{ 2008 07 24 }

Closures Part III – The Compiler

The Squeak bytecode compiler is a bijou jewel. It has flaws in some facets but on the whole it is small and beautiful. OK, its a Curate's Egg. Good in parts. Luckily for me Vassili Bykov has already written the perfect introduction to the compiler that will help you understand the rest of this post if you're unfamiliar with the compiler. So if you want to see how technical texts should properly be written please peruse Vassili's "The Hitch Hiker's Guide to the Smalltalk Compiler"

A Pluggable Compiler Back-End

I have two requirements for the compiler. One is that it compile closures using the bytecode design already presented. The other is that the compiler be bytecode-set agnostic. I want to be able to easily migrate to a new bytecode set with a different encoding. As we saw in the post on the bytecodes the bytecode set contains long forms and short forms. One simple way of migrating the bytecode set is to recompile the system using only the long forms, this means one can reassign the unused short forms as one sees fit. Its actually a five step dance:

1a recompile to the long form of the existing set
1b move to a VM that has the long form of the existing set and the long form of the new set
2a recompile to the long form of the new set
2b move to a VM that has the new bytecode set
3a recompile to the new bytecode set, short forms and all

Doing this easily means being able to plug different bytecode set encodings into the compiler easily. But if you look at the existing compiler the parse nodes have the instruction set embedded in them, which makes this very hard. For example here is the pair of methods that generate a store for either an instance or a temporary variable:

VariableNode methods for code generation

```
emitStorePop: stack on: strm
    (code between: 0 and: 7)
        ifTrue:
            [strm nextPut: ShortStoP + code "short stopop inst"]
        ifFalse:
            [(code between: 16 and: 23)
                ifTrue: [strm nextPut: ShortStoP + 8 + code – 16
"short stopop temp"]
                ifFalse: [(code >= 256 and: [code \\ 256 > 63 and:
[code // 256 = 4]])
                        ifTrue: [self emitLong: Store on: strm.
strm nextPut: Pop]
                        ifFalse: [self emitLong: StorePop on:
strm]]].
    stack pop: 1
```

This is difficult to understand, and is very hard to change. So the first thing to do is relace the back-end with something more readable and flexible. In the existing compiler class Encoder is responsble for encoding literals into literal indices and variable names into parse nodes. Let's extend it to encode the bytecode set as well. Since I need to support multiple bytecode sets we'll add a subclass for each set. We can use an abstract class to hold common code. Here's the hierarchy:

Encoder – the existing encoder, unchanged, still responsible for encoding literals and variables
BytecodeEncoder – the abstract superclass of the bytecode set encoders
EncoderForV3 – the encoder for the existing bytecode set, V3 because it is the bytecode set of Squeak3.x
EncoderForV3PlusClosures – the encoder that adds the closure opcodes
EncoderForLongFormV3 – the encoder that outputs only the long-forms of the V3 bytecodes
EncoderForLongFormV3PlusClosures – the encoder that adds closure opcodes to the long-form

BytecodeEncoder defines a stream instance variable which holds either a scratch buffer or the target method to generate code into. Very sharp of you to notice that I could have, arguably should have, made EncoderForV3 a subclass of EncoderForLongFormV3 because EncoderForV3 adds short-form encodings to the long forms and could inherit the code for outputting the long forms. But if I do that I can't easily eliminate the long-form encoder once it is no longer needed. So in this case I thought it better to keep the classes entirely separate.

BytecodeEncoder and the closure encoders can be queried by the rest of the compiler whether to use closure compilation or not:

BytecodeEncoder methods for testing

```
supportsClosureOpcodes
	"Answer if the receiver supports the
		genPushNewArray:/genPushConsArray:
		genPushRemoteTemp:inVectorAt:
		genStoreRemoteTemp:inVectorAt:
		genStorePopRemoteTemp:inVectorAt:
		genPushClosureCopyCopiedValues:numArgs:jumpSize:
	 opcodes"
	^false
```

EncoderForV3PlusClosures & EncoderForLongFormV3PlusClosures methods for testing

```
supportsClosureOpcodes
	^true
```

The bulk of interface between these bytecode encoders and the parse nodes is an api for sizing and emitting abstract opcodes. Each abstract opcode has a method defined in BytecodeEncoder for computing how many bytes the opcode will generate and a method defined in one of the concrete subclasses for emitting the bytecode encoding of the opcode that the concrete class determines. The compiler needs to be able to size opcodes before they're generated so as to be able to generate the right offsets for jump and block creation opcodes.

In the existing compiler each emit* method has a corresponding size* method that computes how many bytes the emit method will produce.

For example, here's the sizer for the emitter above

```
VariableNode methods for code generation
sizeForStorePop: encoder
	self reserve: encoder.
	(code < 24 and: [code noMask: 8]) ifTrue: [^ 1].
	code < 256 ifTrue: [^ 2].
	code \\ 256 <= 63 ifTrue: [^ 2]. "extended StorePop"
	code // 256 = 1 ifTrue: [^ 3]. "dbl extended StorePopInst"
	code // 256 = 4 ifTrue: [^ 4]. "dbl extended StoreLitVar , Pop"
	self halt. "Shouldn't get here"
```

This is error prone and ugly. The two methods are intimately connected. Change one and forget to change the other and the compiler breaks. There is nothing that guarantees that the sizes agree with the opcodes. Instead of having special versions of the sizing methods the new compiler simply outputs the bytecode to a scratch buffer and measures how much code was output:

```
BytecodeEncoder methods for opcode sizing
sizeOpcodeSelector: genSelector withArguments: args
	stream ifNil: [stream := WriteStream on: (ByteArray new: 8)].
	stream position: 0.
	self perform: genSelector withArguments: args.
	^stream position
```

So a sizing method simply looks like

```
BytecodeEncoder methods for opcode sizing
sizePushTemp: tempIndex
	^self sizeOpcodeSelector: #genPushTemp: withArguments:
{tempIndex}
```

