**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

{ 2014 02 08 }

# Primitives and the Partial Read Barrier

## Aperitivo

If you've read my post on the partial read barrier you'll know that one of the issues in the new Spur VM is how primitives deal with forwarding objects amongst their arguments. Recall that become: is lazy and is implemented by copying objects and updating the originals to be forwarding pointers to the copies. Message lookup fails for forwarded receivers because they have a reserved class index, so sends to forwarded objects are already handled. But nothing is done for message arguments, or references to forwarding objects from instance variables of the receiver or arguments, and so primitives must somehow cope with forwarders anywhere in their parameters. But first I need briefly to explain primitives in a Smalltalk virtual machine; IMO they're another one of Dan Ingalls' strokes of genius; not as all pervasive or influential as BitBlt, but every bit as beautiful.

## Antipasto

Smalltalk's bytecode doesn't include any operators, only data manipulation (variable and argument access), control flow (jump, return and closure creation bytecodes), and message sends. Typically a range of the message send bytecodes (bytecodes 176 through 207 in Squeak) code for common messages such as #+, #-, #> #< et al, and these can be implemented directly in the interpreter's dispatch loop if the receiver and argument are of the right type. In the Squeak VM for example, these arithmetic selectors will be evaluated inline if the receiver and argument are SmallIntegers or Floats. If the types are other than these then the virtual machine simply sends the corresponding message. So these bytecodes accomplish two things, they save space, because they encode their selector without having to store the literal selector in the method (these special selectors are stored in an Array known to the VM), /and/ they allow an interpreter to short-cut the overhead of a send for the common case of arithmetic on SmallIntegers and Floats. This latter optimization is called static receiver type prediction. But a Smalltalk system needs more primitives than merely the arithmetic selectors; it needs object access (at: and at:put:), instantiation (new, basicNew:) and so on. Dan's beautiful invention was how to knit these primitive operations into methods. In Smalltalk primitives can only be associated with a method. Let's take Object>>#at: for example, that's Smalltalk's array dereferencing message:

```
Object methods for accessing
at: index
	"Primitive. Assumes receiver is indexable. Answer the value of an
	indexable element in the receiver. Fail if the argument index is not an
	Integer or is out of bounds. Essential. See Object documentation whatIsAPrimitive."

	<primitive: 60>
	index isInteger ifTrue:
		[self class isVariable
			ifTrue: [self errorSubscriptBounds: index]
			ifFalse: [self errorNotIndexable]].
	index isNumber
		ifTrue: [^self at: index asInteger]
		ifFalse: [self errorNonIntegerIndex]
```

The primitive operation is signified by `<primitive: 60>`. In the VM is a table with room for up to 512 numbered primitives, and in the VM's primitive table at index 60 is a pointer to the primitiveAt routine. In the header of each CompiledMethod is a 9 bit field that defines the method's primitive, with 0 meaning no primitive. When a message is activated, immediately after being sent, the VM checks for the existence of a primitive and, if present, evaluates it (if there's a 0 in the table at a particular index the VM merely continues and activates the method). The primitive validates the receiver and arguments (its parameters) and if they are correct the primitive performs its operation and returns the operation's result just as if it was the result of the method. If the validation fails then the primitive leaves its parameters entirely unchanged and the VM activates the body of the method. So (and this is genius) *primitives either succeed and return a result, or fail without side-effects*.

This means for example that some primitives can be optional, added merely to speed up computation. For example in the old days there used to be a complex text display method scanCharactersFrom:…. with some 7 arguments, that if implemented well would make text display much faster, but was complex enough that a VM implementor's job was made easier by leaving it out. One could get to a working albeit slow system that little bit earlier by not implementing scan characters until later.

## Primo

For Spur, failing without side effects means that failing primitives can be retried. If primitives fail when they encounter forwarding pointers then somehow the VM should be able to find those forwarding pointers, follow them and retry the primitive, with no visible effect. Of course the validation step means that primitives already fail when encountering a forwarding pointer, because we can arrange that a forwarding pointer will fail for the specific validation steps. What do I mean? Here's a good example, primitive 105, primitiveStringReplace. It's a general array copying primitive used to speed up various implementations of #replaceFrom:to:with:startingAt:. Evaluate the following in your Squeak/Pharo/Quis image:

```
	self systemNavigation browseAllSelect: [:m| m primitive = 105]
```

Amongst the results you'll get will be

```
Array methods for accessing
replaceFrom: start to: stop with: replacement startingAt: repStart
	"Primitive. This destructively replaces elements from start to stop in the receiver starting at index, repStart, in the collection, replacement. Answer the receiver. Range cl

	<primitive: 105>

	super replaceFrom: start to: stop with: replacement startingAt: repStart

ByteArray methods for accessing
replaceFrom: start to: stop with: replacement startingAt: repStart
	"Primitive. This destructively replaces elements from start to stop in the receiver starting at index, repStart, in the collection, replacement. Answer the receiver. Range cl

	<primitive: 105>

	replacement class == WideString ifTrue:
		[self becomeForward: (WideString from: self)].

	super replaceFrom: start to: stop with: replacement startingAt: repStart.
```

