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 11 14 }

Mechanised Modifications and Miscellaneous Measurements

You read the last post and I can tell you're unhappy with the size of the remote temp bytecodes. I can hear you grumbling that it has got to hurt performance. "But these bytecodes are really rare in practice" I claim. "Prove it", you say. "I took a look and there are loads of the damn things", you say, and of course you're right.

Let's measure how often these crop up in practice. I'm running these doits in a freshly bootstrapped Croquet 1.0.18 image. You can evaluate these in your own bootstrap image if you're so inclined.

```
CompiledMethod instanceCount 67650
```

OK, so we have quite a few methods. Since the compiler arranges that the first bytecode will be a pushNewArray opcode if there are remote temps in a method we can scan for these quite quickly. Note that this approach won't find temporaries declared inside a block that are only accessed within nested blocks, but it is good enough. We can save complicatons by filtering out quick methods which return self, an instance variable nil, true or false as these don't have bytecodes. We also need to filter out the use of pushNewArray for creating tuples. These have the sign bit set on the following byte, which is the number of elements in the array.

```
(SystemNavigation default allSelect:
    [:m|
    m isQuick not
    and: [(m at: m initialPC) = 138
    and: [(m at: m initialPC + 1) <= 127]]]) size 2428
```

OK, so fully 2428 / 67650 asFloat = 0.03589061345158906, or 3.6% of methods contain remote temps. Argh! What's going on?

Well remember that the bootstrap has just been run in a system that up until now used BlueBook blocks where all temporaries are at method level and are accessed indirectly from within blocks. So up until now there has been little incentive to declare block-local temps at block-level scope. They end up being declared (and decompiled and pretty-printed) at method level anyway. Of course there is a case where a method-level temporary is purposefully used only at block level, when it is read before written. These shouldn't be changed. Take for example the following method; I'm using ‡ to draw your attention to relevant detail.

```
Collection methods for enumerating
detectMax: aBlock
    "Evaluate aBlock with each of the receiver's elements as the argument.
    Answer the element for which aBlock evaluates to the highest magnitude.
    If collection empty, return nil. This method might also be called elect:."

‡    | maxElement maxValue val |
    self do: [:each |
        maxValue == nil
            ifFalse: [
                (val := aBlock value: each) > maxValue ifTrue: [
                    maxElement := each.
                    maxValue := val]]
            ifTrue: ["first element"
                maxElement := each.
                maxValue := aBlock value: each].
                "Note that there is no way to get the first element that works
                for all kinds of Collections. Must test every one."].
    ^ maxElement
```

We can change this by moving val to block-level scope to

```
detectMax: aBlock
    "Evaluate aBlock with each of the receiver's elements as the argument.
    Answer the element for which aBlock evaluates to the highest magnitude.
    If collection empty, return nil. This method might also be called elect:."

    | maxElement maxValue |
‡    self do: [:each | | val |
        maxValue == nil
            ifFalse: [
                (val := aBlock value: each) > maxValue ifTrue: [
                    maxElement := each.
                    maxValue := val]]
            ifTrue: ["first element"
                maxElement := each.
                maxValue := aBlock value: each].
                "Note that there is no way to get the first element that works
                for all kinds of Collections. Must test every one."].
    ^ maxElement
```

but we would break it if we changed it to

```
detectMax: aBlock
    "Evaluate aBlock with each of the receiver's elements as the argument.
    Answer the element for which aBlock evaluates to the highest magnitude.
    If collection empty, return nil. This method might also be called elect:."

    | maxElement |
‡    self do: [:each | | maxValue val |
        maxValue == nil
            ifFalse: [
                (val := aBlock value: each) > maxValue ifTrue: [
                    maxElement := each.
                    maxValue := val]]
            ifTrue: ["first element"
                maxElement := each.
                maxValue := aBlock value: each].
                "Note that there is no way to get the first element that works
                for all kinds of Collections. Must test every one."].
    ^ maxElement
```

because maxValue would be nil on every iteration of the loop. With this in mind let's try and look closer and try and identify which temporaries are declared at method level but only used at block level.

Review the previous post on the Closure Compiler and its closure analysis. TempVariableNodes have readingScopes and writingScopes instance variables tracking their references. We can detect temporaries that are declared at method level but have no reads or writes at method level. We need to parse methods, perform the closure analysis and then examine the remote temps. We can save time by using the filter above to only parse methods that have remote temps:

```
(SystemNavigation default allSelect:
    [:m|
    m isQuick not
    and: [(m at: m initialPC) = 138
    and: [(m at: m initialPC + 1) <= 127
    and: [| methodNode |
        methodNode := m methodClass parserClass new
                                parse: m getSourceFromFile
                                class: m methodClass.
        methodNode ensureClosureAnalysisDone.
        (methodNode instVarNamed: 'temporaries') last remoteTemps anySatisfy:
            [:t|
            ((t instVarNamed: 'readingScopes')
                ifNil: [true]
                ifNotNil: [:rs| (rs includesKey: methodNode block) not])
            and: [(t instVarNamed: 'writingScopes')
                    ifNil: [true]
```

```
                                        ifNotNil: [:ws| (ws includesKey: methodNode block)
not]]]]]]]) size
```

1507. A quiet year. The Aztec New Fire ceremony is held for the last time. da Vinvi completes the Mona Lisa. But no major conflicts. 1507 is apparently 7015 to 7016 in the Byzantine calendar. Don't you just *love* Wikipedia? Not much seems to have happened for sure in 1507 BC, although we have apparently discovered the first remains of domesticated ferrets here abouts. Now where were we? Ah yes…

So in fact only 2428 – 1507 = 921, or 921 / 67650.0 = 1.4% have an apparently legitimate need for remote temps. But that leaves at least 1507 methods that need editing to eliminate the unnecessary and bloated bytecodes. That's a lot of editing. Sigh.

Perhaps we can fix this automatically. We can already find the temps that need moving. Now we have to find where to move them to. How many methods are there that contain exactly one block? These should be easy to fix.

```
(SystemNavigation default allSelect:
    [:m|
    m isQuick not
    and: [(m at: m initialPC) = 138
    and: [(m at: m initialPC + 1) <= 127
    and: [ | numBlocks |
        numBlocks := 0.
        (InstructionStream on: m) scanFor:
            [:b|
            b = 143 ifTrue: [numBlocks := numBlocks + 1].
            numBlocks > 1].
        numBlocks = 1
    and: [| methodNode |
        methodNode := m methodClass parserClass new
                                    parse: m getSourceFromFile
                                    class: m methodClass.
        methodNode ensureClosureAnalysisDone.
        (methodNode instVarNamed: 'temporaries') last remoteTemps anySatisfy:
            [:t|
            ((t instVarNamed: 'readingScopes')
                ifNil: [true]
                ifNotNil: [:rs| (rs includesKey: methodNode block) not])
            and: [(t instVarNamed: 'writingScopes')
                        ifNil: [true]
                        ifNotNil: [:ws| (ws includesKey: methodNode block)
not]]]]]]]) size
```

645. That's 43%. Nearly half. This seems worth-while to me. What do we need to do? For the methods in question we have to find the set of method-level temp names that are only accessed remotely, edit them out of the method level declaration and add them to a block-level declaration. This is going to be complicated enough that we should probably use a class or two instead of doits. Besides I find syntax highlighting really helps. Let's put a halt where the action begins and we can root around to make sense of things.

```
Object subclass: #TempScopeEditor
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Cog-Scripts'

TempScopeEditor methods for source editing
fixBlockLocalTempsInSingleBlockMethods
    "Move temporaries declared at method level but used only at block
     level to block level scope in methods that only contan one block
     (because this should be easier than dealing with multiple block cases)."

    self candidateMethodsSuchThat:
            [:m| | numBlocks |
            numBlocks := 0.
            (InstructionStream on: m) scanFor:
                [:b|
                 b = 143 ifTrue: [numBlocks := numBlocks + 1].
                 numBlocks > 1].
```

```
                numBlocks = 1]
        do:     [:source :class :methodNode :remoteTemps| | tempsToMove blockNode
|
                tempsToMove := remoteTemps select: [:t|
                    ((t instVarNamed: 'readingScopes')
                        ifNil: [true]
                        ifNotNil: [:rs| (rs includesKey: methodNode block) not])
                    and: [(t instVarNamed: 'writingScopes')
                            ifNil: [true]
                            ifNotNil: [:ws| (ws includesKey: methodNode block) not]].
                blockNode := (methodNode accept: BlockNodeCollectingVisitor new) last.
                self halt]]
```

We'll adapt the script above to invoke the important block with some of the extra work already done:

```
TempScopeEditor methods for accessing
candidateMethodsSuchThat: methodSelectBlock do: quadBlock
    "Evaluate quadBlock with the source, class, methodNode and
     remoteTemporaries of all method that contain method-level
     remote temporaries for which methodSelectBlock answers true."
    SystemNavigation default allSelect:
        [:m| | methodNode remoteTemps |
        (m isQuick not
        and: [(m at: m initialPC) = 138
        and: [(m at: m initialPC + 1) <= 127
        and: [(methodSelectBlock value: m)
        and: [methodNode := m methodClass parserClass new
                                            parse: m getSourceFromFile
                                            class: m methodClass.
                methodNode ensureClosureAnalysisDone.
                remoteTemps := (methodNode instVarNamed: 'temporaries') last
remoteTemps.
                remoteTemps anySatisfy:
                    [:t|
                    ((t instVarNamed: 'readingScopes')
                        ifNil: [true]
                        ifNotNil: [:rs| (rs includesKey: methodNode block) not])
                    and: [(t instVarNamed: 'writingScopes')
                            ifNil: [true]
                            ifNotNil: [:ws| (ws includesKey: methodNode block)
not]]]]]]) ifTrue:
                [quadBlock
                    value: m getSourceFromFile asString
                    value: m methodClass
                    value: methodNode
                    value: remoteTemps].
        false]
```

And we need a visitor to fish the block node out of the parse tree:

```
ParseNodeVisitor subclass: #BlockNodeCollectingVisitor
    instanceVariableNames: 'blockNodes'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Cog-Scripts'

BlockNodeCollectingVisitor methods for accessing
blockNodes
    ^blockNodes

BlockNodeCollectingVisitor methods for visiting
visitBlockNode: aBlockNode
    (blockNodes ifNil: [blockNodes := OrderedCollection new]) addLast: aBlockNode
```

My first mistake is in assuming the last block will be the block I'm interested in. But the visitor finds all blocks, including optimized ones that will be inlined (as in ifTrue: [a block the compiler inlines]):

```
blockNode := ((methodNode accept: BlockNodeCollectingVisitor new)
                blockNodes reject: [:bn| bn optimized])
            last.
```

