM. But the Slang code can be executed with a real Smalltalk image to allow debugging the VM in Smalltalk. This is pretty cool and I've been using it a lot recently to create the Stack VM, of which more in subsequent posts. Not everything works. Some plugins do not (yet) function when run in Smalltalk. So one can only debug the VM so far. But in working on the Stack VM I was able to execute the first 195,000 bytecodes in Smalltalk and got a lot of the code working before having to go to C. Very nice.

So implementing the closure bytecodes in the Interpreter VM actually involves modifying the Slang Smalltalk code in the class Interpreter in the VMMaker package. VMMaker isn't in a normal release image. You'll have to download VMMaker, e.g. from SqueakMap. If you're not familiar with

VMMaker please use Tim Rowledge's page above as a starting point. I'm not going to reiterate that information here. And of course you can still read the code without producing your own VM.

To implement closures at the VM level we need to

- – add the methods implementing the closure bytecodes and the BlockClosure value primitives
- – modify the return bytecodes so that method return from within a closure activation still does a non-local return
- – tell the VM what indices to use for the bytecdes and primitives
- – tell the VM where to find class BlockClosure and what is its layout.

The latter two tasks are done by modifying methods on the class side of Interpreter. There are methods two that define the bytecode and primitive tables. I've elided much of the code for clarity. The changed entries are for bytecodes 138 through 142 and 143 and for primitives 200 through 229. Along with the two closure value primitives I've added some compatibility primitives for Context access that we'll need in the Stack VM.

Interpreter class methods for initialization

```
initializeBytecodeTable
    "Interpreter initializeBytecodeTable"
    "Note: This table will be used to generate a C switch statement."

    BytecodeTable := Array new: 256.
    self table: BytecodeTable from:
    #(
        ( 0 15 pushReceiverVariableBytecode)

        …

        (137 pushActiveContextBytecode)
        (138 pushNewArrayBytecode)
        (139 unknownBytecode)
        (140 pushRemoteTempLongBytecode)
        (141 storeRemoteTempLongBytecode)
        (142 storeAndPopRemoteTempLongBytecode)
        (143 pushClosureCopyCopiedValuesBytecode)

        …

        (208 255 sendLiteralSelectorBytecode)
    ).

initializePrimitiveTable
    "This table generates a C function address table use in primitiveResponse along with dispatchFunctionPointerOn:in:"

    "NOTE: The real limit here is 2047 because of the method header layout but there is no point in going over the needed size"
    MaxPrimitiveIndex := 575.
    PrimitiveTable := Array new: MaxPrimitiveIndex + 1.
    self table: PrimitiveTable from:
    #(      "Integer Primitives (0-19)"
        (0 primitiveFail)
        (1 primitiveAdd)

        …
```

```
            "new closure primitives (were Networking primitives)"
            (200 primitiveClosureCopyWithCopiedValues)
            (201 primitiveClosureValue) "value"
            (202 primitiveClosureValue) "value:"
            (203 primitiveClosureValue) "value:value:"
            (204 primitiveClosureValue) "value:value:value:"
            (205 primitiveClosureValue) "value:value:value:value:"
            (206 primitiveClosureValueWithArgs) "valueWithArguments:"

            (207 209 primitiveFail) "reserved for Cog primitives"

            (210 primitiveAt)              "Compatibility with Cog
StackInterpreter Context primitives"
            (211 primitiveAtPut)       "Compatibility with Cog
StackInterpreter Context primitives"
            (212 primitiveSize)       "Compatibility with Cog
StackInterpreter Context primitives"
            (213 219 primitiveFail) "reserved for Cog primitives"

            …

            "Unassigned Primitives"
            (575 primitiveFail)).
```

The VM and image need to share a set of objects so that the VM can actually do anything. This sharing is done using an Array containing all the objects needed by the VM at fixed indices. The VM finds the specialObjectsArray in the first place through its offset in the image file, which is stored in the image file's header. You can inspect it using

```
Smalltalk specialObjectsArray
```

and the method that redefines it is SystemDictionary>>recreateSpecialObjectsArray. The VM's dual of this method is ObjectMemory class>>initializeSpecialObjectIndices. We need to add class BlockClosure to the array at the image level and access it from the VM through the assigned index. Note that the image's indices are 1-relative while the VM's are zero-relative. Again I've elided code for clarity.