And the general method they're optimized substitutes for is

```
SequenceableCollection methods for accessing
replaceFrom: start to: stop with: replacement startingAt: repStart
	"This destructively replaces elements from start to stop in the receiver
	starting at index, repStart, in the sequenceable collection,
	replacementCollection. Answer the receiver. No range checks are performed."

	| index repOff |
	repOff := repStart - start.
	index := start - 1.
	[(index := index + 1) <= stop]
		whileTrue: [self at: index put: (replacement at: repOff + index)]
```

The primitive can be much faster than the generic SequenceableCollection method because bounds checks can be performed once, as part of the primitive's validation step, not twice for each element, in the sends of at: and at:put:. But if the primitive is missing the system will keep on running, albeit slower *(until the VM implements speculative inlining and performs the optimization itself at run-time, and Clément Bera is working on precisely that)*.

Not only that, looking at the ByteString method we can see that the primitive doesn't have to work for all possible parameters, and instead focus on the common case. The primitive simply fails if an attempt is made to replace characters in a ByteString with those from a WideString, and the fully polymorphic Smalltalk code takes over. Cool.

So let's dive into primitiveStringReplace's implementation (you might want to skip forward where I break it down before trying to grok this in one go):

```
InterpreterPrimitives methods for array primitives
primitiveStringReplace
    "primReplaceFrom: start to: stop with: replacement startingAt: repStart"
    | array start stop repl replStart hdr arrayFmt totalLength arrayInstSize replFmt replInstSize srcIndex |
    array := self stackValue: 4.
    start := self stackIntegerValue: 3.
    stop := self stackIntegerValue: 2.
    repl := self stackValue: 1.
    replStart := self stackIntegerValue: 0.

    self successful ifFalse: [^ self primitiveFail].

    hdr := objectMemory baseHeader: array.
    arrayFmt := objectMemory formatOfHeader: hdr.
    totalLength := objectMemory lengthOf: array baseHeader: hdr format: arrayFmt.
    arrayInstSize := objectMemory fixedFieldsOf: array format: arrayFmt length: totalLength.
    (start >= 1 and: [start - 1 <= stop and: [stop + arrayInstSize <= totalLength]])            ifFalse: [^ self primitiveFailFor: PrimErrBadIndex].     hdr := objectMemory baseH
        ifFalse: [^ self primitiveFailFor: PrimErrBadIndex].

    "Array formats (without byteSize bits, if bytes array) must be same "
    arrayFmt < objectMemory firstByteFormat
        ifTrue: [arrayFmt = replFmt
                    ifFalse: [^ self primitiveFailFor: PrimErrInappropriate]]
        ifFalse: [(arrayFmt bitAnd: objectMemory byteFormatMask) = (replFmt bitAnd: objectMemory byteFormatMask)
                    ifFalse: [^ self primitiveFailFor: PrimErrInappropriate]].

    srcIndex := replStart + replInstSize - 1. "- 1 for 0-based access"

    arrayFmt <= objectMemory lastPointerFormat
        ifTrue:
            [start + arrayInstSize - 1 to: stop + arrayInstSize - 1 do: [:i |
                objectMemory storePointer: i ofObject: array withValue: (objectMemory fetchPointer: srcIndex ofObject: repl).
                    srcIndex := srcIndex + 1]]
        ifFalse:
            [arrayFmt < objectMemory firstByteFormat
                ifTrue: "32-bit-word type objects"
                    [start + arrayInstSize - 1 to: stop + arrayInstSize - 1
                        do: [:i | objectMemory storeLong32: i ofObject: array withValue: (objectMemory fetchLong32: srcIndex ofObject: repl).
                            srcIndex := srcIndex + 1]]
                ifFalse: "byte-type objects"
                    [start + arrayInstSize - 1 to: stop + arrayInstSize - 1
                        do: [:i |  objectMemory storeByte: i ofObject: array withValue: (objectMemory fetchByte: srcIndex ofObject: repl).
                            srcIndex := srcIndex + 1]]].

    self pop: argumentCount "leave rcvr on stack"
```

Wow, that's quite a lot of code. Let's look at the first few lines and their supporting definitions. Smalltalk's a stack machine where the receiver is pushed first, followed by the arguments, so the receiver is stackValue: 4, and replStart, the last argument is stackValue: 0. So the first few lines fetch the parameters and, via stackIntegerValue:, set the primitive failure flag if any of start, stop or replStart are not SmallIntegers.

```
    array := self stackValue: 4.
    start := self stackIntegerValue: 3.
    stop := self stackIntegerValue: 2.
    repl := self stackValue: 1.
    replStart := self stackIntegerValue: 0.

    self successful ifFalse: [^ self primitiveFail].

StackInterpreter methods for internal interpreter access
stackIntegerValue: offset
    "In the StackInterpreter stacks grow down."
    | integerPointer |
    integerPointer := stackPages longAt: stackPointer + (offset*BytesPerWord).
    ^self checkedIntegerValueOf: integerPointer

StackInterpreter methods for utilities
checkedIntegerValueOf: intOop
    "Note: May be called by translated primitive code."

    (objectMemory isIntegerObject: intOop)
        ifTrue: [ ^ objectMemory integerValueOf: intOop ]
        ifFalse: [ self primitiveFail. ^ 0 ]
```