Better. Now let's get the start of the block from its source range, the same source range the debugger uses to high-light methods while debugging:

```
startOfBlock := (methodNode encoder sourceRangeFor: blockNode) first.
```

Let's evaluate

```
TempScopeEditor new fixBlockLocalTempsInSingleBlockMethods
```

amd root around at the halt. I want to evaluate

```
source copyFrom: 1 to: startOfBlock
```

in the debugger to see what's what. And… ***Bang!*** We're now two days into the future (with little progress having been made). Since this is Cinéma vérité I need to relate what happened. The VM crashed hard as soon as I evaluated *source copyFrom: 1 to: startOfBlock*. I spent two days trying to track down what I thought was a VM bug but was of course my own fault. My original version of the method in BytecodeEncoder to compute the size of a bytecode looked like this:

```
BytecodeEncoder methods for opcode sizing
sizeOpcodeSelector: selector withArguments: args
    | start |
    stream ifNil: [stream := WriteStream on: (ByteArray new: 64)].
    start := stream position.
    ^[self perform: selector withArguments: args.
     stream position – start]
        ensure: [stream position: start]
```

Later on I realized that the ensure: served little purpose. If there was an error while generating the code the compilation was useless and restoring the stream position pointless. What was important was to see the error and correct it. So I modified the method to

```
sizeOpcodeSelector: selector withArguments: args
    | start result |
    stream ifNil: [stream := WriteStream on: (ByteArray new: 64)].
    start := stream position.
    self perform: selector withArguments: args.
    result := stream position – start.
    stream position: start.
    ^result
```

and then early on while writing this post I realized that remembering the position was silly. I might as well position the stream to zero and simply answer its resulting position. I could then also use a smaller stream for sizing since only one opcode's worth of bytecode was ever going to be written to the stream:

```
sizeOpcodeSelector: genSelector withArguments: args
    stream ifNil: [stream := WriteStream on: (ByteArray new: 64)].
    stream position: 0.
    self perform: genSelector withArguments: args.
    ^stream position
```

Much nicer… …except that it resulted in a memory corruption that crashed the VM in the garbage collector. And fool that I am, I spent two days tracking down the bug thinking it was a VM bug related to my new closure bytecodes rather than to my immediately preceeding compiler "tweaking".

The bug resulted from DebuggerMethodMap, the abstraction I introduced to insulate the Debugger from different bytecode sets. A DebuggerMethodMap holds onto a BytecodeAgnosticMethodNode and uses it to compute both the set of temporary names to display in the debugger and the map of pc to source range for high-lighting the current expression. Like the existing code it replaced the temp names and source extents are derived from doing a scratch recompilation of the mehtod source. When the DebuggerMethodMap instance derived the source map after deriving the temp names in displaying the fixBlockLocalTempsInSingleBlockMethods method in the debugger it caused the BytecodeAgnosticMethodNode to reuse its BytecodeEncoder. The encoder already

had its stream initialised to the first method it generated to produce the temp names. So when

```
stream position: 0.
```

was evaluated this reset the stream to point at the header of the previously generated CompiledMethod whose header word was promptly corrupted when generating the bytecode to be sized. Soon thereafter the allocations done in evaluating *source copyFrom: 1 to: startOfBlock* caused the garbage collector to run which encountered the corrupted CompiledMethod header which completely screwed up the garbage collector's parsing of the heap and caused the hard crash. Ouch.

There are a few lessons to be learnt from this. One, which my own hubris means I have to relearn at alarming frequency is that if one has just changed something and the VM crashes soon thereafter it's almost certainly one's own bloody fault, dimwit!! Another is that a system is only as safe as one makes it. Array accesses in Smalltalk are safe, being bounds and type checked… except for blue-book CompiledMethod whose at:put: method does not check that a byte is being stored at or after a CompiledMethod's initialPC (this at least is fixable). I can hear Tim saying re blue-book CompiledMethods "I told you so!" :).

OK, so two days further on how should we fix this? Since all we need is the count of bytes written we can provide BytecodeEncoder with its own nextPut: and postion and we're done:

```
Encoder subclass: #BytecodeEncoder
    instanceVariableNames: 'stream position rootNode blockExtentsToLocals'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Compiler-Kernel'

BytecodeEncoder methods for opcode sizing
nextPut: aByte
    "For sizing make the encoder its own stream and
     keep track of position with this version of nextPut:"
    position := position + 1

sizeOpcodeSelector: genSelector withArguments: args
    stream := self.
    position := 0.
    self perform: genSelector withArguments: args.
    ^position
```

Much nicer.

Now this might seem much more like a documentary to you than Kino-Pravda but alas soon after writing the first version of the above mini-tragedy (it being driven by my own character flaws) I lost the VM bug story by inadvertently closing the workspace containing my carefully formatted post and saving the image. *Bang!* Hence I have spent some time in the editing room recreating this. Ouch #2.

So where were we? Right; we were trying to edit methods with misplaced block temporaries. Now I can evaluate *source copyFrom: 1 to: startOfBlock*. but it includes the entire block, not just the first '['. Hmm. My second mistake is in assuming the source range of a block is the entire block when in fact it is the last statement. Please wait while I modify the compiler… (perhaps this is more like Magical Realism …Done. Now each BlockNode assigns its entire source range to its closureCreationNode:

```
BlockNode methods for accessing
closureCreationNode
    closureCreationNode ifNil:
        [closureCreationNode := LeafNode new
                                    key: #closureCreationNode
                                    code: nil].
    ^closureCreationNode

BlockNode methods for initialize-release
noteSourceRangeStart: start end: end encoder: encoder
    "Note two source ranges for this node. One is for the debugger
     and is of the last expression, the result of the block. One is for
     source analysis and is for the entire block."
    encoder
```

```
                noteSourceRange: (start to: end)
                forNode: self closureCreationNode.
        startOfLastStatement
                ifNil:
                        [encoder
                                noteSourceRange: (start to: end)
                                forNode: self]
                ifNotNil:
                        [encoder
                                noteSourceRange: (startOfLastStatement to: end – 1)
                                forNode: self]
```

OK, so now our method looks like

```
TempScopeEditor methods for source editing
fixBlockLocalTempsInSingleBlockMethods
        "Move temporaries declared at method level but used only at block
         level to block level scope in methods that only contan one block
         (because this should be easier than dealing with multiple block cases)."

        self candidateMethodsSuchThat:
                        [:m| | numBlocks |
                        numBlocks := 0.
                        (InstructionStream on: m) scanFor:
                                [:b|
                                 b = 143 ifTrue: [numBlocks := numBlocks + 1].
                                 numBlocks > 1].
                        numBlocks = 1]
                do:   [:source :class :methodNode :remoteTemps|
                        | tempsToMove blockNode startOfBlock |
                        tempsToMove := remoteTemps select: [:t|
                                ((t instVarNamed: 'readingScopes')
                                        ifNil: [true]
                                        ifNotNil: [:rs| (rs includesKey: methodNode block) not])
                                and: [(t instVarNamed: 'writingScopes')
                                                ifNil: [true]
                                                ifNotNil: [:ws| (ws includesKey: methodNode block) not]].
                        blockNode := ((methodNode accept: BlockNodeCollectingVisitor new)
‡                                                       blockNodes reject: [:bn| bn optimized])
                                                last.
‡                        startOfBlock := (methodNode encoder sourceRangeFor: blockNode
closureCreationNode) first.
                        self halt]]
```

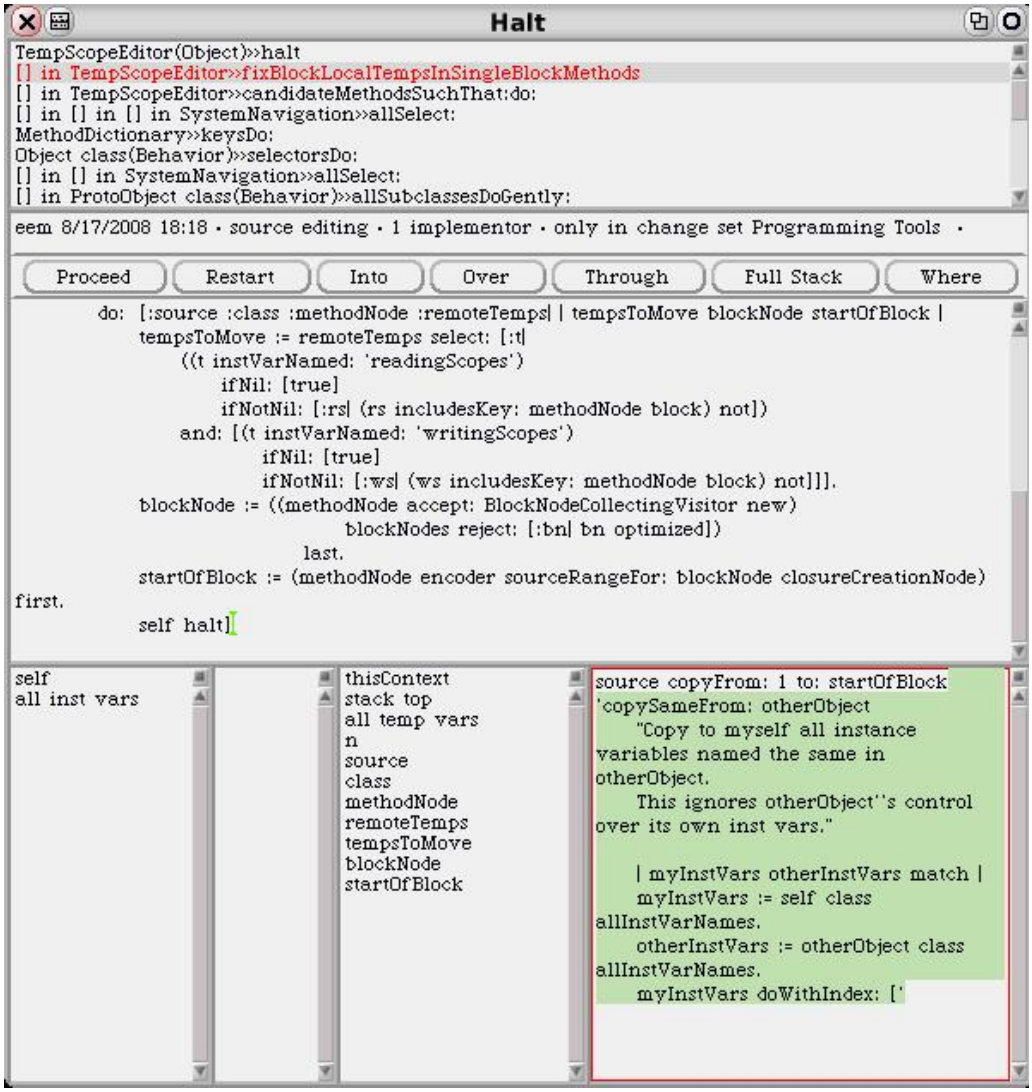
Let's evaluate it and root around the halt to see that my next error is a misplaced closing bracket. There are two following the halt and there should be three at the end of the block computing tempsToMove:

```
TempScopeEditor methods for source editing
fixBlockLocalTempsInSingleBlockMethods
        "Move temporaries declared at method level but used only at block
         level to block level scope in methods that only contan one block
         (because this should be easier than dealing with multiple block cases)."

        self candidateMethodsSuchThat:
                        [:m| | numBlocks |
                        numBlocks := 0.
                        (InstructionStream on: m) scanFor:
                                [:b|
                                 b = 143 ifTrue: [numBlocks := numBlocks + 1].
                                 numBlocks > 1].
                        numBlocks = 1]
                do:     [:source :class :methodNode :remoteTemps| | tempsToMove blockNode
startOfBlock |
                        tempsToMove := remoteTemps select: [:t|
                                ((t instVarNamed: 'readingScopes')
                                        ifNil: [true]
                                        ifNotNil: [:rs| (rs includesKey: methodNode block) not])
                                and: [(t instVarNamed: 'writingScopes')
```

```
                                    ifNil: [true]
                                    ifNotNil: [:ws| (ws includesKey: methodNode block) not]]].
                    blockNode := ((methodNode accept: BlockNodeCollectingVisitor new)
                                            blockNodes reject: [:bn| bn optimized])
                                    last.
                    startOfBlock := (methodNode encoder sourceRangeFor: blockNode
closureCreationNode) first.
                    self halt] ‡
```

