

HOME

{ 2008 06 07 }

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

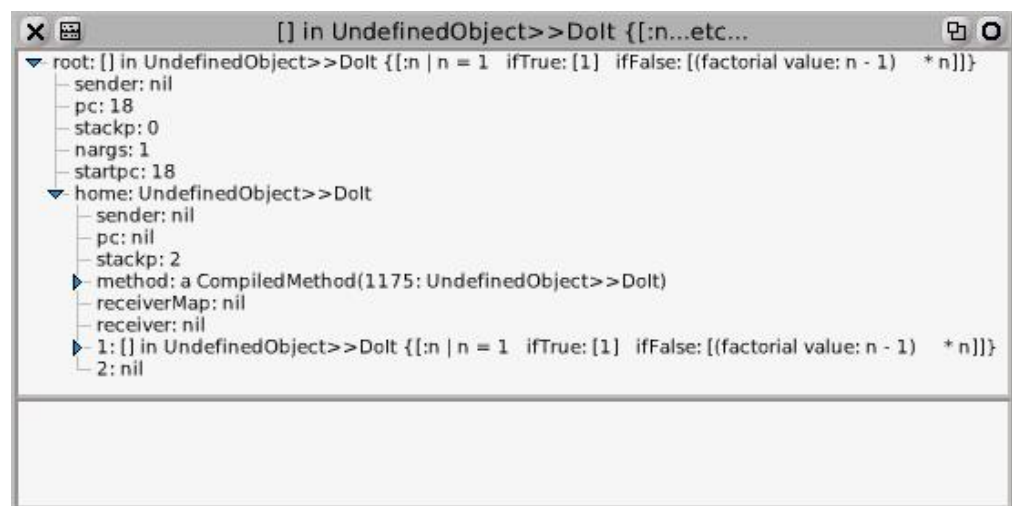
Closures Part I

[BlueBook BlockContexts](#) are nearly closures. They close-over their enclosing environment, providing access to an enclosing method's arguments and temporary variables. But they lack their own local environment, hijacking their method activation's (or home context's) temps to store their own arguments and temporaries (let's call these locals). Worse still, they're not reentrant. BTW, "the home context" is the terminology for the activation record of a method. All blocks get created within some method activation (or within a block activation nested within a method activation). The method activation in which a block is created is called the block's home context.

Try the following in [Squeak](#) prior to Squeak 4.x, and in [VisualWorks](#):

```
| factorial |
factorial := [:n | n = 1 ifTrue: [1] ifFalse: [(factorial value: n - 1) * n]].
(1 to: 10) collect: factorial
```

In VisualWorks you get #(1 2 6 24 120 720 5040 40320 362880 3628800). In Squeak you get a Notifier proclaiming "Error: Attempt to evaluate a block that is already being evaluated." The Squeak code fails because in a Blue Book VM an unevaluated block is an activation record, a BlockContext, that has a sender field used to refer to the block's caller. You can inspect or explore the block to see it. Here's an explorer on it:



You can see the block is referred to from the first temporary of the home context "[factorial]". You can see its sender field which is used to refer to the calling context, the context that sends value: to the block to evaluate it. The block can't be reentered without overwriting the sender field, preventing return to the first caller.

OK, let's deal with the reentrancy problem. Simply evaluating a copy of the block instead of the original produces a fresh activation record for each activation of the block. This is what Allen Wirfs-Brock's team did in the Tektronix implementation on the 4404 back in the '80's. They modified the BlockContext>>value[:value:...] primitives to activate a copy of the receiver. Let's try modifying the example with explicit copies:

```
| factorial |
factorial := [:n | n = 1 ifTrue: [1] ifFalse: [(factorial copy value: n - 1) * n]].
```

(1 to: 10) collect: factorial copy #(1 1 1 1 1 1 1 1 1 1)

which answers... #(1 1 1 1 1 1 1 1 1 1) ?!?!)