The concrete subclasses then define genPushTemp: according to the encoding they choose, i.e.:

```
EncoderForV3 methods for bytecode generation
genPushTemp: tempIndex
	"See BlueBook page 596"
	tempIndex < 0 ifTrue:
		[^self outOfRangeError: 'index' index: tempIndex range: 0 to:
63].
	tempIndex < 16 ifTrue:
		["16-31		0001iiii	Push Temporary Location #iiii"
		 stream nextPut: 16 + tempIndex.
		 ^self].
	tempIndex < 64 ifTrue:
		["128		10000000 jjkkkkkk	Push (Receiver Variable,
Temporary Location, Literal Constant, Literal Variable) [jj] #kkkkkk"
		 stream
			nextPut: 128;
			nextPut: 64 + tempIndex.
		 ^self].
	^self outOfRangeError: 'index' index: tempIndex range: 0 to: 63
```

```
EncoderForLongFormV3 methods for bytecode generation
genPushTemp: tempIndex
	"See BlueBook page 596"
	tempIndex < 0 ifTrue:
		[^self outOfRangeError: 'index' index: tempIndex range: 0 to:
```

```
63].
	tempIndex < 64 ifTrue:
		["128	10000000 jjkkkkkk	Push (Receiver Variable, Temporary Location, Literal Constant, Literal Variable) [jj] #kkkkkk"
		stream
			nextPut: 128;
			nextPut: 64 + tempIndex.
		^self].
	^self outOfRangeError: 'index' index: tempIndex range: 0 to: 63
```

IMO, there is no need for a separate class method to introduce names for the opcode encodings, We can just use the constants directly. The code is clearer.

We can now list the full opcode set:

EncoderForV3 & EncoderForLongFormV3 methods for opcode generation
genBranchPopFalse: distance
genBranchPopTrue: distance
genDup
genJumpLong: distance
genJump: distance
genPop
genPushInstVarLong: instVarIndex
genPushInstVar: instVarIndex
genPushLiteralVar: literalIndex
genPushLiteral: literalIndex
genPushReceiver
genPushSpecialLiteral: aLiteral
genPushTemp: tempIndex
genPushThisContext
genReturnReceiver
genReturnSpecialLiteral: aLiteral
genReturnTop
genReturnTopToCaller
genSendSuper: selectorLiteralIndex **numArgs:** nArgs
genSend: selectorLiteralIndex **numArgs:** nArgs
genStoreInstVarLong: instVarIndex
genStoreInstVar: instVarIndex
genStoreLiteralVar: literalIndex
genStorePopInstVarLong: instVarIndex
genStorePopInstVar: instVarIndex
genStorePopLiteralVar: literalIndex
genStorePopTemp: tempIndex
genStoreTemp: tempIndex

EncoderForV3PlusClosures & EncoderForLongFormV3PlusClosures methods for opcode generation
genPushClosureCopyNumCopiedValues: numCopied **numArgs:** numArgs **jumpSize:** jumpSize
genPushConsArray: size
genPushNewArray: size
genPushRemoteTemp: tempIndex **inVectorAt:** tempVectorIndex
genStorePopRemoteTemp: tempIndex **inVectorAt:** tempVectorIndex
genStoreRemoteTemp: tempIndex **inVectorAt:** tempVectorIndex

The two forms of jump, variable sized and long, are used by the compiler to generate old-style blocks. By always using a long (two byte) jump to follow the block-creation #blockCopy: primitive the compiler allows blockCopy to work out the starting address for the block's code being two bytes after the following jump. e.g.:

```
19 <70> self
20 <89> pushThisContext:
21 <76> pushConstant: 1
22 <C8> send: blockCopy:
23 <A4 08> jumpTo: 33
25 <6B>        popIntoTemp: 3
26 <11>        pushTemp: 1
27 <12>        pushTemp: 2
28 <13>        pushTemp: 3
29 <F0>        send: value:value:
30 <81 42>        storeIntoTemp: 2
32 <7D>        blockReturn
33 <CB> send: do:
34 <87> pop
```

The long forms of the instance variable access opcodes are an evil hack of mine for Context instance variable access that I'll explain in a subsequent post on the Stack VM. For the moment forget they exist.

All these methods are very similar. Most are a few lines long. Here are one of the shortest and the longest:

EncoderForV3 & EncoderForLongFormV3 methods for opcode generation

```
genPop
	"See BlueBook page 596"
	"135	10000111	Pop Stack Top"
	stream nextPut: 135

genSend: selectorLiteralIndex numArgs: nArgs
	"See BlueBook page 596 (with exceptions for 132 & 134)"
	nArgs < 0 ifTrue:
		[^self outOfRangeError: 'numArgs' index: nArgs range: 0 to: 31 "!!"].
	selectorLiteralIndex < 0 ifTrue:
		["Special selector sends.
			176-191	1011iiii	Send Arithmetic Message #iiii
			192-207	1100iiii	Send Special Message #iiii"
		self flag: #yuck.
		 (selectorLiteralIndex negated between: 176 and: 207) ifFalse:
			[^self outOfRangeError: 'special selector code' index: selectorLiteralIndex negated range: 176 to: 207].
		 stream nextPut: selectorLiteralIndex negated.
		 ^self].
	(selectorLiteralIndex < 16 and: [nArgs < 3]) ifTrue:
		["	208-223	1101iiii	Send Literal Selector #iiii With No Arguments
			224-239	1110iiii	Send Literal Selector #iiii With 1 Argument
			240-255	1111iiii	Send Literal Selector #iiii With 2 Arguments"
		 stream nextPut: 208 + (nArgs * 16) + selectorLiteralIndex.
		 ^self].
```

```
    (selectorLiteralIndex < 32 and: [nArgs < 8]) ifTrue:
        ["    131     10000011 jjjkkkkk     Send Literal Selector
#kkkkk With jjj Arguments"
         stream
             nextPut: 131;
             nextPut: ((nArgs bitShift: 5) + selectorLiteralIndex).
         ^self].
    (selectorLiteralIndex < 64 and: [nArgs < 4]) ifTrue:
        ["In Squeak V3
             134     10000110 jjjjjjjj kkkkkkkk     Send Literal
Selector #kkkkkkkk To Superclass With jjjjjjjj Arguments
         is replaced by
             134     10000110 jjkkkkkk     Send Literal Selector
#kkkkkk With jj Arguments"
         stream
             nextPut: 134;
             nextPut: ((nArgs bitShift: 6) + selectorLiteralIndex).
         ^self].
    (selectorLiteralIndex < 256 and: [nArgs < 32]) ifTrue:
        ["In Squeak V3
             132     10000100 jjjjjjjj kkkkkkkk     Send Literal
Selector #kkkkkkkk With jjjjjjjj Arguments
         is replaced by
             132     10000100 ooojjjjj kkkkkkkk
                  ooo = 0 => Send Literal Selector #kkkkkkkk With
jjjjj Arguments
                  ooo = 1 => Send Literal Selector #kkkkkkkk To
Superclass With jjjjj Arguments"
        stream
             nextPut: 132;
             nextPut: nArgs;
             nextPut: selectorLiteralIndex.
         ^self].
    nArgs >= 32 ifTrue:
        [^self outOfRangeError: 'numArgs' index: nArgs range: 0 to:
31].
    selectorLiteralIndex >= 256 ifTrue:
        [^self outOfRangeError: 'selector literal index' index:
selectorLiteralIndex range: 0 to: 255]
```