SystemDictionary methods for special objects

```
recreateSpecialObjectsArray
    "Smalltalk recreateSpecialObjectsArray"
    "The Special Objects Array is an array of object pointers used by the
     Squeak virtual machine. Its contents are critical and unchecked, so
     don't even think of playing here unless you know what you are
doing."
    | newArray |
    newArray := Array new: 50.
    "Nil false and true get used throughout the interpreter"
    newArray at: 1 put: nil.
    newArray at: 2 put: false.
    newArray at: 3 put: true.

        …

    newArray at: 37 put: BlockClosure.

        …

    newArray at: 49 put: #aboutToReturn:through:.
    newArray at: 50 put: #run:with:in:.
    "Now replace the interpreter's reference in one atomic operation"
    self specialObjectsArray become: newArray
```

ObjectMemory class methods for initialization
initializeSpecialObjectIndices
"Initialize indices into specialObjects array."

NilObject := 0.
FalseObject := 1.
TrueObject := 2.

…

ClassBlockClosure := 36.
…

SelectorAboutToReturn := 48.
SelectorRunWithIn := 49.

Since BlockClosure is one of the execution classes it makes sense (to me) to define its layout along with MethodContext:

Interpreter class methods for initialization
initializeContextIndices
"Class MethodContext"
SenderIndex := 0.
InstructionPointerIndex := 1.
StackPointerIndex := 2.
MethodIndex := 3.
ClosureIndex := 4. "N.B. Called receiverMap in the image."
ReceiverIndex := 5.
TempFrameStart := 6. "Note this is in two places!"

"Class BlockContext"
CallerIndex := 0.
BlockArgumentCountIndex := 3.
InitialIPIndex := 4.
HomeIndex := 5.

"Class BlockClosure"
ClosureOuterContextIndex := 0.
ClosureStartPCIndex := 1.
ClosureNumArgsIndex := 2.
ClosureCopiedValuesIndex := 3

Now we can actually implement the bytecodes and primitives in the VM. The bytecode implementations correspond one-to-one with the methods on ContextPart above except for the pushNewArrayBytecode which InstructionStream>>interpretExtension:in:for: demultiplexed into ContextPart>>pushConsArrayWithElements: & ContextPart>>pushNewArrayOfSize:.

Interpreter methods for stack bytecodes
pushNewArrayBytecode
| size popValues array |
size := self fetchByte.
popValues := size > 127.
size := size bitAnd: 127.
self fetchNextBytecode.
self externalizeIPandSP.
array := self instantiateClass: (self splObj: **ClassArray**)
indexableSize: size.
self internalizeIPandSP.
popValues ifTrue:
[0 to: size – 1 do:

```
                [:i|
                "Assume: have just allocated a new Array; it must be
young. Thus, can use unchecked stores."
                self storePointerUnchecked: i ofObject: array withValue:
(self internalStackValue: size – i – 1)].
            self internalPop: size].
        self internalPush: array
```

The externalizeIPandSP/internalizeIPandSP pair are confusing, not to mention being a pain in the proverbial. But they have fair performance justification. The VMMaker does a lot of inlining as it translates the Slang methods to C so that the bulk of the interpreter minus the primitives, and some less frequently evaluated support functions ends up being one very large C function about 5,000 lines long. The interpreter needs to access its instructionPointer stackPointer and homeContext very often and would like the C compiler to put these in registers. Some primitives, notably the process switch, block evaluation and perform primitives also need to access these variables, since these primitives create new method activations or switch between them. The garbage collector also needs access since it may move the homeContext or the current method.

I suppose one could attempt to inline everything but this might get rather unweildy. The clever hack in the current Interpreter is to have two copies of these variables, the internal variables localIP, localSP and localHomeContext, which are local to the single interpreter function, and the global varables instructionPointer, stackPointer and homeContext which are accessible to primitives and garbage collector as required. The C compiler will hopefully assign the local variables to registers since they have function scope, but this leaes the poor VM programmer with the responsbility of copying the values to and forth between the two sets of varables at the right times, and knowing which ones to access when and where. One of these times is around any allocation that could cause a garbage collection.