Next are two very similar pieces of code that do the bounds checks. Here's the first one:

```
    hdr := objectMemory baseHeader: array.
    arrayFmt := objectMemory formatOfHeader: hdr.
    totalLength := objectMemory lengthOf: array baseHeader: hdr format: arrayFmt.
    arrayInstSize := objectMemory fixedFieldsOf: array format: arrayFmt length: totalLength.
    (start >= 1 and: [start - 1 <= stop and: [stop + arrayInstSize <= totalLength]])
        ifFalse: [^ self primitiveFailFor: PrimErrBadIndex].
```

Then there's the dispatch based on object type:

```
    "Array formats (without byteSize bits, if bytes array) must be same "
    arrayFmt < objectMemory firstByteFormat
        ifTrue: [arrayFmt = replFmt
                    ifFalse: [^ self primitiveFailFor: PrimErrInappropriate]]
        ifFalse: [(arrayFmt bitAnd: objectMemory byteFormatMask) = (replFmt bitAnd: objectMemory byteFormatMask)
                    ifFalse: [^ self primitiveFailFor: PrimErrInappropriate]].
```

In the classic Squeak VM an objects' format is a 4 bit field, and values 8 through 15 encode byte objects with the least significant two bits being the number of unused bytes to round the length up to a multiple of 4 bytes, the so-called "odd bits". 8 through 11 are for things like ByteArray and ByteString, 12 through 15 are for CompiledMethod. In Spur, which has 8 byte alignment, format is a 5 bit field and 16 through 31 encode byte objects with the least significant 3 bits being used to round up to an 8 byte boundary, 16 through 23 for ByteString et al, and 24 through 31 for CompiledMethods. Hence if the format is >= firstByteFormat, the odd bits have to be ignored to compare for compatible formats.

In Spur, the format of a forwarder is 7, chosen to be between the pointer formats (0 through 5) and the non-pointer formats (9 through 31). So if the receiver is a pointer type and repl is a forwarder then arrayFmt = replFmt will be false and the primitive will fail, and if it is a non-pointer format and repl is a forwarder (arrayFmt bitAnd: objectMemory byteFormatMask) = (replFmt bitAnd: objectMemory byteFormatMask) will be false and the primitive will fail.

I claimed above that primitives already fail when encountering a forwarding pointer, because we can arrange that a forwarding pointer will fail for the specific validation steps. In general this can't be true. Validation is a series of yes/no questions in if-then-elses and if the wrong question is asked, then the primitive won't fail. But in all my reading of the primitives I've not found one written perversely. For example here's part of the file read primitive:

```
FilePlugin methods for file primitives
primitiveFileRead

    | retryCount count startIndex array file elementSize bytesRead |
    <var: 'file' type: #'SQFile *'>
    <var: 'count' type: #'size_t'>
    <var: 'startIndex' type: #'size_t'>
    <var: 'elementSize' type: #'size_t'>

    retryCount      := 0.
    count           := interpreterProxy positive32BitValueOf: (interpreterProxy stackValue: 0).
    startIndex      := interpreterProxy positive32BitValueOf: (interpreterProxy stackValue: 1).

    [array          := interpreterProxy stackValue: 2.
     file           := self fileValueOf: (interpreterProxy stackValue: 3).

     (interpreterProxy failed
     "buffer can be any indexable words or bytes object except CompiledMethod"
     or: [(interpreterProxy isWordsOrBytes: array) not]) ifTrue:
        [^interpreterProxy primitiveFailFor: PrimErrBadArgument].
```

This primitive reads data into some non-pointer object, failing if given a pointer object such as an Array. If that last phrase had been written

```
     (interpreterProxy failed
     "buffer can be any indexable words or bytes object except CompiledMethod"
     or: [interpreterProxy isPointers: array]) ifTrue:
        [^interpreterProxy primitiveFailFor: PrimErrBadArgument].
```

then it wouldn't fail for forwarders. But, somewhat miraculously, the programmers who've authored primitives in the Squeak VM have consistently written code of the form "if we have what we expect, continue", or "(if we have what we expect) not, fail", but not their converses "(if we have what we expect) not, fail" and "(if we don't have what we expect) not, continue". So all the existing primitives' validation codes I've looked at so far successfully fail when encountering a forwarder (most convenient !!).