That's because the above is actually compiled to code equivalent to

```
| factorial n |
factorial := [:factorialsArgument
  n := factorialsArgument.
  n = 1 ifTrue: [1] ifFalse: [(factorial copy value: n - 1) * n]].
(1 to: 10) collect: factorial copy
```

because with BlueBook BlockContexts all temporaries are stored in the temporary frame of the home context. In the explorer above the "2: nil" field in the home context is the slot used to store the block's argument "n".

By the time we've recursed to the base case with $n = 1$ we've overwritten n with 1 in all the " $n * n$ "s, and so end up evaluating $1 * 1 * 1 \dots$

This way round works as intended:

```
| factorial n |
factorial := [:factorialsArgument
  n := factorialsArgument.
  n = 1 ifTrue: [1] ifFalse: [n * (factorial copy value: n - 1)].
(1 to: 10) collect: factorial copy
```

answers #(1 2 6 24 120 720 5040 40320 362880 3628800), because in " $n * (factorial copy value: n - 1)$ " n is pushed on the stack before the recursion due to Smalltalk's strict left-to-right evaluation rule.

Whether you think this is a big deal or not its not ANSI-compliant so we've got to fix it right? I'm all for standards, and for me not having closures is an issue. It gets in the way of self-expression. It is useful to have recursive blocks. And with Smalltalk's system building machinery it is easy to fix. But its not a huge issue; Squeak has done without closures for thirty years. However, there are also pragmatic reasons for fixing this. Closures are an enabler for efficient implementation of context-to-stack mapping, something to be explained in an upcoming post, which will speed-up the interpreter as well as the fast VM. In fact so effective is the form of this optimization that I plan to implement that when I reimplemented VisualWorks' closures using it execution of block-intensive code such as exception handling sped up by a factor of two. These specifics are why I chose to do my own closure implementation instead of reusing [Anthony Hannan's one](#).

Implementing Closures

Implementing closures is straight-forward. We need three things. One is to defer creating the activation of the block until we send value:. So creating the block results in creating something, a BlockClosure, from which we can create an activation later on. Another is the ability for a block activation to hold local arguments and temporaries. We already have something that can do this, its called MethodContext :). Finally we need a way of accessing locals in enclosing block or method activations. This has already been done for Squeak. Anthony Hannan did a closure implementation which exists in a rump form in 3.8. But I didn't use it because for efficient context-to-stack mapping I want one key ingredient which is to implement access to locals in enclosing activations without access through those activations.

To explain the scheme, which is used in some Lisp compilers, let's look at the following:

counterBlock

```
| count |
count := 0.
[ count := count + 1 ].
```

If counter is stored on the stack of the method activation of counterBlock then there's a problem if activations are mapped to stack frames for the duration of their execution. Up until the block is returned counter can live happily on some native stack frame created when counterBlock was sent. But since the block [counter := counter + 1] outlives the execution of counterBlock we must preserve the value of counter for subsequent evaluations of the block. Since we have contexts the thing to do is to create a context object and associate it with the frame when creating a block that accesses the frame, and to write back the contents of the stack frame into its context when a frame that has a context is returned from. The sad thing is that this return-time processing is very slow.

What we need to do is to break the dependency between the block activation and its enclosing contexts for accessing locals. I'm going to use Collection>>inject:into: as an example.

Collection methods for enumerating

inject: thisValue into: binaryBlock

"Accumulate a running value associated with evaluating the argument, binaryBlock, with the current value of the argument, thisValue, and the receiver as block arguments. For instance, to sum the numeric elements of a collection, aCollection inject: 0 into: [:subTotal :next | subTotal + next]."