The data movement bytecodes are simple and quite similar to their ContextPart duals:

```
Interpreter methods for stack bytecodes
pushRemoteTempLongBytecode
    | remoteTempIndex tempVectorIndex |
    remoteTempIndex := self fetchByte.
    tempVectorIndex := self fetchByte.
    self fetchNextBytecode.
    self pushRemoteTemp: remoteTempIndex inVectorAt:
tempVectorIndex

pushRemoteTemp: index inVectorAt: tempVectorIndex
    | tempVector |
    tempVector := self temporary: tempVectorIndex.
    self internalPush: (self fetchPointer: index ofObject: tempVector)

storeAndPopRemoteTempLongBytecode
    self storeRemoteTempLongBytecode.
    self internalPop: 1

storeRemoteTempLongBytecode
    | remoteTempIndex tempVectorIndex |
    remoteTempIndex := self fetchByte.
    tempVectorIndex := self fetchByte.
    self fetchNextBytecode.
```

```
	self storeRemoteTemp: remoteTempIndex inVectorAt:
tempVectorIndex

storeRemoteTemp: index inVectorAt: tempVectorIndex
	| tempVector |
	tempVector := self temporary: tempVectorIndex.
	self storePointer: index ofObject: tempVector withValue: self
internalStackTop.
```

Creating a closure is a little more involved. The bytecode has to retrieve any copied values from the stack and put them in an Array. It then has to create the closure and jump around the closure's bytecodes, just as the existing BlockContext implementation does. The implementatin is split into two functions since there is a primitive for creating closures also and the bytecode and primitive can share code.

Interpreter methods for stack bytecodes

```
pushClosureCopyCopiedValuesBytecode
	"The compiler has pushed the values to be copied, if any. Find
numArgs and numCopied in the byte following.
	 Pop numCopied values off the stack int an Array (or use nil if none).
Create a Closure with the copiedValues
	 and numArgs so specified, starting at the pc following the block size
and jump over that code."
	| newClosure numArgsNumCopied numArgs numCopied
copiedValues offset |
	numArgsNumCopied := self fetchByte.
	numArgs := numArgsNumCopied bitAnd: 16rF.
	numCopied := numArgsNumCopied bitShift: -4.
	numCopied > 0 ifTrue:
		["self assert: numCopied * BytesPerWord <= 252."
		 self externalizeIPandSP. "This is a pain."
		 copiedValues := self
					instantiateSmallClass: (self splObj:
ClassArray)
					sizeInBytes: (numCopied *
BytesPerWord) + BaseHeaderSize.
		 self internalizeIPandSP.
		 0 to: numCopied – 1 do:
			[:i|
			"Assume: have just allocated a new Array; it must be
young. Thus, can use unchecked stores."
			self storePointerUnchecked: i ofObject: copiedValues
withValue: (self internalStackValue: numCopied – i – 1)].
		 self internalPop: numCopied.
		 self pushRemappableOop: copiedValues].
	"Split offset := (self fetchByte * 256) + self fetchByte. into two
because evaluation order in C is undefined."
	offset := self fetchByte * 256.
	offset := offset + self fetchByte.
	self externalizeIPandSP. "This is a pain."
	newClosure := self
				closureCopyNumArgs: numArgs
				instructionPointer: ((self cCoerce: localIP to:
'sqInt') + 2 – (method+BaseHeaderSize)).
	reclaimableContextCount := 0. "The closure refers to thisContext so
it can't be reclaimed."
	numCopied > 0
		ifTrue: [copiedValues := self popRemappableOop]
		ifFalse: [copiedValues := nilObj].
```

```
        self storePointerUnchecked: ClosureCopiedValuesIndex ofObject:
newClosure withValue: copiedValues.
        self internalizeIPandSP.
        localIP := localIP + offset.
        self fetchNextBytecode.
        self internalPush: newClosure
```

The pushRemappableOop:/popRemappableOop pairs are also confusing and a pain in the proverbial. Since, in the current Interpreter, any allocation can invoke the garbage collector, intermediate objects must be handled carefully if the garbage collector may potentially move them. The painfully error-prone solution is to require the programmer to store intermedate objects in a "remap" buffer prior to any allocation. The garbage collector updates the remap buffer if invoked. After any allocation the programmer must remember to retrieve the intermediate objects. What is error-prone is that it is not necessarily obvious when a function call might include an allocation. It is obvious for the allocation functons, but not at all obvous in e.g. primitivePerform. that calling findNewMethodInClass: will potentially cause a garage collection creating a message if the send is not understood, a bug that has lurked undetected for some time. Conseqently the pushRemappableOop:/popRemappableOop scheme is something that the StackVM is far less dependent upon.