Moving back to the Parse nodes they use the new API through a parallel set of sizers and emitters. By writing a parallel set I am able to a) bootstrap the compiler in place and b) check that the new back-end produces identical code to the old back-end. I generated the parallel set using a poor man's refactoring browser. I used the Shout syntax highlighter to do typed tokenisation of the old methods and produce identical copies that changed all uses of emit* and size*, replacing them with emitCode*, and sizeCode* so that for example emitForValue:on: maps to emitCodeForValue:encoder: and sizeForValue: maps to sizeCodeForValue:. Once I'd generated the duals I edited them by hand to mate them to the BytecodeEncoder API. For example here are old and new versions of send generation:

SelectorNode methods for code generation

```
emit: stack args: nArgs on: aStream super: supered
    | index |
    stack pop: nArgs.
    (supered not and: [code – Send < SendLimit and: [nArgs < 3]])
```

```
ifTrue:
            ["short send"
            code < Send
                  ifTrue: [^ aStream nextPut: code "special"]
                  ifFalse: [^ aStream nextPut: nArgs * 16 + code]].
      index := code < 256 ifTrue: [code – Send] ifFalse: [code \\ 256].
      (index <= 31 and: [nArgs <= 7]) ifTrue:
            ["extended (2-byte) send [131 and 133]"
            aStream nextPut: SendLong + (supered ifTrue: [2] ifFalse:
[0]).
            ^ aStream nextPut: nArgs * 32 + index].
      (supered not and: [index <= 63 and: [nArgs <= 3]]) ifTrue:
            ["new extended (2-byte) send [134]"
            aStream nextPut: SendLong2.
            ^ aStream nextPut: nArgs * 64 + index].
      "long (3-byte) send"
      aStream nextPut: DblExtDoAll.
      aStream nextPut: nArgs + (supered ifTrue: [32] ifFalse: [0]).
      aStream nextPut: index
```

```
size: encoder args: nArgs super: supered
      | index |
      self reserve: encoder.
      (supered not and: [code – Send < SendLimit and: [nArgs < 3]])
            ifTrue: [^1]. "short send"
      (supered and: [code < Send]) ifTrue:
            ["super special:"
            code := self code: (encoder sharableLitIndex: key) type: 5].
      index := code < 256 ifTrue: [code – Send] ifFalse: [code \\ 256].
      (index <= 31 and: [nArgs <= 7])
            ifTrue: [^ 2]. "medium send"
      (supered not and: [index <= 63 and: [nArgs <= 3]])
            ifTrue: [^ 2]. "new medium send"
      ^ 3 "long send"
```

SelectorNode methods for code generation (new scheme)

```
emitCode: stack args: nArgs encoder: encoder super: supered
      stack pop: nArgs.
      ^supered
            ifTrue:
                  [encoder genSendSuper: index numArgs: nArgs]
            ifFalse:
                  [encoder
                        genSend: (code < Send ifTrue: [code negated]
ifFalse: [index])
                        numArgs: nArgs]
```

```
sizeCode: encoder args: nArgs super: supered
      self reserve: encoder.
      ^supered
            ifTrue:
                  [code < Send "i.e. its a special selector" ifTrue:
                        [code := self code: (index := encoder
sharableLitIndex: key) type: 5].
                   encoder sizeSendSuper: index numArgs: nArgs]
            ifFalse:
                  [self flag: #yuck. "special selector sends cause this
problem"
                   encoder
```

```
                            sizeSend: (code < Send ifTrue: [code negated]
ifFalse: [index])
                            numArgs: nArgs]
```

While the new code still has a hack to deal with the special selector sends it is a lot clearer for having moved the details of bytecode encoding into the encoder. The new code for sends is more verbose but, I think, much more comprehensible.

Now that we can change the back-end we need to plug it into the front end. The existing parser hard codes the use of Encoder in Parser>>parse:class:noPattern:notifying:ifFail:

```
        …

         [methNode _ self method: noPattern context: ctxt encoder:
(Encoder new init: class context: ctxt notifying: self)]
on: ParserRemovedUnusedTemps
do:
[ :ex | repeatNeeded _ (requestor isKindOf: TextMorphEditor) not.
myStream _ ReadStream on: requestor text string.
ex resume].
repeatNeeded] whileTrue.

        …
```

The existing compiler holds onto an encoder instance, which makes sense. The encoder needs to be accessed throughout the front-end to map from names to nodes. With a bit of chicanery we can provide an interface that allows us to supply the class of encoder to use rather than supplying an already instantated encoder. This would be a mistake since encoder instantiation is done in the context of a target class to compile within as the encoder initializes its scope tables from the target class, and this would complicate things for the client.