Now source copyFrom: 1 to: startOfBlock is

```
'copySameFrom: otherObject
        "Copy to myself all instance variables named the same in otherObject.
        This ignores otherObject"s control over its own inst vars."

        | myInstVars otherInstVars match |
        myInstVars := self class allInstVarNames.
        otherInstVars := otherObject class allInstVarNames.
        myInstVars doWithIndex: ['
```



and tempsToMove is an OrderedCollection({match}). Now we can start editing, a process complex enough to need its own method. It will likely need the blockNode itself, and we'll want to eyeball the results:

fixBlockLocalTempsInSingleBlockMethods

```
    "Move temporaries declared at method level but used only at block
     level to block level scope in methods that only contan one block
     (because this should be easier than dealing with multiple block cases)."

    self candidateMethodsSuchThat:
                [:m| | numBlocks |
                numBlocks := 0.
                (InstructionStream on: m) scanFor:
                        [:b|
```

```
                        b = 143 ifTrue: [numBlocks := numBlocks + 1].
                        numBlocks > 1].
                numBlocks = 1]
        do:     [:source :class :methodNode :remoteTemps| | tempsToMove blockNode
newSource |
                tempsToMove := remoteTemps select: [:t|
                    ((t instVarNamed: 'readingScopes')
                        ifNil: [true]
                        ifNotNil: [:rs| (rs includesKey: methodNode block) not])
                    and: [(t instVarNamed: 'writingScopes')
                            ifNil: [true]
                            ifNotNil: [:ws| (ws includesKey: methodNode block) not]]].
                blockNode := ((methodNode accept: BlockNodeCollectingVisitor new)
                                    blockNodes reject: [:bn| bn optimized])
                                last.
‡               newSource := self moveTemps: tempsToMove to: blockNode in: source
encoder: methodNode encoder.
                (StringHolder new textContents:
                    (TextDiffBuilder buildDisplayPatchFrom: source to: newSource))
                        openLabel: 'temps edit'.
                self halt]
```

but since we need the methodNode to know which temporaries to keep my first cut ends up being

```
moveTemps: tempsToMove in: methodNode to: blockNode in: sourceString encoder:
encoder
    ^String streamContents:
        [:s|    | tempsToKeep tempsStart tempsEnd startOfBlock endOfArgs |
        tempsToKeep := methodNode block temporaries reject: [:t| tempsToMove
includes: t].
        startOfBlock := (encoder sourceRangeFor: blockNode closureCreationNode)
first.
        endOfArgs := sourceString indexOf: $| startingAt: startOfBlock.
        tempsStart := sourceString indexOf: $|.
        tempsEnd := sourceString indexOf: $| startingAt: tempsStart + 1.
        tempsToKeep isEmpty
            ifTrue:
                [s
                    next: tempsStart – 1 putAll: sourceString;
                    next: endOfArgs – tempsEnd putAll: sourceString startingAt:
tempsEnd + 1.
                ]
            ifFalse:
                [s next: (sourceString indexOf: $|) putAll: sourceString.
                 tempsToKeep do: [:t| s space; nextPutAll: t name; space].
                 s space; next: endOfArgs – tempsEnd + 1 putAll: sourceString
startingAt: tempsEnd].
        s space; nextPut: $|.
        tempsToMove do: [:t| s space; nextPutAll: t name; space].
        s space; nextPut: $|.
        s next: sourceString size – endOfArgs putAll: sourceString startingAt:
endOfArgs + 1]
```

which produces the following diff:

```
    copySameFrom: otherObject
        "Copy to myself all instance variables named the same in otherObject.
        This ignores otherObject's control over its own inst vars."

        | myInstVars otherInstVars i iLimiT <0-6> |
        | myInstVars otherInstVars match |
        myInstVars := self class allInstVarNames.
        otherInstVars := otherObject class allInstVarNames.
        myInstVars doWithIndex: [:each :index | | match |
        myInstVars doWithIndex: [:each :index |
            (match := otherInstVars indexOf: each) > 0 ifTrue:
                [self instVarAt: index put: (otherObject instVarAt: match)]].
```

```
1 to: (self basicSize min: otherObject basicSize) do: [:i |
    self basicAt: i put: (otherObject basicAt: i)].
```

Progress. We need to filter-out the implicit temporary "<0-6>" used to hold indirect temps:

```
tempsToKeep := (methodNode instVarNamed: 'temporaries') reject:
                    [:t| t isIndirectTempVector or: [tempsToMove includes: t]].
```

and filter-out implicit temporaries the compiler introduces when optimizing to:do: blocks (i & iLimiT):

```
tempsToKeep := (methodNode instVarNamed: 'temporaries') reject:
                    [:t| t isIndirectTempVector or: [t scope < 0 or: [tempsToMove includes: t]]].
```

and include the indirect temps the implicit temp vector holds:

```
tempsToKeep := OrderedCollection new.
methodNode block temporaries do:
    [:t|
    t isIndirectTempVector
        ifTrue:
            [t remoteTemps do: [:remoteTemp|
                (tempsToMove includes: remoteTemp) ifFalse:
                    [tempsToKeep addLast: remoteTemp]]]
        ifFalse:
            [(t scope < 0 or: [tempsToMove includes: t]) ifFalse:
                [tempsToKeep addLast: t]]].
```

And there are extra spaces following each temporary:

```
            tempsToKeep do: [:t| s space; nextPutAll: t name].
        …
tempsToMove do: [:t| s space; nextPutAll: t name].
```

Alright, a correct edit!

```
copySameFrom: otherObject
    "Copy to myself all instance variables named the same in otherObject.
    This ignores otherObject's control over its own inst vars."

    | myInstVars otherInstVars |
    | myInstVars otherInstVars match |
    myInstVars := self class allInstVarNames.
    otherInstVars := otherObject class allInstVarNames.
    myInstVars doWithIndex: [:each :index |
        (match := otherInstVars indexOf: each) > 0 ifTrue:
            [self instVarAt: index put: (otherObject instVarAt: match)]].
    1 to: (self basicSize min: otherObject basicSize) do: [:i | | match |
    1 to: (self basicSize min: otherObject basicSize) do: [:i |
        self basicAt: i put: (otherObject basicAt: i)].
```

So now instead of eyeballing let's parse the result to test what we've got so far. We'll print to the transcript to track progress:

```
fixBlockLocalTempsInSingleBlockMethods
    "Move temporaries declared at method level but used only at block
     level to block level scope in methods that only contan one block
     (because this should be easier than dealing with multiple block cases)."

    self candidateMethodsSuchThat:
            [:m| | numBlocks |
            numBlocks := 0.
            (InstructionStream on: m) scanFor:
                [:b|
                 b = 143 ifTrue: [numBlocks := numBlocks + 1].
                 numBlocks > 1].
            numBlocks = 1]
        do:     [:source :class :methodNode :remoteTemps| | tempsToMove blockNode
```

```
newSource |
		Transcript cr; print: class; nextPutAll: '>>'; print: methodNode selector;
flush.
		tempsToMove := remoteTemps select: [:t|
			((t instVarNamed: 'readingScopes')
				ifNil: [true]
				ifNotNil: [:rs| (rs includesKey: methodNode block) not])
			and: [(t instVarNamed: 'writingScopes')
					ifNil: [true]
					ifNotNil: [:ws| (ws includesKey: methodNode block) not]]].
		blockNode := ((methodNode accept: BlockNodeCollectingVisitor new)
					blockNodes reject: [:bn| bn optimized])
				last.
		newSource := self	moveTemps: tempsToMove
					in: methodNode
					to: blockNode
					in: source
					encoder: methodNode encoder.
		class parserClass new parse: newSource class: class.
		"(StringHolder new textContents:
			(TextDiffBuilder buildDisplayPatchFrom: source to: newSource))
				openLabel: 'temps edit'."]
```

Bang! **Error: subscript is out of bounds: -6**. Huh? This while trying to edit

```
Object methods for *Tweak-Core-Object
get: fieldName
	| var |
	^self instVarAt: (self class allInstVarNames indexOf: fieldName ifAbsent:[
		^(var := self class bindingOf: fieldName) ifNotNil:[var value].
	])
```

And of course there are no arguments or temporaries in

```
	[^(var := self class bindingOf: fieldName) ifNotNil:[var value].]
```

so endOfArgs is zero and the

```
	next: endOfArgs – tempsEnd + 1 putAll: sourceString startingAt: tempsEnd
```

expression ends up moving the stream backwards 6 characters! So:

```
		endOfArgs := blockNode arguments isEmpty
				ifTrue: [startOfBlock]
				ifFalse: [sourceString indexOf: $| startingAt:
startOfBlock].
```