## Secondo

So now we get to the real issue, and the meat in this post. How does the VM know when and if to look for forwarders, and how much work to do in fixing them up? For example, one simple but extremely slow solution would be to scan the entire heap and the stack on every primitive failure, and follow every forwarding pointer in the heap. This would certainly work for most primitives, and be simple to implement, but it would be dog slow; in my measurements while primitives fail less than 2% of the time that's still hundreds of failures per second on machines of moderate performance, and scanning the entire heap that often will kill performance. Instead the VM needs to know on a per-primitive basis how much state to examine.

The example I'll use is the primitive that sets the current cursor. In Squeak/Pharo Smalltalk a cursor is an object with an image bitmap, that defines the colours of the cursor image, a mask bitmap that defines the visible portions of the image, and hence the cursor's shape. In turn each bitmap is an object containing a non-pointer object holding the bits, width, height and depth integers, and a possibly nil offset point used to translate the image when displaying it. Since Cursor inherits from Form, Smalltalk's bitmap class, its first 6 instance variables are the bits, width, height, depth and offset for the image bitmap, and the offset is used to define the cursor's hot-spot (where it actually points). The mask bitmap is passed into the primitive as a second parameter. So the primitive traverses its input parameters to a depth of two, 0 for the stack to the arguments, 1 for instance variables of the cursor and mask object, and 2 for instance variables of the cursor and mask's offset points:

```
InterpreterPrimitives methods for I/O primitives
primitiveBeCursor
    "Set the cursor to the given shape."

    | cursorObj maskBitsIndex maskObj bitsObj extentX extentY depth offsetObj offsetX offsetY cursorBitsIndex |

    cursorObj := self stackValue: 1.
    maskObj := self stackTop.

    self success: ((objectMemory isPointers: cursorObj) and: [(objectMemory lengthOf: cursorObj) >= 5]).
    self successful ifTrue: [
        bitsObj := objectMemory fetchPointer: 0 ofObject: cursorObj.
        extentX := self fetchInteger: 1 ofObject: cursorObj.
        extentY := self fetchInteger: 2 ofObject: cursorObj.
        depth := self fetchInteger: 3 ofObject: cursorObj.
        offsetObj := objectMemory fetchPointer: 4 ofObject: cursorObj].
        self success: ((objectMemory isPointers: offsetObj) and: [(objectMemory lengthOf: offsetObj) >= 2]).

    self successful ifTrue: [
        offsetX := self fetchInteger: 0 ofObject: offsetObj.
        offsetY := self fetchInteger: 1 ofObject: offsetObj.
        self success: ((extentX = 16) and: [extentY = 16 and: [depth = 1]]).
        self success: ((offsetX >= -16) and: [offsetX <= 0]).
        self success: ((offsetY >= -16) and: [offsetY <= 0]).
        self success: ((objectMemory isWords: bitsObj) and: [(objectMemory lengthOf: bitsObj) = 16]).
        cursorBitsIndex := bitsObj + objectMemory baseHeaderSize].

    self success: ((objectMemory isPointers: maskObj) and: [(objectMemory lengthOf: maskObj) >= 5]).
    self successful ifTrue: [
        bitsObj := objectMemory fetchPointer: 0 ofObject: maskObj.
        extentX := self fetchInteger: 1 ofObject: maskObj.
        extentY := self fetchInteger: 2 ofObject: maskObj.
        depth := self fetchInteger: 3 ofObject: maskObj].

    self successful ifTrue: [
        self success: ((extentX = 16) and: [extentY = 16 and: [depth = 1]]).
        self success: ((objectMemory isWords: bitsObj) and: [(objectMemory lengthOf: bitsObj) = 16]).
        maskBitsIndex := bitsObj + objectMemory baseHeaderSize].

    self successful ifTrue: [
        self cCode: 'ioSetCursorWithMask(cursorBitsIndex, maskBitsIndex, offsetX, offsetY)'.
        self pop: argumentCount]
```

One obvious but tedious and error-prone solution would be to state the depth for each primitive. e.g. using a pragma:

```
primitiveBeCursor
    "Set the cursor to the given shape. The Mac only supports 16x16 pixel cursors. Cursor offsets are handled by Smalltalk."

    <primitiveAccessorDepth: 2>
    ...
```

Then Slang, the Smalltalk-to-C translator, could extract the information from the pragma and store it in an Array of depths, parallel to the primitive table. But the thought of reading over a thousand primitive methods (in addition to the 200 odd core primitives there are a lot of additional plugin primitives such as primitiveFileRead above) is more than I can stomach.

So if I'm to be this lazy I need to find a better way, and given that the system has extensive code manipulation facilities in Slang, and that primitives access their state in quite stereotypical ways, why not do it automatically by analysing the parse trees of the primitives?