Let's change Parser>>parse:class:noPattern:notifying:ifFail: to get the encoder from an accessor:

```
Parser methods for public access
parse: sourceStream class: class category: aCategory noPattern:
noPattern context: ctxt notifying: req ifFail: aBlock
        "Answer a MethodNode for the argument, sourceStream, that is
the root of
         a parse tree. Parsing is done with respect to the argument, class,
to find
         instance, class, and pool variables; and with respect to the
argument,
         ctxt, to find temporary variables. Errors in parsing are reported to
the
         argument, req, if not nil; otherwise aBlock is evaluated. The
argument
         noPattern is a Boolean that is true if the the sourceStream does
not
         contain a method header (i.e., for DoIts)."

        | methNode repeatNeeded myStream s p |
        category := aCategory.
        myStream := sourceStream.
        [repeatNeeded := false.
         p := myStream position.
         s := myStream upToEnd.
         myStream position: p.
         self init: myStream notifying: req failBlock: [^ aBlock value].
         doitFlag := noPattern.
         failBlock:= aBlock.
         [methNode := self
                                 method: noPattern
                                 context: ctxt
                                 encoder: (self encoder init: class context:
ctxt notifying: self)]
                on: ReparseAfterSourceEditing
                do:     [ :ex |
                        repeatNeeded := true.
                        myStream := ReadStream on: requestor text string].
         repeatNeeded] whileTrue:
                [encoder := self encoder class new].
        methNode sourceText: s.
        ^methNode
```

```
encoder
        encoder isNil ifTrue:
                [encoder := Encoder new].
        ^encoder
```

```
encoderClass: anEncoderClass
        encoder notNil ifTrue:
                [self error: 'encoder already set'].
        encoder := anEncoderClass new
```

and if we needed it we could implement

```
encoderClass
        ^self encoder class
```

So now we can experiment, saying things like

```
        | c |
        c := Compiler new.
        c parser encoderClass: EncoderForLongFormV3.
        c evaluate: '3 + 4' in: nil to: nil
```

or

```
        | 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]
```

Compiling To Closures

Now we have everything we need to compile to closures. The existing compiler has already made the change to ANSI-style block syntax, supporting block-local temporary declarations, even if it still implements these as method-scope temporaries. As a result we only need to modify the middle of the compiler, the ParseNode hierarchy. Once the parser has generated the parse node tree we need to traverse the tree analysing temporary variable usage. This is best done where methods are actually generated, in MethodNode>>generate:. However, since we have to do this modification in place without breaking the existing compiler let's be a bit circumspect and introduce a subclass of MethodNode that we'll use to insulate the existing compiler from our meddling:

```
MethodNode subclass: #BytecodeAgnosticMethodNode
	instanceVariableNames: 'locationCounter localsPool'
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Compiler-ParseNodes'
```

Encoder methods for accessing
methodNodeClass
```
	^MethodNode
```

BytecodeEncoder methods for accessing
methodNodeClass
```
	^BytecodeAgnosticMethodNode
```

Parser methods for expression types
newMethodNode
```
	^self encoder methodNodeClass new
```

Now BytecodeAgnosticMethodNode (a bit of a mouthful) can have its own generate: method:

BytecodeAgnosticMethodNode methods for code generation (new scheme)
generate: trailer
```
	"The receiver is the root of a parse tree. Answer a
CompiledMethod.
	 The argument, trailer, is the reference to the source code that is
	 stored with every CompiledMethod."

	| blkSize nLits literals stack method |
	self generate: trailer ifQuick:
				[:m |
				 m	literalAt: 2 put: encoder associationForClass;
					properties: properties.
				^m].
	encoder supportsClosureOpcodes ifTrue:
			[temporaries := block analyseArguments: arguments
temporaries: temporaries rootNode: self.
			 encoder rootNode: self. "this is for
BlockNode>>sizeCodeForClosureValue:"].
	blkSize := block sizeCodeForEvaluatedValue: encoder.
	method := CompiledMethod
					newBytes: blkSize
					trailerBytes: trailer
					nArgs: arguments size
					nTemps: (encoder supportsClosureOpcodes
							ifTrue: [| locals |
									locals := arguments,
temporaries.
									encoder

	noteBlockExtent: block blockExtent
```

```
                                                              hasLocals:
locals.
                                                           locals size]
                                                    ifFalse: [encoder maxTemp])
                                   nStack: 0
                                   nLits: (nLits := (literals := encoder allLiterals)
size)
                                   primitive: primitive.
        nLits > 255 ifTrue:
                [^self error: 'Too many literals referenced'].
        1 to: nLits do: [:lit | method literalAt: lit put: (literals at: lit)].
        encoder streamToMethod: method.
        stack := ParseStack new init.
        block emitCodeForEvaluatedValue: stack encoder: encoder.
        stack position ~= 1 ifTrue:
                [^self error: 'Compiler stack discrepancy'].
        encoder methodStreamPosition ~= (method size – trailer size)
ifTrue:
                [^self error: 'Compiler code size discrepancy'].
        method needsFrameSize: stack size.
        method properties: properties.
        ^method
```

The analysis pass is performed by

```
        encoder supportsClosureOpcodes ifTrue:
                [temporaries := block analyseArguments: arguments
temporaries: temporaries rootNode: self.
                 encoder rootNode: self. "this is for
BlockNode>>sizeCodeForClosureValue:"].
```

and the method uses the new sizers and emitters via sizeCodeForEvaluatedValue: and emitCodeForEvaluatedValue:.

The Closure Analysis

We're finally at an interesting bit, namely how to analyse the parse tree to handle temporaries correctly when using closures. We need to know for each temporary whether it is closed over (used in a block other than – necessarily within – its block scope), and if it is assigned to after being closed-over. If a temporary is not assigned to after being closed over the closures can be given read-only copies of the temporary. If a temporary is written to after being closed over then it needs to be made remote, i.e. not live as a temporary in the declaring block but live in an Array which, being read-only, *can* be copied. Remember the rationale for all this is to break the dependence of closing over scopes on the stack frames of declaring scopes. The Closures Part I post explains this in more detail.

It turns out this analysis is quite simple to do. We simply assign different numbers to different block scopes. We can initialize temporary nodes with their defining scope's number and then compare this against the scope number at each access point. Here's the numbering applied to our old saw Collection>>inject:into:. The method's scope number starts at 0. After a scope's temporaries are declared the number is incremented. When a block is entered or exited the block count is incremented.

```
        Collection methods for enumerating
0:          inject: thisValue into: binaryBlock
```

```
            | nextValue |
1:              nextValue := thisValue.
            self do:
2:                      [:each |
3:                       nextValue := binaryBlock value: nextValue value: each
4:                      ].
5:      ^nextValue
6:
```

The scope extent of the whole method is 0 to 6. The scope of the block is 2 to: 4, which we'll call the block extent, and the scope within it is 3 to: 3.