And now the parser chugs along parsing about 90 edited methods until a SyntaxError:

```
	readsTweakField: field
		"Answer whether the receiver reads the given field"
		| toGet scanner byte type max offset |
		toGet := field toGet ifNil:[^false].
		(self hasLiteral: toGet) ifFalse:[^false].
		max := self numLiterals.
		"We scan the first sixteen accurately"
		max > 16 ifTrue:[max := 16].
		1 to: self numLiterals do:[:Name is already defined ->i|
			(self literalAt: i) == toGet ifTrue:[
				"scan for push: self; send: toGet"
				scanner := InstructionStream on: self.
				scanner scanFor: [:insn|
					(insn = 16r70 "push self") ifTrue:[
						byte := self at: scanner pc+1.
						type := byte // 16.
						offset := byte \\ 16.
						type > 12 ifTrue:[
							(offset+1 = i) ifTrue:[^true].
						].
					].
					false
```

```
                ].
            ].
        ].
        17 to: self numLiterals do:[:i|
            (self literalAt: i) == toGet ifTrue:[^true].
        ].
        ^false
```

Here i is remote because it is implicitly assigned to at the end of the 1 to: self numLiterals do:[:i|…] loop and read by the

```
                [:insn|
                    (insn = 16r70 "push self") ifTrue:[
                        byte := self at: scanner pc+1.
                        type := byte // 16.
                        offset := byte \\ 16.
                        type > 12 ifTrue:[
                            (offset+1 = i) ifTrue:[^true].
                        ].
                    ].
                    false
                ]
```

block. So we need to filter-out implicitly declared temps from tempsToMove in both arms:

```
        methodNode block temporaries do:
            [:t|
            t isIndirectTempVector
                ifTrue:
                    [t remoteTemps do: [:rt|
                        (rt scope < 0 or: [tempsToMove includes: rt]) ifFalse:
                            [tempsToKeep addLast: rt]]]
                ifFalse:
                    [(t scope < 0 or: [tempsToMove includes: t]) ifFalse:
                        [tempsToKeep addLast: t]]].
```

Which results in 135 successful parses until Color class>>colorTest:extent:colorMapper:. One reason to keep the StringHolder diff expression in a comment in the fixBlockLocalTempsInSingleBlockMethods method is so I can easily evaluate it in the debugger and see the diff:

```
    colorTest: depth extent: chartExtent colorMapper: colorMapper
        "Create a palette of colors sorted horizontally by hue and vertically by lightness.
Useful for eyeballing the color gamut of the display, or for choosing a color interactively."
        "Note: It is slow to build this palette, so it should be cached for quick access."
        "(Color colorTest: 32 extent: 570@180 colorMapper: [:c | startHue palette
transHt vSteps transCaption grayWidth hSteps x y c | Color
        "(Color colorTest: 32 extent: 570@180 colorMapper: [:c | c]) display"
        "(Color colorTest: 32 extent: 570@180 colorMapper:
            [:c | Color
                r: (c red * 7) asInteger / 7
                g: (c green * 7) asInteger / 7
                b: (c blue * 3) asInteger / 3]) display"
        "(Color colorTest: 32 extent: 570@180 colorMapper:
            [:c | Color
                r: (c red * 5) asInteger / 5
                g: (c green * 5) asInteger / 5
                b: (c blue * 5) asInteger / 5]) display"
        "(Color colorTest: 32 extent: 570@180 colorMapper:
            [:c | Color
                r: (c red * 15) asInteger / 15
                g: (c green * 15) asInteger / 15
                b: (c blue * 15) asInteger / 15]) display"
        "(Color colorTest: 32 extent: 570@180 colorMapper:
            [:c | Color
                r: (c red * 31) asInteger / 31
                g: (c green * 31) asInteger / 31
                b: (c blue * 31) asInteger / 31]) display"
```

```
        | basicHue x y c startHue palette transHt vSteps transCaption grayWidth hSteps
|
        …
```

Oops! My editor is naive enough to be fooled by code inside comments. We need to search for the method-level temps from the start of the temporaries proper. Luckily the parser has a pair of methods that can be used to do just this. Parser>>parseMethodComment:setPattern: parses the selector and arguments and any initial comments, leaving the parser at the first non-comment token of the method body. Parser>>startOfNextToken answers this token's source position. So let's add this as an argument to the editing method. Lets also factor out the tempsToKeep computation for readability.

```
fixBlockLocalTempsInSingleBlockMethods
    "Move temporaries declared at method level but used only at block
     level to block level scope in methods that only contan one block
     (because this should be easier than dealing with multiple block cases)."

    self candidateMethodsSuchThat:
            [:m| | numBlocks |
            numBlocks := 0.
            (InstructionStream on: m) scanFor:
                [:b|
                 b = 143 ifTrue: [numBlocks := numBlocks + 1].
                 numBlocks > 1].
            numBlocks = 1]
        do:    [:source :class :methodNode :remoteTemps| | tempsToMove blockNode
methodBodyStart newSource |
            Transcript cr; print: class; nextPutAll: '>>'; print: methodNode selector;
flush.
            tempsToMove := remoteTemps select: [:t|
                ((t instVarNamed: 'readingScopes')
                    ifNil: [true]
                    ifNotNil: [:rs| (rs includesKey: methodNode block) not])
                and: [(t instVarNamed: 'writingScopes')
                        ifNil: [true]
                        ifNotNil: [:ws| (ws includesKey: methodNode block) not]]].
            blockNode := ((methodNode accept: BlockNodeCollectingVisitor new)
                                blockNodes reject: [:bn| bn optimized])
                            last.
‡            methodBodyStart := class parserClass new
                                parseMethodComment: source
setPattern: [:ignored|];
                                startOfNextToken.
            newSource := self    moveTemps: tempsToMove
                                in: methodNode
                                to: blockNode
                                source: source
                                methodBodyStart: methodBodyStart
                                encoder: methodNode encoder.
            class parserClass new parse: newSource class: class.
            "(StringHolder new textContents:
                (TextDiffBuilder buildDisplayPatchFrom: source to: newSource))
                    openLabel: 'temps edit'."]

moveTemps: tempsToMove in: methodNode to: blockNode source: sourceString
methodBodyStart: methodBodyStart encoder: encoder
    ^String streamContents:
        [:s| | tempsToKeep tempsStart tempsEnd startOfBlock endOfArgs |
‡         tempsToKeep := self tempsToKeepAtMethodLevelOf: methodNode
tempsToMove: tempsToMove.
        startOfBlock := (encoder sourceRangeFor: blockNode closureCreationNode)
first.
        endOfArgs := blockNode arguments isEmpty
                            ifTrue: [startOfBlock]
                            ifFalse: [sourceString indexOf: $| startingAt:
startOfBlock].
        tempsStart := sourceString indexOf: $| startingAt: methodBodyStart.
```

```
            tempsEnd := sourceString indexOf: $| startingAt: tempsStart + 1.
‡           tempsToKeep isEmpty
                ifTrue:
                    [s next: tempsStart – 1 putAll: sourceString;
                     next: endOfArgs – tempsEnd putAll: sourceString startingAt:
tempsEnd + 1]
                ifFalse:
                    [s next: tempsStart putAll: sourceString.
                     tempsToKeep do: [:t| s space; nextPutAll: t name].
                     s space; next: endOfArgs – tempsEnd + 1 putAll: sourceString
startingAt: tempsEnd].
            s space; nextPut: $|.
            tempsToMove do: [:t| s space; nextPutAll: t name].
            s space; nextPut: $|.
            s next: sourceString size – endOfArgs putAll: sourceString startingAt:
endOfArgs + 1]

tempsToKeepAtMethodLevelOf: methodNode tempsToMove: tempsToMove
    | tempsToKeep |
    tempsToKeep := OrderedCollection new.
    methodNode block temporaries do:
        [:t|
        t isIndirectTempVector
            ifTrue:
                [t remoteTemps do: [:rt|
                    (rt scope < 0 or: [tempsToMove includes: rt]) ifFalse:
                        [tempsToKeep addLast: rt]]]
            ifFalse:
                [(t scope < 0 or: [tempsToMove includes: t]) ifFalse:
                    [tempsToKeep addLast: t]]].
    ^tempsToKeep
```

Good. Now we can edit and parse 228 methods before an obvious case I should have thought of, that of moving temps into a block that already has block-local temps:

```
    privateAddAllMorphs: aCollection atIndex: index
        "Private. Add aCollection of morphs to the receiver"
        | myWorld otherSubmorphs |
        | myWorld itsWorld otherSubmorphs |
        myWorld := self world.
        otherSubmorphs := submorphs copyWithoutAll: aCollection.
        (index between: 0 and: otherSubmorphs size)
            ifFalse: [^ self error: 'index out of range'].
        index = 0
            ifTrue:[        submorphs := aCollection asArray, otherSubmorphs]
            ifFalse:[        index = otherSubmorphs size
                ifTrue:[        submorphs := otherSubmorphs, aCollection]
                ifFalse:[        submorphs := otherSubmorphs copyReplaceFrom:
index + 1 to: index with: aCollection ]].
        aCollection do: [:m | | itsWorld | | itsOwner |
        aCollection do: [:m | | itsOwner |
            itsOwner := m owner.
            itsOwner ifNotNil: [
                itsWorld := m world.
                (itsWorld == myWorld) ifFalse: [
                    itsWorld ifNotNil: [self privateInvalidateMorph: m].
                    m outOfWorld: itsWorld].
                (itsOwner ~~ self) ifTrue: [
                    m owner privateRemove: m.
                    m owner removedMorph: m ]].
            m privateOwner: self.
            myWorld ifNotNil: [self privateInvalidateMorph: m].
            (myWorld == itsWorld) ifFalse: [m intoWorld: myWorld].
            itsOwner == self ifFalse: [
                self addedMorph: m.
                m noteNewOwner: self ].
        ].
        self layoutChanged.
```