All these primitives start by taking their parameters from the stack, so let's define them:

```
StackInterpreter class methods for spur compilation support
isStackAccessor: selector
    ^#( stackTop stackValue: stackTopPut: stackValue:put:
        stackFloatValue: stackIntegerValue: stackObjectValue:) includes: selector
```

The methods used to access objects are more various. We an define them using introspection:

```
StackInterpreter class methods for spur compilation support
isObjectAccessor: selector
	"Answer if selector is one of fetchPointer:ofObject: storePointer:ofObject:withValue:
	 et al."
	^(InterpreterProxy whichCategoryIncludesSelector: selector) = #'object access'
	 or: [(SpurMemoryManager whichCategoryIncludesSelector: selector) = #'object access']
```

InterpreterProxy is the object that defines the functions that can be used by plugin primitives. And for concreteness we can obtain a list via:

```
((InterpreterProxy allMethodsInCategory: #'object access') asSet
	addAll: (SpurMemoryManager allMethodsInCategory: #'object access');
	asArray) sort

#(argumentCountOf: arrayValueOf:
  byteLengthOf: byteSizeOf:
  characterObjectOf: characterTag characterValueOf: compactClassIndexOf:
  fetchArray:ofObject: fetchByte:ofObject: fetchClassOf: fetchClassOfNonImm: fetchFloat:ofObject: fetchInteger:ofObject: fetchLong32:ofObject: fetchPointer:ofObject: fet
  instanceSizeOf: is:instanceOf:compactClassIndex: isClassOfNonImm:equalTo:compactClassIndex:
  keyOfEphemeron:
  lengthOf: lengthOf:baseHeader:format: lengthOf:format: literal:ofMethod: literalCountOf:
  methodArgumentCount methodPrimitiveIndex
  numPointerSlotsOf: numSlotsOf: numSlotsOfAny: numStrongSlotsOf:ephemeronInactiveIf: numTagBits
  obsoleteDontUseThisFetchWord:ofObject:
  pinObject: primitiveIndexOf: primitiveMethod
  rawNumSlotsOf: rawOverflowSlotsOf:
  sizeBitsOf: sizeBitsOfSafe: sizeOfSTArrayFromCPrimitive: slotSizeOf: stObject:at: stObject:at:put: stSizeOf: storeByte:ofObject:withValue: storeInteger:ofObject:withVal
```

The introspective approach is much nicer than a brittle and explicit list. As you can see its quite a large API and it gets extended as new facilities (such as pinning) are added. It would be easy for someone to add an accessor but forget to update the list in isObjectAccessor:, whose existence they may even be entirely unaware of. It's so fabulous having the code of the system in the system. How do people program without that information? [Hint: "sl*w*r"].

We can compile each primitive to an AST, extract the relevant parse nodes, starting from the stack accessors, and build up chains to object accessors, the links being assigning accessors to variables, and passing accessors and variables as parameters to other accessors. For example, in the above the chains, obtained by

```
CCodeGenerator new
	accessorChainsForMethod: ((InterpreterPrimitives >> #primitiveBeCursor) methodNode
						asTranslationMethodOfClass: TMethod)
	interpreterClass: StackInterpreter
```

are

-
  ```
  cursorObj := self stackValue: 1
  bitsObj := objectMemory fetchPointer: 0 ofObject: cursorObj
  ```

-
  ```
  cursorObj := self stackValue: 1
  offsetObj := objectMemory fetchPointer: 4 ofObject: cursorObj
  offsetY := self fetchInteger: 1 ofObject: offsetObj
  ```

-
  ```
  cursorObj := self stackValue: 1
  offsetObj := objectMemory fetchPointer: 4 ofObject: cursorObj
  offsetX := self fetchInteger: 0 ofObject: offsetObj
  ```

-
  ```
  cursorObj := self stackValue: 1
  depth := self fetchInteger: 3 ofObject: cursorObj
  ```

-
  ```
  cursorObj := self stackValue: 1
  extentX := self fetchInteger: 1 ofObject: cursorObj
  ```

-
  ```
  cursorObj := self stackValue: 1
  extentY := self fetchInteger: 2 ofObject: cursorObj
  ```

-
  ```
  maskObj := self stackTop
  extentY := self fetchInteger: 2 ofObject: maskObj
  ```

-
  ```
  maskObj := self stackTop
  depth := self fetchInteger: 3 ofObject: maskObj
  ```

-
  ```
  maskObj := self stackTop
  extentX := self fetchInteger: 1 ofObject: maskObj
  ```

-
  ```
  maskObj := self stackTop
  bitsObj := objectMemory fetchPointer: 0 ofObject: maskObj
  ```

The longest are indeed those that derive offsetX and offsetY. They have length 3, which we'll use as a zero-relative index, i.e. 2.