So now let's label each argument and temporary with its uses, d@N for definitions, r@N for reads, w@N for writes

```
    inject: thisValue d@0 into: binaryBlock d@0
            | nextValue d@0 |
            nextValue w@1 := thisValue r@1.
            self do:
                  [:each d@2 |
                   nextValue w@3 := binaryBlock r@3 value: nextValue r@3 value: each r@3].
            ^nextValue r@5
```

Now let's analyse:

thisValue	d@0 r@1
binaryBlock	d@0 r@3
nextValue	d@0 r@3 r@5 w@1 w@3
each	d@2 r@3

So the block [2 to: 4] can determine that it closes over binaryBlock and nextValue because these are defined before the start of its block extent and referenced within it. nextValue can determine that it must be remote because it is written to in an inner scope. It therefore asks its defining scope block [0 to: 6] to add a remote temporary, represented by an instance of RemoteTempVectorNode, adds itself to the node's sequence of remote temps, and notes that it is remote by setting its remoteNode instance varable to the new remote temp vector node.

Since there only ever needs to be one remote temp vector node per scope we can use the block extent as its name. After we have introduced the remote temp vector node our method numbering looks like this. nextValue now lives inside the remote temp vector <0-6> and not on the stack of method block [0 to: 6].

```
    inject: thisValue d@0 into: binaryBlock d@0
            | <0-6: nextValue> d@0 |
            <0-6> r@1 at: 1 put: thisValue r@1.
            self do:
                  [:each d@2 |
                   <0-6> r@3 at: 1 put: (binaryBlock r@3 value: (<0-6> r@3 at: 1) value: each r@3)].
            ^<0-6> r@5 at: 1
```

and the analysis becomes

thisValue	d@0 r@1
binaryBlock	d@0 r@3
<0-6>	d@0 r@3 r@5

```
	each				d@2 r@3
```

We have eliminated the writes. So now block [2 to: 4] can determine it needs to copy binaryBlock and <0-6>. Simple! When we later perform the sizing and emitting passes to generate code nextValue knows it is remote and defers to its remoteNode <0-6> to generate code using the relevant remote temp opcodes. Also simple. Let’s look at the code. Recall that generate: above invokes the analysis pass that starts here. Note that an inlined block has its optimized flag set to true.

BlockNode methods for code generation (closures)

```
analyseArguments: methodArguments temporaries: methodTemporaries rootNode: rootNode
	arguments := methodArguments asArray. "won’t change"
	self assert: (temporaries isNil or: [temporaries isEmpty or: [temporaries hasEqualElements: methodTemporaries]]).
	temporaries := OrderedCollection withAll: methodTemporaries.

	self assert: optimized not. "the top-level block should not be optimized."
	self analyseTempsWithin: self rootNode: rootNode.

	"The top-level block needs to reindex temporaries since analysis may have rearranged them.
	 This happens during sizing for nested blocks."
	temporaries withIndexDo:
		[:temp :offsetPlusOne| temp index: arguments size + offsetPlusOne – 1].

	"Answer the (possibly modified) sequence of temps."
	^temporaries asArray

analyseTempsWithin: scopeBlock "<BlockNode>" rootNode: rootNode "<MethodNode>"
	| blockStart |
	optimized ifTrue:
		[self assert: (temporaries isEmpty and: [arguments isNil or: [arguments size <= 1]]).
		 statements do:
			[:statement|
			 statement analyseTempsWithin: scopeBlock rootNode: rootNode].
		 ^self].

	rootNode noteBlockEntry:
		[:entryNumber|
		 blockStart := entryNumber.
		 arguments notNil ifTrue: [arguments do: [:temp| temp definingScope: self]].
		 temporaries notNil ifTrue: [temporaries do: [:temp| temp definingScope: self]]].

	statements do:
		[:statement|
		 statement analyseTempsWithin: self rootNode: rootNode].

	rootNode noteBlockExit:
		[:exitNumber|
		 blockExtent := blockStart to: exitNumber].

	self postNumberingProcessTemps: rootNode
```

The root method node is responsible for providing the scope count:

BytecodeAgnosticMethodNode methods for code generation (closures)

```
noteBlockEntry: aBlock
    "Evaluate aBlock with the numbering for the block entry."
    locationCounter isNil ifTrue:
        [locationCounter := -1].
    aBlock value: locationCounter + 1.
    locationCounter := locationCounter + 2


noteBlockExit: aBlock
    "Evaluate aBlock with the numbering for the block exit."
    aBlock value: locationCounter + 1.
    locationCounter := locationCounter + 2
```

and yes, these are effectively the same but I didn’t realise that when I started 🙂

The only non-trivial implementions of analyseTempsWithin:rootNode: are in AssignmentNode and TempVariableNode.

AssignmentNode methods for code generation (closures)

```
analyseTempsWithin: scopeBlock "<BlockNode>" rootNode:
rootNode "<MethodNode>"
    "N.B. since assigment happens _after_ the value is evaluated the
value is sent the message _first_."
    value analyseTempsWithin: scopeBlock rootNode: rootNode.
    variable beingAssignedToAnalyseTempsWithin: scopeBlock
rootNode: rootNode
```

TempVariableNode methods for code generation (closures)

```
analyseTempsWithin: scopeBlock "<BlockNode>" rootNode:
rootNode "<MethodNode>"
    self addReadWithin: scopeBlock at: rootNode locationCounter

beingAssignedToAnalyseTempsWithin: scopeBlock "<BlockNode>"
rootNode: rootNode "<MethodNode>"
    self addWriteWithin: scopeBlock at: rootNode locationCounter

addReadWithin: scopeBlock "<BlockNode>" at: location "<Integer>"
    readingScopes ifNil: [readingScopes := Dictionary new].
    (readingScopes at: scopeBlock ifAbsentPut: [Set new]) add:
location

addWriteWithin: scopeBlock "<BlockNode>" at: location "<Integer>"
    writingScopes ifNil: [writingScopes := Dictionary new].
    (writingScopes at: scopeBlock ifAbsentPut: [Set new]) add:
location
```

The action starts in postNumberingProcessTemps:.