Obviously we've potentially got the same problem finding the start of the block-level temps as finding the start of the method-level temps; a comment could come before the block-level temps and include a | character. But this is really unlikely style so let's just do the simple thing:

```
moveTemps: tempsToMove in: methodNode to: blockNode source: sourceString
methodBodyStart: methodBodyStart encoder: encoder
	^String streamContents:
		[:s| | tempsToKeep tempsStart tempsEnd startOfBlock endOfArgs restStart
blockTempsStart |
		tempsToKeep := self tempsToKeepAtMethodLevelOf: methodNode
tempsToMove: tempsToMove.
		startOfBlock := (encoder sourceRangeFor: blockNode closureCreationNode)
first.
		endOfArgs := blockNode arguments isEmpty
					ifTrue: [startOfBlock]
					ifFalse: [sourceString indexOf: $| startingAt:
startOfBlock].
		tempsStart := sourceString indexOf: $| startingAt: methodBodyStart.
		tempsEnd := sourceString indexOf: $| startingAt: tempsStart + 1.
		tempsToKeep isEmpty
			ifTrue:
				[s next: tempsStart – 1 putAll: sourceString;
				 next: endOfArgs – tempsEnd putAll: sourceString startingAt:
tempsEnd + 1]
			ifFalse:
				[s next: tempsStart putAll: sourceString.
				 tempsToKeep do: [:t| s space; nextPutAll: t name].
				 s space; next: endOfArgs – tempsEnd + 1 putAll: sourceString
startingAt: tempsEnd].
‡		 blockNode temporaries isEmpty
			ifTrue:
				[s space; nextPut: $|.
				tempsToMove do: [:t| s space; nextPutAll: t name].
				s space; nextPut: $|.
				restStart := endOfArgs + 1]
			ifFalse:
				[blockTempsStart := sourceString indexOf: $| startingAt: endOfArgs
+ 1.
				 s next: blockTempsStart – endOfArgs + 1 putAll: sourceString
startingAt: endOfArgs + 1.
				 tempsToMove do: [:t| s space; nextPutAll: t name].
				 restStart := blockTempsStart + 1].
		s next: sourceString size – restStart + 1 putAll: sourceString startingAt:
restStart]
```

Boom! It now blows up much earlier. Here's a diff:

```
	nextOrNilSuchThat: aBlock
		"Answer the next object that satisfies aBlock, skipping any intermediate objects.
		If no object has been sent, answer <nil> and leave me intact.
		NOTA BENE: aBlock MUST NOT contain a non-local return (^)."

		| value |
		accessProtect critical: [[
		| value readPos |
		accessProtect critical: [
			value := nil.
			readPos := readPosition.
			[readPos < writePosition and: [value isNil]] whileTrue: [
				value := contentsArray at: readPos.
				readPos := readPos + 1.
				(aBlock value: value) ifTrue: [
					readPosition to: readPos – 1 do: [ :j | readPos
					readPosition to: readPos – 1 do: [ :j |
						contentsArray at: j put: nil.
					].
```

```
                        readPosition := readPos.
                    ] ifFalse: [
                        value := nil.
                    ].
                ].
                readPosition >= writePosition ifTrue: [readSynch initSignals].
            ].
            ^value
```

because the critical: block only has implicit temporaries (the i in readPosition to: readPos – 1 do: [:j |…]). We could try filtering-out implicit temps as in

```
(blockNode temporaries select: [:t| t scope >= 0]) isEmpty
```

but this won't work for a couple of reasons. One, the compiler marks block temporaries as out-of-scope once a block has been parsed so all block temps are going to have their scopes less than zero in any parse tree. Two, there could be a temporary declaration in an optimized block nested within it and the start of those nested temporaries won't be anywhere near the right place. We need to search for the block's temporaries immediately following the arguments:

```
maybeBlockTempsStart := sourceString indexOf: $| startingAt: endOfArgs + 1
ifAbsent: sourceString size + 1.
((sourceString copyFrom: endOfArgs + 1 to: maybeBlockTempsStart – 1)
allSatisfy:
    [:c| c isSeparator])
    ifTrue:
        [s next: maybeBlockTempsStart – endOfArgs + 1 putAll:
sourceString startingAt: endOfArgs + 1.
         tempsToMove do: [:t| s space; nextPutAll: t name].
         restStart := maybeBlockTempsStart + 1]
    ifFalse:
        [s space; nextPut: $|.
        tempsToMove do: [:t| s space; nextPutAll: t name].
        s space; nextPut: $|.
        restStart := endOfArgs + 1].
s next: sourceString size – restStart + 1 putAll: sourceString startingAt:
restStart]
```

and lo and behold all 645 odd methods are edited and parse correctly. Arguably we've got by on a wing and a prayer, but that's at least part of the XP way. Do the simplest thing that could possibly work. I haven't been able to demonstrate how quickly this can be done because I've been writing this as I've gone along, and have had a couple of unexpected stumbles along the way, so it has been far from fleet. But I hope I've demonstrated how exploratory programming has got us to a solution quite easily, the debugger being a great tool to use to understand a problem. Had we started off with BUFD I might have been lazy or intimidated and given up; after all the code still works without the editing. But using the environment we've come up with a solution and it's been fun (at least for me). But we're not done yet. We need to generalize to methods with more than one block and we need to check for read-before-written temps before this is a tool we an use to eliminate unnecessary remote temps and proceed to take some meaningful measurements.

A More Comprehensive Implementation

Now we understand the problem enough that we can do some up-front design. To generalise we need to move temporaries to their correct block scope. Handling multiple blocks doesn't change the fact that we're moving temps so the start of the process, deleting the temps from method-level scope, is unchanged. Further, since block declarations follow each other, if we iterate through blocks in source order our cheap and cheerful editing process can be extended. We need to maintain the position in the source we've got to so far and output from this point up through the current block's temporaries.

Let's make the editor deal with a single method at a time. We can get all of the information we need to compile from the method. We need an output stream on which to write the modifed source, a map of some sort indicating which temps to move to which block and we need the block nodes in the method. In extending the source editing approach we'll need to know where we are in the input source so far. We also need the encoder since source ranges for blocks are obtained from it.

```
Object subclass: #TempScopeEditor
    instanceVariableNames: 'method out tempMap blockNodes sourceString soFar encoder'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Cog-Scripts'
```

```
TempScopeEditor methods for initialize-release
forMethod: aCompiledMethod
    | methodNode |
    method := aCompiledMethod.
    sourceString := aCompiledMethod getSourceFromFile asString.
    methodNode := method methodClass parserClass new
                                parse: sourceString
                                class: method methodClass.
    methodNode ensureClosureAnalysisDone.
    encoder := methodNode encoder.
    blockNodes := (methodNode accept: BlockNodeCollectingVisitor new)
                                blockNodes reject: [:bn| bn optimized].
    out := (String new: sourceString size) writeStream.
    tempMap := IdentityDictionary new
```

We need to edit the method. That'll be the main entry point. We need to construct the temp map and then edit it. I'll end with a halt for now, to check the output.

```
TempScopeEditor methods for editing
edit
    self buildTempMap.
    self copyMethodMovingTemps.
    self halt
```

Since visitors visit the parse nodes in order of evaluation we can find the smallest enclosing scope by choosing the last block that fully encloses a temp.

```
TempScopeEditor methods for editing
buildTempMap
    "Build the map for moving remote temps. Each remote temp
     that should be moved is entered into the map referencing its
     smallest enclosing scope. This may seem backwards but it
     means that the map is one-to-one, not one-to-many."
    blockNodes do:
        [:blockNode|
        (blockNode temporaries notEmpty
         and: [blockNode temporaries last isIndirectTempVector]) ifTrue:
            [blockNode temporaries last remoteTemps do:
                [:remoteTemp| | enclosingScopes smallestEnclosingBlockScope |
                 enclosingScopes := blockNodes select: [:blockScope|
                                                                    self
blockNode: blockScope

    isEnclosingScopeFor: remoteTemp].
                    enclosingScopes notEmpty ifTrue:
                        [smallestEnclosingBlockScope := enclosingScopes last.
                         smallestEnclosingBlockScope ~~ blockNode ifTrue:
                            [tempMap at: remoteTemp put:
smallestEnclosingBlockScope]]]]]
```

Finding out whether a block is a valid scope is easy if we find the converse. Reject temps that have any references outside of a given block.

```
TempScopeEditor methods for editing
blockNode: aBlockNode isEnclosingScopeFor: aTempVariableNode
    ^((self
        anyScopes: (aTempVariableNode instVarNamed: 'readingScopes')
        outsideExtent: aBlockNode blockExtent)
    or: [self
        anyScopes: (aTempVariableNode instVarNamed: 'writingScopes')
        outsideExtent: aBlockNode blockExtent]) not

anyScopes: referenceScopeDict outsideExtent: blockExtent
    ^referenceScopeDict notEmpty
```

```
        and: [referenceScopeDict anySatisfy:
                [:set|
                set anySatisfy: [:location| (blockExtent rangeIncludes: location) not]]]
```

If we don't find any temps to move the method doesn't need editing. So our edit method should be more like

```
TempScopeEditor methods for editing
edit
    self buildTempMap.
    tempMap isEmpty ifFalse:
        [self copyMethodMovingTemps.
         self halt.
         method methodClass compile: out contents classified: method
methodReference category]
```

Now we can adapt the old code. We need to enumerate over all blocks except the first one, which is actually a block node for the entire method.

```
TempScopeEditor methods for editing
copyMethodMovingTemps
    | methodBodyStart tempsToKeep tempsStart tempsEnd |
    methodBodyStart := method methodClass parserClass new
                                    parseMethodComment: sourceString setPattern:
[:ignored|];
                                    startOfNextToken.
    (tempsToKeep := self tempsToKeepAtMethodLevel) isEmpty
        ifTrue:
            [soFar := 1]
        ifFalse:
            [tempsStart := sourceString indexOf: $| startingAt: methodBodyStart.
             tempsEnd := sourceString indexOf: $| startingAt: tempsStart + 1.
             out next: tempsStart putAll: sourceString.
             tempsToKeep do: [:t| out space; nextPutAll: t name].
             soFar := tempsEnd].
    blockNodes allButFirst do:
        [:blockNode|
        self processBlockNode: blockNode].
    out next: sourceString size – soFar + 1 putAll: sourceString startingAt: soFar
```

Since the first block is that of the entire method we can compute the set of method-level temps to keep from it. But we need to compute temps to keep for each block node so we can factor-out. But there's a wrinkle. The compiler marks all block temps as out of scope on leaving the block but not all method-level scopes. Hence

```
TempScopeEditor methods for editing
tempsToKeepAtMethodLevel
    ^(self tempsToKeepFor: blockNodes first) select:
        [:t|t scope >= 0]

tempsToKeepFor: blockNode
    | tempsToKeep |
    tempsToKeep := OrderedCollection new.
    blockNode temporaries do:
        [:t|
        t isIndirectTempVector
            ifTrue:
                [t remoteTemps do:
                    [:rt|
                    (tempMap includesKey: rt) ifFalse:
                        [tempsToKeep addLast: rt]]]
            ifFalse:
                [tempsToKeep addLast: t]].
    ^tempsToKeep
```