The computation of offset above us split into two lines. We'd rather write it as

```
        offset := self fetchByte * 256 + self fetchByte.
```

which is fine in Smalltalk because of its well-defined left-to-right execution order. But this would translate to

```
        offset = ((byteAtPointer(++localIP)) * 256) +
byteAtPointer(++localIP);
```

which isn't well-defined in C.

Interpreter methods for control primitives

```
closureCopyNumArgs: numArgs instructionPointer: initialIP
        | newClosure |
        self inline: true.
        newClosure := self
                                instantiateSmallClass: (self splObj:
ClassBlockClosure)
                                sizeInBytes: (BytesPerWord * 4) +
BaseHeaderSize.
        "Assume: have just allocated a new closure; it must be young. Thus,
can use unchecked stores."
        "N.B. It is up to the caller to store the copiedValues!"
        self storePointerUnchecked: ClosureOuterContextIndex ofObject:
newClosure withValue: activeContext.
        self storePointerUnchecked: ClosureStartPCIndex ofObject:
newClosure withValue: (self integerObjectOf: initialIP).
        self storePointerUnchecked: ClosureNumArgsIndex ofObject:
newClosure withValue: (self integerObjectOf: numArgs).
        ^newClosure
```

The primitive counterpart to the closure creation bytecode shares the above. It is much simpler since it gets its parameters as arguments.

```
Interpreter methods for control primitives
primitiveClosureCopyWithCopiedValues
	| newClosure numArgs copiedValues |
	"self assert: (self stackValue: 2) == activeContext."
	numArgs := self stackIntegerValue: 1.
	successFlag ifFalse:
		[^self primitiveFail].
	newClosure := self
					closureCopyNumArgs: numArgs
							"greater by 1 due to preIncrement of localIP"
					instructionPointer: instructionPointer + 2 – (method+BaseHeaderSize).
	copiedValues := self stackTop.
	self storePointerUncheckedAsserting: ClosureCopiedValuesIndex ofObject: newClosure withValue: copiedValues.
	self pop: 3 thenPush: newClosure
```

Then there are the two closure evaluation primitives, primitiveClosureValue & primitiveClosureValueWithArgs. We need only look at one here as they're quite similar.

```
Interpreter methods for control primitives
primitiveClosureValue
	| blockClosure blockArgumentCount closureMethod copiedValues outerContext |
	blockClosure := self stackValue: argumentCount.
	blockArgumentCount := self argumentCountOfClosure: blockClosure.
	argumentCount = blockArgumentCount ifFalse:
		[^self primitiveFail].

	"Somewhat paranoiac checks we need while debugging that we may be able to discard
	 in a robust system."
	outerContext := self fetchPointer: ClosureOuterContextIndex ofObject: blockClosure.
	(self isContext: outerContext) ifFalse:
		[^self primitiveFail].
	closureMethod := self fetchPointer: MethodIndex ofObject: outerContext.
	"Check if the closure's method is actually a CompiledMethod."
	((self isNonIntegerObject: closureMethod) and: [self isCompiledMethod: closureMethod]) ifFalse:
		[^self primitiveFail].
	"Check if copiedValues is either nil or anArray."
	copiedValues := self fetchPointer: ClosureCopiedValuesIndex ofObject: blockClosure.
	(copiedValues == nilObj or: [(self fetchClassOf: copiedValues) = (self splObj: ClassArray)]) ifFalse:
		[^self primitiveFail].

	self activateNewClosureMethod: blockClosure
```