BlockNode methods for code generation (closures)

```
postNumberingProcessTemps: rootNode "<MethodNode>"
    (temporaries isNil or: [temporaries isEmpty]) ifTrue:
        ["Add all arguments to the pool so that copiedValues can be
computed during sizing."
         rootNode addLocalsToPool: arguments.
         ^self].

    "A temp can be local (and copied if it is not written to after it is
captured.
```

```
         A temp cannot be local if it is written to remotely.
         Need to enumerate a copy of the temporaries because any temps
becoming remote
         will be removed from temporaries in analyseClosure: (and a
single remote temp node
         will get added)"
        temporaries copy do:
            [:each| each analyseClosure: rootNode].

        "Now we may have added a remoteTempNode. So we need a
statement to initialize it."
        remoteTempNode ~~ nil ifTrue:
            ["statements isArray ifTrue:
                [statements := statements asOrderedCollection]." "true
for decompiled trees"
            (statements notEmpty
             and: [statements first isAssignmentNode
             and: [statements first variable isTemp
             and: [statements first variable isIndirectTempVector]]])
                ifTrue: "If this is a decompiled tree there already is a
temp vector initialization node."
                    [statements first variable become:
remoteTempNode]
                ifFalse:
                    [statements addFirst: (remoteTempNode
nodeToInitialize: rootNode encoder)]].

        "Now add all arguments and locals to the pool so that
copiedValues can be computed during sizing."
        rootNode
            addLocalsToPool: arguments;
            addLocalsToPool: temporaries
```

So if the analysis in analyseClosure: creates a remoteTempNode we'll add a statement to initialize it at the start of the block via nodeToInitialize:. The explicit Array code dates from early in the project before I had defined the bytecodes. Its harmless, and we can still generate the old pre-closure bytecode closure code if we want to.

RemoteTempVectorNode methods for code generation (closures)

```
nodeToInitialize: encoder
        ^AssignmentNode new
            variable: self
            value: (encoder supportsClosureOpcodes
                            ifTrue: [NewArrayNode new numElements:
remoteTemps size]
                            ifFalse:
                                [MessageNode new
                                    receiver: (encoder
encodeVariable: 'Array')
                                    selector: (encoder
encodeSelector: #new:)
                                    arguments: (Array with:
(encoder encodeLiteral: remoteTemps size))
                                    precedence: 3])
```

The analysis proper is done by TempVariableNode in analyseClosure:. Looking at addReadWithin: and addWriteWithin: you'll see that both readingScopes and writingScopes are Dictionaries from BlockNode to Set of scope count.

TempVariableNode methods for code generation (closures)

```
analyseClosure: rootNode "<MethodNode>"
	"A temp cannot be local if it is written to remotely,
	 or if it is written to after it is closed-over."
	| latestWrite |
	latestWrite := 0.
	((writingScopes notNil
	 and: [writingScopes associations anySatisfy: [:assoc|
			[:blockScope :refs|
			refs do: [:write| latestWrite := write max: latestWrite].
			"A temp cannot be local if it is written to remotely."
			blockScope ~~ definingScope]
				value: assoc key value: assoc value]])
	or: [readingScopes notNil
		and: [readingScopes associations anySatisfy: [:assoc|
				[:blockScope :refs|
				 "A temp cannot be local if it is written to after it is
closed-over."
				 blockScope ~~ definingScope
				 and: [refs anySatisfy: [:read| read < latestWrite]]]
					value: assoc key value: assoc value]]])
ifTrue:
		[remoteNode := definingScope addRemoteTemp: self
rootNode: rootNode]
```

If the temporary is written to remotely, as nextBlock is in inject:into: then it must be remoted. But there is another case, of a temporary that is closed over before it is written to. So the pass over writingScopes computes the latest scope count at which the temporary is written to in other than in its defining scope. The pass over the readingScopes can then determine if the temporary needs to be made remote by detecting an out of scope read before any write. An example of this might be

```
	plusOneBlockFrom: anInteger
		| count block |
		block := [count + 1].
		count := anInteger.
		^block
```

The only thing remaining is for BlockNodes to notice which temporaries they copy when sizing and emitting their creation code. let’s look at the sizers which is where the copied values creation code gets generated:

BlockNode methods for testing

```
generateAsClosure
	"Answer if we’re compiling under the closure regime. If blockExtent
has been set by
	analyseTempsWithin:rootNode: et al then we’re compiling under
the closure regime."
	^blockExtent ~~ nil
```

BlockNode methods for code generation (new scheme)

```
sizeCodeForValue: encoder
	self generateAsClosure ifTrue:
		[^self sizeCodeForClosureValue: encoder].

	… old code elided …
```

With the closure bytecodes the BlockNode merely needs to note its copied values, which it does in an instace varable copiedValues. Without the bytecodes things are more involved. Focus on the code path with the closure bytecodes. I'll grey-out the other code path.

```
BlockNode methods for code generation (closures)
sizeCodeForClosureValue: encoder
	"Compute the size for the creation of the block and its code."
	"If we have the closure bytecodes constructClosureCreationNode: will note
	 the copied values in the copiedValues inst var and answer #pushCopiedValues."
	closureCreationSupportNode := self constructClosureCreationNode: encoder.
	"Remember size of body for emit time so we know the size of the jump around it."
	size := self sizeCodeForEvaluatedClosureValue: encoder.
	^encoder supportsClosureOpcodes
		ifTrue:
			[(copiedValues inject: 0 into: [:sum :node| sum + (node sizeCodeForValue: encoder)])
			 + (encoder sizePushClosureCopyNumCopiedValues: copiedValues size numArgs: arguments size jumpSize: size)
			 + size]
		ifFalse:
			["closureCreationSupportNode is	send closureCopy:copiedValues:"
			(closureCreationSupportNode sizeCodeForValue: encoder)
			 + (encoder sizeJumpLong: size)
			 + size]

constructClosureCreationNode: encoder
	copiedValues := self computeCopiedValues: encoder rootNode.
	encoder supportsClosureOpcodes ifTrue:
		[^#pushCopiedValues].
	"Without the bytecode we can still get by."
	^MessageNode new
		receiver: (encoder encodeVariable: 'thisContext')
		selector: #closureCopy:copiedValues:
		arguments: (Array
					with: (encoder encodeLiteral: arguments size)
					with: (copiedValues isEmpty
							ifTrue: [NodeNil]
							ifFalse: [BraceNode new elements: copiedValues]))
		precedence: 3
		from: encoder
```