So the Big Kahuna is

```
TempScopeEditor methods for editing
processBlockNode: blockNode
    | tempsToMoveHere startOfBlock endOfArgs maybeBlockTempsStart
```

```
blockTempsInSource |
	tempsToMoveHere := (tempMap select: [:aBlockNode| aBlockNode == blockNode])
keys.
	startOfBlock := (encoder sourceRangeFor: blockNode closureCreationNode) first.
	endOfArgs := blockNode arguments isEmpty
				ifTrue: [startOfBlock]
				ifFalse: [sourceString indexOf: $| startingAt: startOfBlock].
	out next: endOfArgs – soFar + 1 putAll: sourceString startingAt: soFar.
	maybeBlockTempsStart := sourceString indexOf: $| startingAt: endOfArgs + 1
ifAbsent: sourceString size + 1.
	blockTempsInSource := (sourceString copyFrom: endOfArgs + 1 to:
maybeBlockTempsStart – 1) allSatisfy:
					[:c| c isSeparator].
	blockTempsInSource
		ifTrue:
			[out next: maybeBlockTempsStart – endOfArgs putAll: sourceString
startingAt: endOfArgs + 1.
			 (self tempsToKeepFor: blockNode) do:
				[:tempNode| out space; nextPutAll: tempNode name].
			 tempsToMoveHere do: [:t| out space; nextPutAll: t name].
			 soFar := sourceString indexOf: $| startingAt: maybeBlockTempsStart + 1.
			 (sourceString at: soFar – 1) isSeparator ifTrue:
				[soFar := soFar – 1]]
		ifFalse:
			[out space; nextPut: $|.
			tempsToMoveHere do: [:t| out space; nextPutAll: t name].
			out space; nextPut: $|.
			soFar := endOfArgs + 1]
```

Finding an interesting method to test, one that has an indirect temp in a block that needs moving let's test it:

```
	"Find a block scope with a temp that needs moving."
	| scanner |
	scanner := InstructionStream new.
	SystemNavigation default browseAllSelect:
		[:m| | seenBlock |
		seenBlock := false.
		m isQuick not
		and: [scanner method: m pc: m initialPC.
			scanner scanFor:
				[:b|
				b = 143 ifTrue: [seenBlock := true].
				seenBlock and: [b = 138 and: [scanner followingByte <= 127]]]]]
```

which throws up SARInstaller class>>ensurePackageWithId:

```
ensurePackageWithId: anIdString

	self squeakMapDo: [ :sm | | card newCS |
		self withCurrentChangeSetNamed: 'updates' do: [ :cs |
			newCS := cs.
			card := sm cardWithId: anIdString.
			(card isNil or: [ card isInstalled not or: [ card isOld ]])
				ifTrue: [ sm installPackageWithId: anIdString ]
		].
		newCS isEmpty ifTrue: [ ChangeSet removeChangeSet: newCS ]
	].
```

Editing it works nicely:

```
	ensurePackageWithId: anIdString

		self squeakMapDo: [ :sm | | card newCS |
			self withCurrentChangeSetNamed: 'updates' do: [ :cs |
		self squeakMapDo: [ :sm | | newCS |
			self withCurrentChangeSetNamed: 'updates' do: [ :cs | | card |
				newCS := cs.
				card := sm cardWithId: anIdString.
				(card isNil or: [ card isInstalled not or: [ card isOld ]])
```

```
                    ifTrue: [ sm installPackageWithId: anIdString ]
            ].
            newCS isEmpty ifTrue: [ ChangeSet removeChangeSet: newCS ]
        ].
```

To this we need to add a read-before-written checker and we should be able to fix all methods in the system (rather than all conceivable methods). We can implement another simple visitor:

```
ParseNodeVisitor subclass: #ReadBeforeWrittenVisitor
    instanceVariableNames: 'readBeforeWritten written'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Cog-Scripts'

ReadBeforeWrittenVisitor methods for accessing
readBeforeWritten
    ^readBeforeWritten ifNil: [IdentitySet new]

ReadBeforeWrittenParseNodeVisitor methods for visiting
visitAssignmentNode: anAssignmentNode
    anAssignmentNode variable isTemp ifTrue:
        [written ifNil: [written := IdentitySet new].
        written add: anAssignmentNode variable]

visitTempVariableNode: aTempVariableNode
    (aTempVariableNode isArg
     or: [written notNil
        and: [written includes: aTempVariableNode]]) ifTrue:
        [^self].
    readBeforeWritten ifNil:
        [readBeforeWritten := IdentitySet new].
    readBeforeWritten add: aTempVariableNode
```

and let's test it.

```
    | mc |
    mc := #none.
    SystemNavigation default browseAllSelect:
        [:m|
        m methodClass ~~ mc ifTrue:
            [Transcript cr; print: (mc := m methodClass); flush].
        m isQuick not
        and: [(m methodNode accept: ReadBeforeWrittenVisitor new)
readBeforeWritten notEmpty]]
```

Let's see what the editor does before adding the read-before-written check:

```
TempScopeEditor methods for editing
editNoCompile
    self buildTempMap.
    ^tempMap isEmpty ifFalse:
        [self copyMethodMovingTemps.
         out contents]
```

and test it on something that has a read-before-written temp:

```
    (TempScopeEditor new forMethod: Collection >> #detectMax:) editNoCompile
```

produces

```
    'detectMax: aBlock
        "Evaluate aBlock with each of the receiver"s elements as the argument.
        Answer the element for which aBlock evaluates to the highest magnitude.
        If collection empty, return nil. This method might also be called elect:."

        | maxElement|
        self do: [:each | | maxValue val |
            maxValue == nil
                ifFalse: [
                    (val := aBlock value: each) > maxValue ifTrue: [
                        maxElement := each.
```

```
                                maxValue := val]]
                        ifTrue: ["first element"
                                maxElement := each.
                                maxValue := aBlock value: each].
                                "Note that there is no way to get the first element that works
                                for all kinds of Collections. Must test every one."].
        ^ maxElement'
```

Indeed, maxValue is incorrectly moved into the inner scope. But there's also a minor bug with the missing space before the closing vertical bar. Let's fix that.

```
TempScopeEditor methods for editing
copyMethodMovingTemps
    | methodBodyStart tempsToKeep tempsStart tempsEnd |
    methodBodyStart := method methodClass parserClass new
                                parseMethodComment: sourceString setPattern:
[:ignored|];
                                startOfNextToken.
    (tempsToKeep := self tempsToKeepAtMethodLevel) isEmpty
        ifTrue:
            [soFar := 1]
        ifFalse:
            [tempsStart := sourceString indexOf: $| startingAt: methodBodyStart.
             tempsEnd := sourceString indexOf: $| startingAt: tempsStart + 1.
             out next: tempsStart putAll: sourceString.
             tempsToKeep do: [:t| out space; nextPutAll: t name].
             soFar := tempsEnd.
‡             (sourceString at: soFar – 1) isSeparator ifTrue:
                [soFar := soFar – 1]].
    blockNodes allButFirst do:
        [:blockNode|
        self processBlockNode: blockNode].
    out next: sourceString size – soFar + 1 putAll: sourceString startingAt: soFar
```

To add the read-before-written check I need to make sure the visitor visits the same parse tree we derive the block nodes from so that the visitor answers temp nodes in the block node's tree. I need to make the method node an instance variable. I can derive the encoder from the method node, so…

```
Object subclass: #TempScopeEditor
    instanceVariableNames: 'method methodNode out tempMap blockNodes sourceString soFar'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Cog-Scripts'
```

```
TempScopeEditor methods for editing
buildTempMap
    "Build the map for moving remote temps. Each remote temp
     that should be moved is entered into the map referencing its
     smallest enclosing scope. This may seem backwards but it
     means that the map is one-to-one, not one-to-many."
    | readBeforeWritten |
‡    readBeforeWritten := (methodNode accept: ReadBeforeWrittenVisitor new)
readBeforeWritten.
    blockNodes do:
        [:blockNode|
        (blockNode temporaries notEmpty
         and: [blockNode temporaries last isIndirectTempVector]) ifTrue:
            [blockNode temporaries last remoteTemps do:
                [:remoteTemp| | enclosingScopes smallestEnclosingBlockScope |
‡                 (readBeforeWritten includes: remoteTemp) ifFalse:
                    [enclosingScopes := blockNodes select: [:blockScope|
                                                                    self
blockNode: blockScope

    isEnclosingScopeFor: remoteTemp].
                     enclosingScopes notEmpty ifTrue:
                        [smallestEnclosingBlockScope := enclosingScopes last.
                         smallestEnclosingBlockScope ~~ blockNode ifTrue:
```

```
                                        [tempMap at: remoteTemp put:
smallestEnclosingBlockScope]]]]]]
```

and now

```
(TempScopeEditor new forMethod: Collection >> #detectMax:) editNoCompile
```

produces

```
‘detectMax: aBlock
        "Evaluate aBlock with each of the receiver”s elements as the argument.
        Answer the element for which aBlock evaluates to the highest magnitude.
        If collection empty, return nil. This method might also be called elect:."

        | maxElement maxValue |
        self do: [:each | | val |
                maxValue == nil
                        ifFalse: [
                                (val := aBlock value: each) > maxValue ifTrue: [
                                        maxElement := each.
                                        maxValue := val]]
                        ifTrue: ["first element"
                                maxElement := each.
                                maxValue := aBlock value: each].
                                "Note that there is no way to get the first element that works
                                for all kinds of Collections. Must test every one."].
        ^ maxElement’
```

Measurements

Good. I hope we’re done. Let’s count and see the change:

```
| numMethods numMethodsWithClosure numMethodsWithIndirectTemps
 numClosures numClosuresWithCopiedValues numCopiedValues
 numRemoteTemps numScopesWithRemoteTemps |
numMethods := numMethodsWithClosure := numMethodsWithIndirectTemps :=
numClosures := numClosuresWithCopiedValues := numCopiedValues :=
numRemoteTemps := numScopesWithRemoteTemps := 0.
SystemNavigation default allSelect:
        [:m| | s hasBlock hasIndirectTemps |
        hasBlock := hasIndirectTemps := false.
        s := InstructionStream on: m.
        s scanFor:
                [:b|
                b = 143 "closure creation" ifTrue:
                        [hasBlock := true.
                        numClosures := numClosures + 1.
                        s followingByte >= 16 ifTrue:
                                [numClosuresWithCopiedValues :=
numClosuresWithCopiedValues + 1.
                                 numCopiedValues := numCopiedValues + (s followingByte >>
4)]].
                (b = 138 "indirect temp vector creation"
                 and: [s followingByte <= 127]) ifTrue:
                        [hasIndirectTemps := true.
                         numScopesWithRemoteTemps := numScopesWithRemoteTemps +
1.
                         numRemoteTemps := numRemoteTemps + s followingByte].
                false].
        numMethods := numMethods + 1.
        hasBlock ifTrue: [numMethodsWithClosure := numMethodsWithClosure + 1].
        hasIndirectTemps ifTrue: [numMethodsWithIndirectTemps :=
numMethodsWithIndirectTemps + 1].
        false].
{ numMethods. numMethodsWithClosure. numMethodsWithIndirectTemps.
 numClosures. numCopiedValues. numClosuresWithCopiedValues.
 numRemoteTemps. numScopesWithRemoteTemps }
```

gives

```
#(67701 10966 2430 18380 14843 9249 4828 2436)
```

So 10966 / 677.01 = 16.2% of methods contain a closure. Prior to editing 2430 / 109.66 = 22.2% of these methods need indirect temps.