Going bottom-up here's the method that extracts the relevant parse nodes and evaluates a closure with the root accessor send nodes, the object accessor send nodes and the assignment nodes in the method:

```
CCodeGenerator methods for spur primitive compilation
accessorsAndAssignmentsForMethod: method interpreterClass: interpreterClass into: aTrinaryBlock
	"Evaluate aTrinaryBlock with the root accessor sends, accessor sends and assignments in the method."
	| accessors assignments roots |
	accessors := Set new.
	assignments := Set new.
	roots := Set new.
	method parseTree nodesDo:
		[:node|
		node isSend ifTrue:
			[(interpreterClass isStackAccessor: node selector) ifTrue:
				[roots add: node].
			 (interpreterClass isObjectAccessor: node selector) ifTrue:
				[accessors add: node]].
		(node isAssignment
		 and: [(roots includes: node expression)
			or: [(accessors includes: node expression)
			or: [node expression isVariable and: [node expression name ~= 'nil']]]]) ifTrue:
			[assignments add: node]].
	^aTrinaryBlock
		value: roots
		value: accessors
		value: assignments
```

With that factored out, computing the accessor chains is simply computing the transitive closure of the uses of the roots:

```
CCodeGenerator methods for spur primitive compilation
accessorChainsForMethod: method interpreterClass: interpreterClass
	| accessors assignments roots chains extendedChains extended lastPass |
	self accessorsAndAssignmentsForMethod: method
		interpreterClass: interpreterClass
		into: [:theRoots :theAccessors :theAssignments|
			roots := theRoots.
			accessors := theAccessors.
			assignments := theAssignments].
	"Compute the transitive closure of assignments of accessor sends or variables to variables from the roots.
	 Start from the stack accesses (the roots).
	 On the last pass look only for accessors of the targets of the tip assignments."
	chains := OrderedCollection new.
	roots do: [:root| chains addAll: (assignments
								select: [:assignment| assignment expression = root]
								thenCollect: [:assignment| OrderedCollection with: assignment])].
	lastPass := false.
	[extended := false.
	 extendedChains := OrderedCollection new: chains size * 2.
```

```
    chains do:
        [:chain| | tip refs accessorRefs variableRefs |
        tip := chain last variable.
        refs := accessors select: [:send| send args anySatisfy: [:arg| tip isSameAs: arg]].
        lastPass ifFalse:
            [accessorRefs := refs collect: [:send|
                                                assignments
                                                    detect: [:assignment|
                                                            assignment expression = send
                                                            and: [(chain includes: assignment) not]]
                                                    ifNone: [[]]
                                        thenSelect: [:assignmentOrNil| assignmentOrNil notNil].
             variableRefs := assignments select:
                                    [:assignment|
                                     (tip isSameAs: assignment expression)
                                     and: [(tip isSameAs: assignment variable) not
                                     and: [(chain includes: assignment) not]]].
             refs := (Set withAll: accessorRefs) addAll: variableRefs; yourself].
        refs isEmpty
            ifTrue:
                [extendedChains add: chain]
            ifFalse:
                [lastPass ifFalse: [extended := true].
                 self assert: (refs noneSatisfy: [:assignment| chain includes: assignment]).
                 extendedChains addAll: (refs collect: [:assignment| chain, {assignment}])]].
    extended or: [lastPass not]] whileTrue:
        [chains := extendedChains.
         extended ifFalse: [lastPass := true]].
   ^chains
```

and finally deriving the accessor depth as a number is simple:

```
*CCodeGenerator methods for spur primitive compilation*
accessorDepthForMethod: method
    "Compute the depth the method traverses object structure, assuming it is a primitive.
     This is in support of Spur's lazy become.  A primitive may fail because it may encounter
     a forwarder.  The primitive failure code needs to know to what depth it must follow
      arguments to follow forwarders and, if any are found and followed, retry the primitive.
     This method determines that depth. It starts by collecting references to the stack and
     then follows these through assignments to variables and use of accessor methods
     such as fetchPointer:ofObject:. For example
        | obj field  |
        obj := self stackTop.
        field := objectMemory fetchPointer: 1 ofObject: obj.
        self storePointer: 1 ofObject: field withValue: (self stackValue: 1)
    has depth 2, since field is accessed, and field is an element of obj."

    ^((self
            accessorChainsForMethod: method
            interpreterClass: (vmClass ifNil: [StackInterpreter]))
        inject: 0
        into: [:length :chain| length max: (self accessorDepthForChain: chain)]) - 1
```

Methods that don't access their arguments (for example, those that just answer results), end up with an accessor depth of -1.