It is trivial to collect the relevant copied values once we have numbered each temporary:

```
BlockNode methods for code generation (closures)
computeCopiedValues: rootNode
	| referencedValues |
	referencedValues := rootNode referencedValuesWithinBlockExtent: blockExtent.
	^((referencedValues reject:
```

```
            [:temp| temp isDefinedWithinBlockExtent: blockExtent])
asSortedCollection:
                [:t1 :t2 | t1 index < t2 index]) asArray
```

To create the closure we need to push any copied values, output the closure creation bytecode and output the body of the block. The bytecode takes care opf packaging up aby copied values in an Array and putting them in the closure.

```
BlockNode methods for code generation (closures)
emitCodeForClosureValue: stack encoder: encoder
    "if not supportsClosureOpcodes closureCreationSupportNode is
the
     node for thisContext closureCopy: numArgs [ copiedValues: {
values } ]"
    encoder supportsClosureOpcodes
        ifTrue:
            [copiedValues do:
                [:copiedValue| copiedValue emitCodeForValue:
stack encoder: encoder].
             encoder
                genPushClosureCopyNumCopiedValues:
copiedValues size
                numArgs: arguments size
                jumpSize: size.
             stack
                pop: copiedValues size;
                push: 1]
        ifFalse:
            [closureCreationSupportNode emitCodeForValue:
stack encoder: encoder.
             encoder genJumpLong: size]. "Force a two byte jump."
    "Emit the body of the block"
    self emitCodeForEvaluatedClosureValue: stack encoder: encoder
```

So far so beautiful. There is a wrinkle (um, hack) in this story but I plead TSTTCPW. The issue arises from the fact that we have not introduced different scopes for each block and so have shared one set of temporaries amongst all blocks in a method. This has simplified the analysis above because we haven’t had to deal with a given temporary being represented by different nodes in different scopes (c.f. the VisualWorks compiler). But it means that the indices for the stack locations of temporaries are potentially different in different scopes. So we must renumber temporaries with their correct indices when generating the code for each block, and restore them to their previous values once we’re done. This is managed by the following method which also copies the current bockExtent into the encodeer so that temporary nodes know the current scope when referenced.

```
BlockNode methods for code generation (closures)
reindexingLocalsDo: aBlock encoder: encoderOrNil
    "Evaluate aBlock wih arguments, temporaries and copiedValues
reindexed for
     their positions within the receiver’s block, restoring the correct
indices afterwards.
     If encoder is not nil remember the temps for this block’s extent."
    | tempIndices result tempsToReindex |
    self assert: copiedValues notNil.
    tempsToReindex := arguments asArray, copiedValues,
temporaries.
	tempIndices := tempsToReindex collect: [:temp| temp index].
	tempsToReindex withIndexDo:
		[:temp :newIndex| temp index: newIndex – 1. self assert:
temp index + 1 = newIndex].
	encoderOrNil ifNotNil:
		[encoderOrNil noteBlockExtent: blockExtent hasLocals:
tempsToReindex].
	result := aBlock ensure:
				["Horribly pragmatic hack. The copiedValues will
have completely
				 unrelated indices within the closure method and
sub-method.
				 Avoiding the effort of rebinding temps in the inner
scope simply
				 update the indices to their correct ones during
the generation of
				 the closure method and restore the indices
immedately there-after."
				 tempsToReindex with: tempIndices do:
					[:temp :oldIndex| temp index: oldIndex. self
assert: temp index = oldIndex]].
	^result
```

Again by TSTTCPW the initialization of local temps (not arguments or copiedValues) is done by explicit pushes of nil.

BlockNode methods for code generation (closures)
emitCodeForEvaluatedClosureValue: stack **encoder:** encoder

```
	temporaries size timesRepeat:
		[NodeNil emitCodeForValue: stack encoder: encoder].
	self
		reindexingLocalsDo: [self emitCodeForEvaluatedValue:
stack encoder: encoder]
		encoder: encoder.
	self returns ifFalse:
		[encoder genReturnTopToCaller.
		 pc := encoder methodStreamPosition].
	stack pop: 1 "for block return" + temporaries size
```

OK, we’re nearly done. So let’s look at the generated bytecode for Collection>>inject:into: in all of our six code models. I have a testing method generateUsingClosures: that will do the closure analysis and generate closure code without using the closure bytecodes. The last one is the code one gets in the system once the switch to closure compilation is thrown by redefining Parser>>encoder:

Parser methods for public access
encoder

```
	encoder isNil ifTrue:
		[encoder := EncoderForV3PlusClosures new].
	^encoder
```

Here’s the code to generate all six versions, the existing compiler’s code using the long-form bytecodes, the existing compiler’s code, closures using only the long forms of the old bytecodes, closures using only the old bytecodes, closures using the long forms of the closure bytecodes and finally the closure bytecodes proper.

```
String streamContents:
	[:stream|
	{ Encoder. EncoderForLongFormV3.
	 EncoderForLongFormV3. EncoderForV3.
	 EncoderForLongFormV3PlusClosures.
EncoderForV3PlusClosures }
			with: #(	generate: generate:
					generateUsingClosures:
generateUsingClosures:
					generate: generate:)
			do:
				[:encoderClass :generator| | method |
				method := ((Parser new
							encoderClass:
encoderClass;
							parse: (Collection
sourceCodeAt: #inject:into:)
							class: Object)
						perform: generator with: #
(0 0 0 0)).
				stream print: encoderClass; cr; nextPutAll:
method symbolic; cr; cr]]
```

which duly prints

```
Encoder
17 <10> pushTemp: 0
18 <6A> popIntoTemp: 2
19 <70> self
20 <89> pushThisContext:
21 <76> pushConstant: 1
22 <C8> send: blockCopy:
23 <A4 08> jumpTo: 33
25 <6B> popIntoTemp: 3
26 <11> pushTemp: 1
27 <12> pushTemp: 2
28 <13> pushTemp: 3
29 <F0> send: value:value:
30 <81 42> storeIntoTemp: 2
32 <7D> blockReturn
33 <CB> send: do:
34 <87> pop
35 <12> pushTemp: 2
36 <7C> returnTop

EncoderForLongFormV3
25 <80 40> pushTemp: 0
27 <82 42> popIntoTemp: 2
29 <70> self
30 <89> pushThisContext:
31 <76> pushConstant: 1
32 <83 22> send: blockCopy:
34 <A4 0D> jumpTo: 49
36 <82 43> popIntoTemp: 3
38 <80 41> pushTemp: 1
40 <80 42> pushTemp: 2
42 <80 43> pushTemp: 3
44 <83 41> send: value:value:
46 <81 42> storeIntoTemp: 2
```

```
48 <7D> blockReturn
49 <83 20> send: do:
51 <87> pop
52 <80 42> pushTemp: 2
54 <7C> returnTop
```