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

The block [:each | nextValue := binaryBlock value: nextValue value: each] reads the method argument binaryBlock and reads and writes nextValue. If we allocate an explicit array to hold nextValue indirectly (something I'm going to call an "indirect temp vector") we can avoid writing the local in the method activation:

inject: thisValue into: binaryBlock

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

Now the block only reads the locals of the method activation, and none of the locals it reads changes value after the block is created. So the block can keep a private copy of those values without affecting semantics. The compiler does this behind the scenes but the code is somewhat equivalent to

inject: thisValue into: binaryBlock

```
| indirectTemps |
indirectTemps := Array new: 1.
indirectTemps at: 1 put: thisValue.
self do: (thisContext
    closureCopy:
        [:each || binaryBlockCopy indirectTempsCopy |
            indirectTempsCopy
                at: 1
                put: (binaryBlockCopy
                    value: (indirectTempsCopy at: 1)
                    value: each)]
    copiedValues: (Array with: binaryBlock with: indirectTemps)).
^indirectTemps at: 1
```

closureCopy:copiedValues: answers a BlockClosure that holds the pc in the

method to start executing the block's code (just like BlockContext's startpc) and the array of copiedValues. When the block is activated the copiedValues are pushed onto the activation's stack after any block arguments, becoming locals of the activation. Now there is no dependency on the enclosing activation for local access. Nothing needs to happen when returning from an activation that encloses some closure so returns are simple and hence fast.

The compiler analysis to do this is extremely simple. Any local that is accessed by an inner scope must either be copied or put in an indirect temp vector. If a local is assigned to after it is closed over (after a block is created that accesses the local) then the local must be made indirect. We can do slightly better than this, but it's not worth the effort. This simple analysis works fine. I'll detail the analysis in a post on the closure compiler and its resultant bytecodes. Note that we only need one indirect temp vector per scope, it needs an element to each indirect local.

Prototyping The Implementation

The above facilities, creating a closure, creating an indirect temp vector and reading and writing from it, and the evaluation primitives all make sense when implemented in the VM with specific bytecodes and primitives. But one thing that's very nice about Smalltalk is that one can prototype the above scheme without any VM modifications. We can't prototype non-local return without trickery but blocks like factorial above work fine. I did just this before I implemented the closure bytecodes. For example here's the implementations of closureCopy:copiedValues:, the message one can use to create Closures, and value:, one of the evaluation primitives that can be written in pure Smalltalk due to it having first-class activation records. Neat!

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

ContextPart methods for controlling

closureCopy: numArgs copiedValues: anArray

```
"Distinguish a block of code from its enclosing method by
creating a BlockClosure for that block. The compiler inserts into all
methods that contain blocks the bytecodes to send the message
closureCopy:copiedValues:. Do not use closureCopy:copiedValues: in code that you
write! Only the
compiler can decide to send the message closureCopy:copiedValues:."
```

```
^BlockClosure new outerContext: self startpc: pc + 2 numArgs: numArgs
copiedValues: anArray
```

BlockClosure methods for evaluating

value: anArg

```
"Activate the receiver, creating a closure activation (MethodContext)
whose closure is the receiver and whose caller is the sender of this message.
Supply the argument and copied values to the activation as its arguments and copied
temps."
```

```
| newContext sz |
numArgs ~= 1 ifTrue:
    [self numArgsError: 1].
newContext := self asContextWithSender: thisContext sender.
sz := copiedValues basicSize.
newContext stackp: sz + 1.
newContext at: 1 put: anArg.
sz > 0 ifTrue: "nil basicSize = 0"
    [1 to: copiedValues basicSize do:
        [:il newContext at: i + 1 put: (copiedValues at: i)]].
thisContext privSender: newContext
```

BlockClosure methods for private

asContextWithSender: aContext

"Inner private support method for evaluation. Do not use unless you know what you're doing."

```
^(MethodContext newForMethod: outerContext method)
  setSender: aContext
  receiver: outerContext receiver
  method: outerContext method
  closure: self
  startpc: startpc
```

and here's the decompilation of factorial compiled using the prototype compiler:

```
| t1 |
t1 := Array new: 1.
t1
  at: 1
  put: [:t2 | t2 = 1
    ifTrue: [1]
    ifFalse: [t2
      * ((t1 at: 1)
        value: t2 - 1)]]].
(1 to: 10)
  collect: (t1 at: 1)
```