## Contorno

With the depths extracted it's a simple step to store the depths in an array parallel to the primitive table. When a primitive fails the VM tests its corresponding accessor depth, and if non-negative, traverses the input parameters to the depth, following forwarding pointers, and then retrying the primitive:

```
*StackInterpreter methods for primitive support*
slowPrimitiveResponse
    "Invoke a normal (non-quick) primitive.
     Called under the assumption that primFunctionPointer has been preloaded."
    self initPrimCall.
    self perform: primitiveFunctionPointer.
    "In Spur a primitive may fail due to encountering a forwarder.
     On failure check the accessorDepth for the primitive and
     if non-negative scan the args to the depth, following any
     forwarders.  Retry the primitive if any are found."
    (objectMemory hasSpurMemoryManagerAPI
     and: [self successful not
     and: [(objectMemory isOopCompiledMethod: newMethod)
     and: [self checkForAndFollowForwardedPrimitiveState]]]) ifTrue:
        [self initPrimCall.
         self perform: primitiveFunctionPointer].
    ^self successful

checkForAndFollowForwardedPrimitiveState
    "In Spur a primitive may fail due to encountering a forwarder.
     On failure check the accessorDepth for the primitive and
     if non-negative scan the args to the depth, following any
     forwarders.  Answer if any are found so the prim can be retried."

    | primIndex accessorDepth found |
    self assert: self successful not.
    found := false.
    primIndex := self primitiveIndexOf: newMethod.
    accessorDepth := primitiveAccessorDepthTable at: primIndex.
    accessorDepth < 0 ifTrue:
        [^false].
    0 to: argumentCount do:
        [:index| | oop |
        oop := self stackValue: index.
        (objectMemory isNonImmediate: oop) ifTrue:
            [(objectMemory isForwarded: oop) ifTrue:
                [self assert: index < argumentCount. "receiver should have been caught at send time."
                 found := true.
                 oop := objectMemory followForwarded: oop.
                 self stackValue: index put: oop].
            ((objectMemory hasPointerFields: oop)
             and: [objectMemory followForwardedObjectFields: oop toDepth: accessorDepth]) ifTrue:
                [found := true]]].
    ^found
```

## Dolce

Ok, I lied. Well, not really; I only simplified. First, the Cursor primitive is simplified. The real version takes either 0 or 1 arguments, ignoring the mask if absent. Second, some primitives are complex enough to make calls to validation and parameter extraction routines, so that the parse tree analysis has to cope with more than one method and calls between them. Third, and this is desert, the system also has to store the accessor depth for plugin primitives, so-called "named primitives".

Computing the depth for named primitives is no different from their numbered bretheren, but storing the depth is problematic. Named primitives are listed in a per-plugin table that is searched the first time a primitive is invoked, and the information in the table, the primitive function to call and its accessor depth is copied into the method for subsequent use. Each method with a named primitive has a hidden first literal, an Array of the function name, the plugin name, the function pointer, and the accessor depth. Here's such a table:

```
void* FFTPlugin_exports[][3] = {
    {"FFTPlugin", "getModuleName", (void*)getModuleName},
    {"FFTPlugin", "primitiveFFTPermuteData", (void*)primitiveFFTPermuteData},
    {"FFTPlugin", "primitiveFFTScaleData", (void*)primitiveFFTScaleData},
    {"FFTPlugin", "primitiveFFTTransformData", (void*)primitiveFFTTransformData},
    {"FFTPlugin", "setInterpreter", (void*)setInterpreter},
    {NULL, NULL, NULL}
};
```

So far so trivial. (Why not one occurrence of the plugin name? Search me… Volunteers anyone?). It would be easy to extend the table, e.g. adding the depths as additional elements:

```
void* FFTPlugin_exports[][4] = {
    {"FFTPlugin", "getModuleName", (void*)getModuleName, -1},
    {"FFTPlugin", "primitiveFFTPermuteData", (void*)primitiveFFTPermuteData, 2},
    {"FFTPlugin", "primitiveFFTScaleData", (void*)primitiveFFTScaleData, 2},
    {"FFTPlugin", "primitiveFFTTransformData", (void*)primitiveFFTTransformData, 2},
    {"FFTPlugin", "setInterpreter", (void*)setInterpreter, -1},
    {NULL, NULL, NULL}
};
```

But since I'm parsimonious by nature and am building the Spur VM alongside the existing VM I like have only one copy of the primitives, in their own directory tree, and share them between the xisting and Spur VMs. I also want to modify the search code as little as possible. In particular I *don't* want to write

```
#if SPURVM
void* FFTPlugin_exports[][4] = {
    {"FFTPlugin", "getModuleName", (void*)getModuleName, -1},
    {"FFTPlugin", "primitiveFFTPermuteData", (void*)primitiveFFTPermuteData, 2},
    {"FFTPlugin", "primitiveFFTScaleData", (void*)primitiveFFTScaleData, 2},
    {"FFTPlugin", "primitiveFFTTransformData", (void*)primitiveFFTTransformData, 2},
    {"FFTPlugin", "setInterpreter", (void*)setInterpreter, -1},
    {NULL, NULL, NULL}
};
#else /* SPURVM */
void* FFTPlugin_exports[][3] = {
    {"FFTPlugin", "getModuleName", (void*)getModuleName},
    {"FFTPlugin", "primitiveFFTPermuteData", (void*)primitiveFFTPermuteData},
    {"FFTPlugin", "primitiveFFTScaleData", (void*)primitiveFFTScaleData},
    {"FFTPlugin", "primitiveFFTTransformData", (void*)primitiveFFTTransformData},
    {"FFTPlugin", "setInterpreter", (void*)setInterpreter},
    {NULL, NULL, NULL}
};
#endif /* SPURVM */
```

Instead I want to hide the depth in the existing table, without affecting the existing VM.