EncoderForLongFormV3

```
45 <80 C1> pushLit: Array
47 <76> pushConstant: 1
48 <83 20> send: new:
50 <82 42> popIntoTemp: 2
52 <80 42> pushTemp: 2
54 <76> pushConstant: 1
55 <80 40> pushTemp: 0
57 <83 42> send: at:put:
59 <87> pop
60 <70> self
61 <89> pushThisContext:
62 <76> pushConstant: 1
63 <80 C1> pushLit: Array
65 <80 41> pushTemp: 1
67 <80 42> pushTemp: 2
69 <83 47> send: braceWith:with:
71 <83 46> send: closureCopy:copiedValues:
73 <A4 11> jumpTo: 92
75 <80 42> pushTemp: 2
77 <76> pushConstant: 1
78 <80 41> pushTemp: 1
80 <80 42> pushTemp: 2
82 <76> pushConstant: 1
83 <83 25> send: at:
85 <80 40> pushTemp: 0
87 <83 44> send: value:value:
89 <83 42> send: at:put:
91 <7D> blockReturn
92 <83 23> send: do:
94 <87> pop
95 <80 42> pushTemp: 2
97 <76> pushConstant: 1
98 <83 25> send: at:
100 <7C> returnTop
```

EncoderForV3

```
29 <40> pushLit: Array
30 <76> pushConstant: 1
31 <CD> send: new:
32 <6A> popIntoTemp: 2
33 <12> pushTemp: 2
34 <76> pushConstant: 1
35 <10> pushTemp: 0
36 <C1> send: at:put:
37 <87> pop
38 <70> self
39 <89> pushThisContext:
40 <76> pushConstant: 1
41 <40> pushLit: Array
42 <11> pushTemp: 1
43 <12> pushTemp: 2
```

```
44 <F3> send: braceWith:with:
45 <F2> send: closureCopy:copiedValues:
46 <A4 0A> jumpTo: 58
48 <12> pushTemp: 2
49 <76> pushConstant: 1
50 <11> pushTemp: 1
51 <12> pushTemp: 2
52 <76> pushConstant: 1
53 <C0> send: at:
54 <10> pushTemp: 0
55 <F1> send: value:value:
56 <C1> send: at:put:
57 <7D> blockReturn
58 <CB> send: do:
59 <87> pop
60 <12> pushTemp: 2
61 <76> pushConstant: 1
62 <C0> send: at:
63 <7C> returnTop
```

EncoderForLongFormV3PlusClosures

```
21 <8A 01> push: (Array new: 1)
23 <82 42> popIntoTemp: 2
25 <80 40> pushTemp: 0
27 <8E 00 02> popIntoTemp: 0 inVectorAt: 2
30 <70> self
31 <80 41> pushTemp: 1
33 <80 42> pushTemp: 2
35 <8F 21 00 0D> closureNumCopied: 2 numArgs: 1 bytes 39 to
51
39 <80 41> pushTemp: 1
41 <8C 00 02> pushTemp: 0 inVectorAt: 2
44 <80 40> pushTemp: 0
46 <83 41> send: value:value:
48 <8D 00 02> storeIntoTemp: 0 inVectorAt: 2
51 <7D> blockReturn
52 <83 20> send: do:
54 <87> pop
55 <8C 00 02> pushTemp: 0 inVectorAt: 2
58 <7C> returnTop
```

EncoderForV3PlusClosures

```
17 <8A 01> push: (Array new: 1)
19 <6A> popIntoTemp: 2
20 <10> pushTemp: 0
21 <8E 00 02> popIntoTemp: 0 inVectorAt: 2
24 <70> self
25 <11> pushTemp: 1
26 <12> pushTemp: 2
27 <8F 21 00 0A> closureNumCopied: 2 numArgs: 1 bytes 31 to
40
31 <11> pushTemp: 1
32 <8C 00 02> pushTemp: 0 inVectorAt: 2
35 <10> pushTemp: 0
36 <F0> send: value:value:
37 <8D 00 02> storeIntoTemp: 0 inVectorAt: 2
40 <7D> blockReturn
41 <CB> send: do:
```

```
42 <87> pop
43 <8C 00 02> pushTemp: 0 inVectorAt: 2
46 <7C> returnTop
```

The next post will describe how we cope with all this chaos in the debugger which needs to be independent of the exact layout of CompiledMethods if we're to make sense of the above while developing these different versions. It will also do some analysis of the system after recompilation to closures.

Posted by admin on Thursday, July 24th, 2008, at 12:05 am, and filed under Cog.
Follow any responses to this entry with the RSS 2.0 feed.
You can post a comment.

{ 1 }

Comments

1. **tim@rowledge.org** | 24-Jul-08 at 2:09 pm | Permalink

   "Its actually a five step dance:

   1a recompile to the long form of the existing set
   1b move to a VM that has the long form of the existing set and the long form of the new set
   2a recompile to the long form of the new set
   2b move to a VM that has the new bytecode set
   3a recompile to the new bytecode set, short forms and all"

   Hunh? SystemTracer, dear boy.
   1. write new compiler
   2. Add systemtracer code to recompile compiledmethods using new compiler and set appropriate state in new image
   3. run new image on new vm
   With added luxury approach of 3a – run new image in simulator.

   The cloner can substitute classes during the clone; for example you can have two classes in the image and set up the cloning so that the 'new' class replaces all mentions of the old class as well as removing all now-redundant pointers and instances. I developed and used this in '98 to clone images with the cleaner compiled method format (the older cloner code couldn't cope) for Interval. It has the advantage of fewer interlocked steps to replicate when you discover a bug that means restarting. DAMHIKT.

Post a Comment

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

Name		*
Email		*

Website

Message

Post

« CLOSURES PART II – THE BYTECODES

MECHANISED MODIFICATIONS AND MISCELLANEOUS MEASUREMENTS »