Now let's edit. We only need look at methods which create a closure:

```
SystemNavigation default allSelect:
    [:m| | s |
    s := InstructionStream on: m.
    (s scanFor:
            [:b|
             b = 143 "closure creation"
             and: [s followingByte >= 16]]) ifTrue:
        [(TempScopeEditor new forMethod: m) edit].
    false]
```

Boom! SyntaxError for Object>>get:

```
get: fieldName
    | var |
    ^self instVarAt: (self class allInstVarNames indexOf: fieldName ifAbsent:[ |
Name is already defined ->var |
        ^(var := self class bindingOf: fieldName) ifNotNil:[var value].
    ])
```

I'm forgetting to remove temporaries if all method-level temps are moved. So

```
TempScopeEditor methods for editing
copyMethodMovingTemps
    | methodBodyStart tempsToKeep tempsStart tempsEnd |
    methodBodyStart := method methodClass parserClass new
                                    parseMethodComment: sourceString setPattern:
[:ignored|];
                                    startOfNextToken.
‡   tempsStart := sourceString indexOf: $| startingAt: methodBodyStart.
‡   tempsEnd := sourceString indexOf: $| startingAt: tempsStart + 1.
    (tempsToKeep := self tempsToKeepAtMethodLevel) isEmpty
        ifTrue:
‡           [out next: tempsStart – 1 putAll: sourceString.
            soFar := tempsEnd + 1]
        ifFalse:
‡           [out next: tempsStart putAll: sourceString.
            tempsToKeep do: [:t| out space; nextPutAll: t name].
            soFar := tempsEnd.
            (sourceString at: soFar – 1) isSeparator ifTrue:
                [soFar := soFar – 1]].
    blockNodes allButFirst do:
        [:blockNode|
        self processBlockNode: blockNode].
    out next: sourceString size – soFar + 1 putAll: sourceString startingAt: soFar
```

Boom! SyntaxError for EventSensor >> #eventTickler

```
eventTickler
    "Poll infrequently to make sure that the UI process is not been stuck.
    If it has been stuck, then spin the event loop so that I can detect the
    interrupt key."
    | delay |
    delay := Delay forMilliseconds: self class eventPollPeriod.
    self lastEventPoll.        "ensure not nil."
    [| delta |
    [ delay wait.
    delta := Time millisecondClockValue – lastEventPoll.
    (delta < 0
            or: [delta > self class eventPollPeriod])
        ifTrue: ["force check on rollover"
            self fetchMoreEvents]] on: Error do: [:ex | | | ].
    true ] whileTrue.
```

This has a method-level temp declared inside an optimized block, a case I was hoping not to encounter. Let me see if I can just ignore these. Hopefully there shouldn't be very many

of them.

```
SystemNavigation default browseAllSelect:
	[:m| | s browseDueToSyntaxError |
	browseDueToSyntaxError := false.
	s := InstructionStream on: m.
	(s scanFor:
			[:b|
			 b = 143 "closure creation"
			 and: [s followingByte >= 16]]) ifTrue:
		[[(TempScopeEditor new forMethod: m) edit]
			on: SyntaxErrorNotification
			do: [:ex|
				browseDueToSyntaxError := true.
				ex return]].
		false].
	browseDueToSyntaxError]
```

But I also notice there is an empty temp declaration in the on: Error do: [:ex | | |] error handler block. So…

```
TempScopeEditor methods for editing
processBlockNode: blockNode
	| tempsToMoveHere startOfBlock endOfArgs maybeBlockTempsStart
blockTempsInSource |
	tempsToMoveHere := (tempMap select: [:aBlockNode| aBlockNode == blockNode])
keys.
‡	tempsToMoveHere isEmpty ifTrue: [^self].
	startOfBlock := (methodNode encoder sourceRangeFor: blockNode
closureCreationNode) first.
	endOfArgs := blockNode arguments isEmpty
				ifTrue: [startOfBlock]
				ifFalse: [sourceString indexOf: $| startingAt: startOfBlock].
	out next: endOfArgs – soFar + 1 putAll: sourceString startingAt: soFar.
	maybeBlockTempsStart := sourceString indexOf: $| startingAt: endOfArgs + 1
ifAbsent: sourceString size + 1.
	blockTempsInSource := (sourceString copyFrom: endOfArgs + 1 to:
maybeBlockTempsStart – 1) allSatisfy:
					[:c| c isSeparator].
	blockTempsInSource
		ifTrue:
			[out next: maybeBlockTempsStart – endOfArgs putAll: sourceString
startingAt: endOfArgs + 1.
			 (self tempsToKeepFor: blockNode) do:
				[:tempNode| out space; nextPutAll: tempNode name].
			 tempsToMoveHere do: [:t| out space; nextPutAll: t name].
			 soFar := sourceString indexOf: $| startingAt: maybeBlockTempsStart + 1.
			 (sourceString at: soFar – 1) isSeparator ifTrue:
				[soFar := soFar – 1]]
		ifFalse:
			[out space; nextPut: $|.
			 tempsToMoveHere do: [:t| out space; nextPutAll: t name].
			 out space; nextPut: $|.
			 soFar := endOfArgs + 1]
```

Lot's of editing going on but then MessageNotUnderstood: UndefinedObject>>notEmpty

```
	anyScopes: referenceScopeDict outsideExtent: blockExtent
		^referenceScopeDict notEmpty
		 and: [referenceScopeDict anySatisfy:
				[:set|
				set anySatisfy: [:location| (blockExtent rangeIncludes: location) not]]]
```

Oops. A TempVariableNode's reading (or writing) scope's dictionary will be nil if there are no reads (or writes) to it. So this needs to be

```
TempScopeEditor methods for editing
anyScopes: referenceScopeDict outsideExtent: blockExtent
	^referenceScopeDict notNil
	 and: [referenceScopeDict notEmpty
```

```
        and: [referenceScopeDict anySatisfy:
                [:set|
                set anySatisfy: [:location| (blockExtent rangeIncludes: location) not]]]
```

Great. 13 syntax errors. So lots of methods edited and the system is still running 🙂 But there is a syntax error reported for SARInstaller class>>ensurePackageWithId: and it has been misedited, missing a vertical bar before newCS:

```
    ensurePackageWithId: anIdString

        self squeakMapDo: [ :sm | newCS |
            self withCurrentChangeSetNamed: 'updates' do: [ :cs | | card |
                newCS := cs.
                card := sm cardWithId: anIdString.
                (card isNil or: [ card isInstalled not or: [ card isOld ]])
                    ifTrue: [ sm installPackageWithId: anIdString ]
            ].
            newCS isEmpty ifTrue: [ ChangeSet removeChangeSet: newCS ]
        ].
```

Ah, here there are no method-level temps so the copyMethodMovingTemps method confuses the temp declaration in the block for a method-level temp declaration. The tempsStart is for method-level only if tempsStart is before the start of the first proper block. So…

```
TempScopeEditor methods for editing
copyMethodMovingTemps
    | methodBodyStart tempsToKeep tempsStart tempsEnd |
    methodBodyStart := method methodClass parserClass new
                                    parseMethodComment: sourceString setPattern:
[:ignored|];
                                    startOfNextToken.
    tempsStart := sourceString indexOf: $| startingAt: methodBodyStart.
    tempsEnd := sourceString indexOf: $| startingAt: tempsStart + 1.
    (tempsToKeep := self tempsToKeepAtMethodLevel) isEmpty
        ifTrue:
            [| startOfFirstBlock |
‡               startOfFirstBlock := (methodNode encoder sourceRangeFor: blockNodes
second closureCreationNode) first.
             tempsStart < startOfFirstBlock
                ifTrue:
                    [out next: tempsStart – 1 putAll: sourceString.
                    soFar := tempsEnd + 1]
                ifFalse:
                    [soFar := 1]]
        ifFalse:
            [out next: tempsStart putAll: sourceString.
             tempsToKeep do: [:t| out space; nextPutAll: t name].
             soFar := tempsEnd.
             (sourceString at: soFar – 1) isSeparator ifTrue:
                [soFar := soFar – 1]].
    blockNodes allButFirst do:
        [:blockNode|
        self processBlockNode: blockNode].
    out next: sourceString size – soFar + 1 putAll: sourceString startingAt: soFar
```

Great, 1426 methods edited successfully and 13 syntax errors that I have to fix-up by hand. I'm happy. Of course this would have been much easier with Lisp (and arguably using the Refactoring Editor, but my current understanding is that the publicly available version doesn't preserve formatting). But writing 8 methods to edit 1426 automatically feels like a win to me.

What were these syntax errors? There are two cases where a temp that needs to be moved is declared within an optimized control structure, so we were right not to bother handling this case. The others are stores into block arguments, something the old compiler allows but the new compiler only allows if one sets a preference (allowBlockArgumentAssignment). Of these about half are short-cuts and half are cases where the block temp is being nilled because if compiled with the old compiler the temp

would be at method level and could cause the system to hold onto the block temp's value. See e.g. WorldState>>alarmSortBlock:

WorldState methods for alarms
alarmSortBlock

```
| answer |

"Please pardon the hackery below. Since the block provided by this method is retained elsewhere, it is possible that the block argument variables would retain references to objects that were no longer really needed. In one case, this feature resulted in doubling the size of a published project."

^[ :alarm1 :alarm2 |
    answer := alarm1 scheduledTime < alarm2 scheduledTime.
    alarm1 := alarm2 := nil.
    answer
]
```

Of course with closures none of this is necessary since the closure doesn't hold onto its local variables, only the activation of the closure does, and the activation doesn't persist beyond its invocation. So the above becomes simply

WorldState methods for alarms
alarmSortBlock

```
^[ :alarm1 :alarm2 | alarm1 scheduledTime < alarm2 scheduledTime]
```

But these are changes we're going to have to make manually. At least the syntax errors allow us to spot them.

So evaluating the counter above now answers

```
#(67702 10966 1481 18380 13763 8910 2241 1518)
```

which is a healthy change from

```
#(67701 10966 2430 18380 14843 9249 4828 2436)
```

Let me list this properly so I don't go insane:

```
Methods:                    67702
MethodsWithClosure:         10966
MethodsWithIndirectTemps:   1481
Closures:                   18380
CopiedValues:               13763
ClosuresWithCopiedValues:   8910
RemoteTemps:                2241
ScopesWithRemoteTemps:      1518
```

So out of 67702 methods 10966 (16.2%) create a closure, but only 1481 (2.2%) require remote temps. This corresponds somewhat to the 1.7% of methods in VisualWorks 5i that used remote temps, but perhaps indicates a stylistic difference between the two systems because of closures.