and its opcodes:

```
pushLit: Array
pushConstant: 1
send: #new:
popIntoTemp: 0
pushTemp: 0
pushConstant: 1
pushThisContext
pushConstant: 1
pushLit: Array
pushTemp: 0
send: #braceWith:
send: #closureCopy:copiedValues:
jumpTo: L3
  pushTemp: 0
  pushConstant: 1
  send: #=
  jumpFalseTo: L1
  pushConstant: 1
  jumpTo: L2
L1:
  pushTemp: 0
  pushTemp: 1
  pushConstant: 1
  send: #at:
  pushTemp: 0
  pushConstant: 1
  send: #-
  send: #value:
  send: #*
L2:
  blockReturn
L3:
  send: #at:put:
  pop
  pushConstant: 1
  pushConstant: 10
  send: #to:
```

```
pushTemp: 0
pushConstant: 1
send: #at:
send: #collect:
pop
returnSelf
```

and it works 😊

Next post, the closure bytecodes, costs, etc.

 Send article as PDF	<input type="text" value="Enter email address"/>	<input type="button" value="Send"/>
---	--	-------------------------------------

Posted by admin on Saturday, June 7th, 2008, at 7:52 pm, and filed under [Cog](#).
Follow any responses to this entry with the [RSS 2.0](#) feed.
You can [post a comment](#).

{ 5 }

Comments

1. **Colin Curtin** | 09-Jun-08 at 9:48 am | [Permalink](#)

This is awesome, and the level of detail is perfect. Keep it up!

2. **Steven Swerling** | 09-Jun-08 at 2:35 pm | [Permalink](#)

Enjoying this very much, thanks.

3. **Anthony Lander** | 12-Jun-08 at 9:00 am | [Permalink](#)

Eliot,

Teaching as you are building is a wonderful idea, and we (the interested readers) are the lucky beneficiaries. Thank you for sharing with us.

-Anthony

I feel so privileged to have this fantastic context within which to work on Cog. Great blogging software, an affordable hosting service, an open source project to work on, a great product to contribute to and great colleagues. I feel like the Ferrari test driver I saw in a program on car design who, on getting out of a 456 he was testing to destruction doing doughnuts until a radiator hose broke, said "I am the Ferrari test driver. I have the best job in the world.". Later in the program you can guess what the Lamborghini test driver said 😊

Eliot

4. **Jecel Assumpção Jr** | 13-Jun-08 at 3:18 pm | [Permalink](#)

In the example that follows "This way round works as intended:" I think you meant to put the "n" before the rest of the expression as indicated in the paragraph after that. Right now the two versions are identical. It took me quite a while to convince myself of that 😊

Thanks Jecel! Good catch. I completely reformatted the code, going to generated HTML: instead of using <pre> and I screwed up second time around. So now this way round it works 😊

*The fact that I can edit my previous attempts is convenient.
Corrupting even.*

Eliot

5. **Mathueu Suen** | 16-Jun-08 at 6:51 am | [Permalink](#)

Hi Eliot

I am wondering why you do not use the Anthony Hannan's compiler. It seem to me that all the feature you want is inside.

I'm a slow learner and find picking up new things hard. I also know exactly what I want to do with Cog. So it is easier for me to use the existing compiler that I know very well. It took me two weeks to modify the compiler to do closures. It probably would have taken me that amount of time to get to grips with Anthony's compiler.

As far as I understand is because of optimization that need to be done at the compiler level. I would be very interest if you explain in detail why.

I hope I will in future posts. The issue is to do with how the Vm fools the image into thinking there are always contexts and only contexts when in fact the Vm uses a much more conventional and much faster stack organization.

One thing that can be done with the compiler of Anthony is rewriting the BytecodeGenerator so you can change the bytecode generated to create/access ClosureEnvironment.

Well I think I did the same thing, just on what for me is more familiar ground. No disrespect to Anthony's compiler. And I had fun doing it myself 😊

The post is very interesting and I am interest to see the post to come about context-to-stack mapping and the scope analysis 😊.

Thanks

Thank you! I'll try not to keep you waiting!

Eliot

Post a Comment

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

Name *

Email *

Website

Message

« [COG](#)

[BLUEBOOK COMPILEDMETHODS –
HAVING OUR CAKE AND EATING IT
TOO](#) »