Activating the method is very similar to activating a method for a message send (see activateNewMethod). The main difference is that closure activation pushes any copied vaues and does not initialize temporaries, leaving it to bytecodes inserted by the compiler to do that (TSSTCPW). Another simplification is that if a method or any block within it needs a large context (becaue it has a large stack) then all blocks within it do also.

```
Interpreter methods for control primitives
activateNewClosureMethod: blockClosure
	"Similar to activateNewMethod but for Closure and newMethod."
	| theBlockClosure closureMethod newContext methodHeader
copiedValues numCopied where outerContext |

	outerContext := self fetchPointer: ClosureOuterContextIndex
ofObject: blockClosure.
	closureMethod := self fetchPointer: MethodIndex ofObject:
outerContext.
	methodHeader := self headerOf: closureMethod.
	self pushRemappableOop: blockClosure.
	newContext := self allocateOrRecycleContext: (methodHeader
bitAnd: LargeContextBit). "All for one, and one for all!"

	"allocateOrRecycleContext: may cayse a GC; restore blockClosure
and refetch outerContext et al"
	theBlockClosure := self popRemappableOop.
	outerContext := self fetchPointer: ClosureOuterContextIndex
ofObject: theBlockClosure.

	copiedValues := self fetchPointer: ClosureCopiedValuesIndex
ofObject: theBlockClosure.
	"Should evaluate to 0 for nilObj"
	numCopied := self fetchWordLengthOf: copiedValues.

	"Assume: newContext will be recorded as a root if necessary by the
	 call to newActiveContext: below, so we can use unchecked stores."
	where := newContext + BaseHeaderSize.
	self longAt: where + (SenderIndex << ShiftForWord)
		put: activeContext.
	self longAt: where + (InstructionPointerIndex << ShiftForWord)
		put: (self fetchPointer: ClosureStartPCIndex ofObject:
theBlockClosure).
	self longAt: where + (StackPointerIndex << ShiftForWord)
		put: (self integerObjectOf: argumentCount + numCopied).
	self longAt: where + (MethodIndex << ShiftForWord)
		put: (self fetchPointer: MethodIndex ofObject: outerContext).
	self longAt: where + (ClosureIndex << ShiftForWord)
		put: theBlockClosure.
	self longAt: where + (ReceiverIndex << ShiftForWord)
		put: (self fetchPointer: ReceiverIndex ofObject: outerContext).

	"Copy the arguments…"
	1 to: argumentCount do:
		[:i | self longAt: where + ((ReceiverIndex+i) << ShiftForWord)
put: (self stackValue: argumentCount-i)].

	"Copy the copied values…"
	where := newContext + BaseHeaderSize + ((ReceiverIndex + 1 +
argumentCount) << ShiftForWord).
	0 to: numCopied – 1 do:
		[:i|
		self longAt: where + (i << ShiftForWord) put: (self
fetchPointer: i ofObject: copiedValues)].

	"The initial instructions in the block nil-out remaining temps."

	self pop: argumentCount + 1.
	self newActiveContext: newContext
```

We do have to make one crucial change to activateNewMethod and internalActvateNewMethod (because yes, dear reader, the local vs global variable scheme necessitates that there be two almost identical versions of them). Now that the VM is using the old receiverMap slot, now the closure slot, it must be set to nil on method activation. Here's actvateNewMethod, which you can compare to activateNewClosureMethod:.

```
Interpreter methods for message sending
activateNewMethod
    | newContext methodHeader initialIP tempCount nilOop where |

    methodHeader := self headerOf: newMethod.
    newContext := self allocateOrRecycleContext: (methodHeader
bitAnd: LargeContextBit).

    initialIP := ((LiteralStart + (self literalCountOfHeader:
methodHeader)) * BytesPerWord) + 1.
    tempCount := (methodHeader >> 19) bitAnd: 16r3F.

    "Assume: newContext will be recorded as a root if necessary by the
     call to newActiveContext: below, so we can use unchecked stores."

    where := newContext + BaseHeaderSize.
    self longAt: where + (SenderIndex << ShiftForWord) put:
activeContext.
    self longAt: where + (InstructionPointerIndex << ShiftForWord)
put: (self integerObjectOf: initialIP).
    self longAt: where + (StackPointerIndex << ShiftForWord) put:
(self integerObjectOf: tempCount).
    self longAt: where + (MethodIndex << ShiftForWord) put:
newMethod.
    self longAt: where + (ClosureIndex << ShiftForWord) put: nilObj.

    "Copy the reciever and arguments…"
    0 to: argumentCount do:
        [:i | self longAt: where + ((ReceiverIndex+i) << ShiftForWord)
put: (self stackValue: argumentCount-i)].

    "clear remaining temps to nil in case it has been recycled"
    nilOop := nilObj.
    argumentCount+1+ReceiverIndex to: tempCount+ReceiverIndex
do:
        [:i | self longAt: where + (i << ShiftForWord) put: nilOop].

    self pop: argumentCount + 1.
    reclaimableContextCount := reclaimableContextCount + 1.
    self newActiveContext: newContext
```