Of 18380 closures 8910 (48.5%) require copied values, and they average 13763 / 8910 = 1.5 copied values per copying closure, or 13763 / 18380 = 0.75 copied values per generic closure. Of 13763 copied values 2241 (16.3%) are for remote temps, so 83.7% of copied values are read-only copies of arguments and temporaries. I can't easily measure how many closures require remote temps by scanning the bytecode. I'll leave that as an exercise for the reader 😉 But it should be approximately 1481 / 10966 * 18380 / 100 or 25% of all closures.

I've responded to the figure of essentially half of all closures needing copied values by modifying the closure design to store copied values in indexable fields of the closure (something Paolo Bonzini asked about). You'll find the latest version of the bootstrap uses this representation for BlockClosure:

```
Object variableSubclass: #BlockClosure
    instanceVariableNames: 'outerContext startpc numArgs'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Kernel-Methods'
```

While this arrangement currently precludes adding instance variables to BlockClosure, or using subclasses of BlockClosure, in any new object representation in new versions of

Cog, I will try and make determining the number of named instance variables cheap enough that the closure value primitives will be able to accept subclasses of BlockClosure with additional instance variables.

Cool, Clean Closures

There are a few other things we should measure, namely how often a block does an up-arrow return and how often a block references self or self's instance variables, and how often bocks make no access to their enclosing environment at all. To do either of these things the closure has to reference its enclosing context object. [*This isn't strictly correct; one could access self and its instance variables via a copied value, as does VisualWorks, but this is quite complex, and definitely against the spirit of DTSTTCPW.*] Since the VM will evolve into one that doesn't create contexts unless absolutely necessary, if a block doesn't need to reference its enclosing context it wll be faster to create, since the enclosing context will not have to be created as well. In other words, in a VM that optimizes contexts away the current closure design involves at least two allocations, one for the closure and one for the context.

Blocks that don't access their outer environment at all, blocks like alarmSortBlock above, which don't need any copied values, don't access self or its instance variables and don't do up-arrow returns, so called "clean blocks" clearly don't need to have a context instantiated, but my current design requires a context because the closure accesses the method through the context. VisualWorks calls these self-contained blocks "clean" blocks, and, because they make no reference to the outside environment the VisualWorks compiler can create them at compile time. Were we to modify the closure value primitives suitably we could do the same thing. One implementation would be to make the outerContext instance variable do double duty as either an outer context or a method, e.g.

```
Object variableSubclass: #BlockClosure
    instanceVariableNames: 'methodOrOuterContext startpc numArgs'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Kernel-Methods'
```

and modify the primitiveClosureValue methods to distinguish between a context or a compiled method (easy, sicne both contexts and methdos have a special unique field value in their object header) and substitute nil as the receiver in the block's activation when activating a block whose methodOrOuterContext instance variable holds a method.

All this is easy to do and is probably worth-while for performance but it does have the down side that clean blocks don't identify themselves properly in the debugger since they don't have an outer context to refer to. In the VisualWorks debugger when you're in a clean block you can't see the values of any variables in the outer context, because the clean block isn't created within any specific method activation. You can see only in which method the block is defined. So for the moment I've deferred implementing this optimization. In any case the VM modifications are small. Most of the work is in the bytecode compiler and this could easily be a system preference so that the compiler would only produce clean blocks if so desired by the more power-hungry programmer.

OK, let's measure those clean vs non-clean statisics. Because this is Cinema Verite and not a documentary I'm going to allow myself a short-cut. I'm writing this last section a long time after the earlier measurements. In the interim I've made some changes to the compiler (mostly fixes to the decompiler) and have changed the closure representation of copied values as described. So these measurements won't match exactly with those above but you'll see they're really close. Hey, if a work of genius like The Wrong Trousers can survive a few continuity glitches then so can a ropey old blog like this! [*um, The Wrong trowsers can survive continuity glitches precisely because it is a work of genius. Your ropey old blog, on the other hand, could do well to pull its socks up and show a little humility.* ed.]

Here's the workspace script that also collect the additional measurements.

```
| numMethods numMethodsWithClosure numMethodsWithIndirectTemps
 numClosures numClosuresWithCopiedValues numCopiedValues
 numRemoteTemps numScopesWithRemoteTemps
 upArrowReturns usesSelfs upArrowReturnAndUsesSelfs numClean |

numMethods := numMethodsWithClosure := numMethodsWithIndirectTemps :=
numClosures := numClosuresWithCopiedValues := numCopiedValues :=
numRemoteTemps := numScopesWithRemoteTemps :=
upArrowReturns := usesSelfs := upArrowReturnAndUsesSelfs := numClean := 0.
SystemNavigation default allSelect:
```

```
	[:m| | s hasBlock hasIndirectTemps blkPc blkSz doesUAR usesSelf hasCopied |
	hasBlock := hasIndirectTemps := false.
	s := InstructionStream on: m.
	s scanFor:
		[:b|
		b = 143 "closure creation" ifTrue:
			[hasBlock := true.
			numClosures := numClosures + 1.
			s followingByte >= 16 ifTrue:
				[numClosuresWithCopiedValues :=
numClosuresWithCopiedValues + 1.
				 numCopiedValues := numCopiedValues + (s followingByte >>
4)]].
		(b = 138 "indirect temp vector creation"
		 and: [s followingByte <= 127]) ifTrue:
			[hasIndirectTemps := true.
			 numScopesWithRemoteTemps := numScopesWithRemoteTemps +
1.
			 numRemoteTemps := numRemoteTemps + s followingByte].
		false].
	numMethods := numMethods + 1.
	hasBlock ifTrue:
		[numMethodsWithClosure := numMethodsWithClosure + 1.
		 s pc: m initialPC; scanFor: [:b| b = 143].
		 blkSz := s interpretNextInstructionFor: BlockStartLocator new.
		 blkPc := s pc.
		 doesUAR := usesSelf := false.
		 hasCopied := s followingByte >= 16.
		 s scanFor:
			[:b|
			s pc >= (blkPc + blkSz)
				ifTrue: [true]
				ifFalse:
					[doesUAR := doesUAR or: [s willReturn and: [s
willBlockReturn not]].
					 usesSelf := usesSelf or: [b = 112 "pushSelf"
											or: [b < 16 "pushInstVar"
											or: [(b = 128 and: [s
followingByte <= 63]) "pushInstVar"
											or: [(b between: 96 and: 96 +
7) "storePopInstVar"
											or: [(b = 130 and: [s
followingByte <= 63]) "storePopInstVar"
											or: [(b = 129 and: [s
followingByte <= 63]) "storeInstVar"
											or: [b = 132 and: [s
followingByte = 160]]]]]]].
					false]].
		 doesUAR ifTrue:
			[upArrowReturns := upArrowReturns + 1].
		 usesSelf ifTrue:
			[usesSelfs := usesSelfs + 1].
		 (doesUAR and: [usesSelf]) ifTrue:
			[upArrowReturnAndUsesSelfs := upArrowReturnAndUsesSelfs + 1].
		 (doesUAR or: [usesSelf or: [hasCopied]]) ifFalse:
			[numClean := numClean + 1]].
	hasIndirectTemps ifTrue: [numMethodsWithIndirectTemps :=
numMethodsWithIndirectTemps + 1].
	false].
  { {'Methods'. numMethods}. {'MethodsWithClosure'. numMethodsWithClosure}.
{'MethodsWithIndirectTemps'. numMethodsWithIndirectTemps}.
   {'Closures'. numClosures}. {'CopiedValues'. numCopiedValues}.
{'ClosuresWithCopiedValues'. numClosuresWithCopiedValues}.
   {'RemoteTemps'. numRemoteTemps}. {'ScopesWithRemoteTemps'.
numScopesWithRemoteTemps}.
   {'UpArrowReturns'. upArrowReturns}. {'ReferencesSelf'. usesSelfs}. {'Both'.
```

upArrowReturnAndUsesSelfs}.
	{'Clean'. numClean} }

which produces

```
Methods:                    67805
MethodsWithClosure:         11022
MethodsWithIndirectTemps:   1545
Closures:                   18437
CopiedValues:               13860
ClosuresWithCopiedValues:   8933
RemoteTemps:                2366
ScopesWithRemoteTemps:      1578
UpArrowReturns:             1025
ReferencesSelf:             4049
Both:                       346
Clean:                      188
```

Clearly clean blocks are too rare to bother with; only 1% are clean. 20% access self. 5.6% of closures use ^-return. 1.9% do both. So it could be worth-while considering an implementation where closures can hold onto their receiver independently of the outerContext, because a whopping 94.4% of closures don't need their outer context. I'm a bit peeved because in my earliest sketches I had an explicit receiver slot in BlockClosure and I took it out thinking that it was simpler to access receiver and method though the outer context. The greast thing about software is that nothing is set in stone so there will be opportunity to revisit this later.

After far too long a break the next posts will describe the stack interpreter which, inside the VM, invisibly maps contexts to stack frames.

Posted by admin on Friday, November 14th, 2008, at 3:41 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. **Steve Rees** | 15-Nov-08 at 1:13 pm | Permalink

   Love the alliteration in the title and that Ed is clearly one smart fellow 😉

   One thing I would have found helpful when reading the Smalltalk code in the post would have been highlighting of the changes in the code, particularly as some of the methods are a bit longer than I can easily digest in a quick sweep, but perhaps that's just me – brain the size of a pea and all.

   Still, an interesting post. I look forward to the follow-ups on the stack interpreter.

2. **tim@rowledge.org** | 15-Nov-08 at 5:47 pm | Permalink

   As requested – "hrrm. I told you so"

3. **Eliot Miranda** | 16-Nov-08 at 2:22 pm | Permalink

   Hi Steve, (thanks Tim!)

   brain the size of a pea?! You too, eh? I think it's a great suggestion to high-light the operative pieces of the code. I should have thought of that in the first place, but my pea wasn't up to it. I've used double daggers to point out relevant changes. Hopefully there are enough. Please point out where things are still murky.

   Best
   Eliot

4. **Nicolas Cellier** | 26-Dec-09 at 6:11 pm | Permalink

   I had to add a
   super visitBlockNode: aBlockNode
   in the BlockNodeCollectingVisitor and update #forMethod:
   That seems to be working great

Post a Comment

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

Name *

Email *

Website

Message

Post

« CLOSURES PART III – THE COMPILER SIMULATE OUT OF THE BOCHS »