C gets a bad rap for being a low-level language unsuitable for implementing complex abstract systems. I agree. But it makes a decent portable assembler, and its low-levels are quite flexible. C's strings are null-terminated, but one can still embed nulls in literal strings using the \NNN notation for defining arbitrary bytes in octal. Hence Slang now outputs this as the actual table:

```
void* FFTPlugin_exports[][3] = {
    {"FFTPlugin", "getModuleName", (void*)getModuleName},
    {"FFTPlugin", "primitiveFFTPermuteData\000\002", (void*)primitiveFFTPermuteData},
    {"FFTPlugin", "primitiveFFTScaleData\000\002", (void*)primitiveFFTScaleData},
    {"FFTPlugin", "primitiveFFTTransformData\000\002", (void*)primitiveFFTTransformData},
    {"FFTPlugin", "setInterpreter", (void*)setInterpreter},
    {NULL, NULL, NULL}
};
```

When the standard VM searches the table it sees only the function name up until the embedded null, and is unaware of the extra byte lurking after it. But the Spur VM extracts the depth, and until there are primitives with accessor depths greater than 127 I think this scheme is safe ;-).

Posted by admin on Saturday, February 8th, 2014, at 6:57 pm, and filed under Cog, Spur.
Follow any responses to this entry with the RSS 2.0 feed.
You can post a comment, or trackback from your site.

{ 6 }
## Comments

1. **Benoit St-Jean** | 18-Apr-14 at 2:37 pm | Permalink

   Eliot,

   Have you ever thought about publishing a book on VM construction (a Smalltalk one of course!) ? Something one could read from page to page where you'd explain how to build a VM from scratch explaining the *how* and *why* of things?

2. **Eliot** | 18-Apr-14 at 4:44 pm | Permalink

   Hi Benoit,

   yes. In part that's why I chose to write the blog. My problem is that it's difficult to find the balance between addressing a large audience and getting too detailed. But at some stage I'd love to write a book. The "how" I have at my finger tips. The "why" is so interesting but more difficult to gather the evidence for. The answers vary from the evidence-based to the historical or merely expedient.

   But since you asked the question, what would you like to get from a VM book? Here are some suggestions:

   – a book that documents a specific VM so that you can extend it
   – a book that illustrates generic techniques with a set of exercises so that you can apply them
   – a book that surveys lots of VMs comparing and contrasting them
   – a book that explains the Smalltalk stack, from source to bytecode to interpreter, to machine code?

   Clément suggests "From Smalltalk to Native Code", to which I'd add "From Smalltalk to Native Code and Back" because in Sista we bounce back up to Smalltalk when the counters trip.

3. **Benoit St-Jean** | 19-Apr-14 at 7:52 pm | Permalink

   Ideally, it would focus on a Smalltalk VM and would point out difference between Smalltalk variants (e.g. there are surely tricks you can do in a Windows specific VM like Dolphin as compared to a generic one like Squeak). Obviously, you'd have to describe generic techniques and explain their rationale & purpose as well as how and why they work. It would be very interesting to have a book that would detail differences in implementation of the various Smalltalk dialects you have worked with (or know about). The main line : you describe *one* VM that you build from ground up and, along the way, you describe alternatives and differences with others. In the end, one have the tools to implement from scratch. I see this as a "Smalltalk VM Techniques" encyclopedia of some sort…

4. **Philippe Back** | 20-Apr-14 at 12:47 pm | Permalink

   Thanks for all of the detailed explanations.

   As for a book, one on the cog vm in as much detail as possible. What/why whatever.

   A section on Slang would be good.

   Cool trick in C BTW.

5. **Kirk W. Fraser** | 28-Dec-14 at 10:06 pm | Permalink

   Eliot,

   Running two Squeak images, preferably on separate cores, could use a socket based toggle handshake on garbage collection so one image would always be on point, not doing GC, so the free image can make time critical decisions. Can this feature be supported? Or perhaps a simple on/off to enable or disable garbage collection like Python has?

   Avoiding the 100ms garbage collection during critical moments is vital for self driving cars. A car going 60MPH with a 100ms delay in worse case could get 8 feet off track, into oncoming traffic or into a ditch. Thank you.

6. **Eliot Miranda** | 03-Jan-15 at 8:54 pm | Permalink

   Hi Kirk,

   where do you get 100ms pauses? Scavenges on current hardware (e.g. 2.2GHz Intel Core i7) are around 1ms. I see no problem to keeping an incremental scan-mark invocation to at most 5ms.

## Post a Comment

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

Name *

Email *

Website

Message

Post

« **LAZY BECOME AND A PARTIAL READ BARRIER FOR SPUR**

**WE ARE VERY HAPPY TO MAKE THE FOLLOWING ANNOUNCEMENT** »