Finally we have to make sure that non-local return still functions. This turns out ot be very easy since the existing code uses Interpreter>>sender to discover the contet to return to. Prior to closures its implementation is

```
Interpreter methods for contexts
sender
    ^ self fetchPointer: SenderIndex ofObject: localHomeContext
```

This changes if the active context is a closure. If so the lexical chain has to be walked to find the outermost (method) activation:

```
Interpreter methods for contexts
sender
```

```
	| context closureOrNil |
	context := localHomeContext.
	[(closureOrNil := self fetchPointer: ClosureIndex ofObject: context)
~~ nilObj] whileTrue:
		[context := self fetchPointer: ClosureOuterContextIndex
ofObject: closureOrNil].
	^self fetchPointer: SenderIndex ofObject: context
```

Voila, closures in the VM. Performance appears to be no worse than the BlockContext scheme. The VM can run old and new images alike, and mix closure and blue-book block code side-by-side, simplifying the next step in the bootstrap, which is to recompile the system using closures. Which brings us to the next post which will cover the bytecode compiler for closures. And which brings us to the code for the scheme so far. The bootstrap is in www.mirandabanda.org/files/Cog/Closures0808/Bootstrap/. Warning, this is for brave and hardy souls only, and has been tested only for Croquet.1.0.18 and Squeak3.9-final-7067. Enjoy!

Posted by admin on Tuesday, July 22nd, 2008, at 6:01 pm, and filed under Cog.
Follow any responses to this entry with the RSS 2.0 feed.
You can post a comment.

{ 4 }

Comments

1. **Paolo Bonzini** | 23-Jul-08 at 12:00 am | Permalink

   Nice recap.

   I disagree that “interpreting the short forms is a lot quicker than interpreting the longer forms.” It causes a lot of code duplication and icache thrashing. The best thing would be to have a simple encoding common to all bytecodes as in the specification I sent to you. (That specification has a 40% performance increase on P4 compared to the Blue Book bytecode set; probably a bit less on more modern architectures).

   As I mentioned in the post one representative code sample that counted all the bytecodes in the image was 40% slower when using the long-forms to do the counting than the short-forms. So it really is quicker to interpret the short forms. I think the Squeak interpreter is small enough that it all fits in cache anyway.

   However, I agree that given the structure of the Squeak bytecode set, those instructions are faster.

   One question is, why don’t you store the copied values in the indexed instance variables of BlockClosure? Do you want to ease adding further instance variables to BlockClosure?

   I stared doing TSSTCPW. Then I hummed and hawed over changing it but I haven’t yet accumulated any good benchmarks to determine if there is any performance difference. It is not too late to change it, and personally I prefer the use of indexable inst vars.

Thanks!

You're so welcome!

Cheers,
Eliot

2. **Ken Causey** | 24-Jul-08 at 3:47 pm | Permalink

   I really really appreciate the level of detail you are willing to go into in these posts. I've learned a lot from them so far and I'm certain that others will as well.

3. **Michael Haupt** | 06-Jan-09 at 6:33 am | Permalink

   Eliot,

   thank you so much for this series of posts, which I finally came around to read today. Implicitly, it's a great tutorial on Squeak VM hacking, which I very much appreciate. 🙂

   In this post, I found a little glitch – just after the text saying "and now I can see what Collection>>inject:into: looks like:", some method implementations are given without mentioning where they belong. I believe they go to ContextPart. Is that right?

   Best,

   Michael

4. **Eliot Miranda** | 06-Jan-09 at 1:12 pm | Permalink

   Hi Michael,

   good catch! I had written "Collection>>inject:into: inside an <img src= … tag. Hence a lot of code was missing.

   Thanks!

Post a Comment

Your email is *never* published nor shared. Required fields are marked *

Name *

Email *

Website

Message

Post

« BLUEBOOK COMPILEDMETHODS – HAVING OUR CAKE AND EATING IT TOO

CLOSURES PART III – THE COMPILER »