Clément Béra ~ Smalltalk, Tips 'n Tricks

Context and BlockClosure implementation

21 - *Wednesday* - *JAN 2015*

POSTED BY CLEMENT BERA IN COG

≈ **1 COMMENT**

Today I'm going to discuss about the internal representation and the implementation of Contexts and BlockClosures running on top of the Cog VM (that includes Pharo, Squeak, NewSpeak contexts and closures).

Introduction

First things first, what is a Context and what is a BlockClosure ?

A *Context* represents the state of a method activation. A Context is created when a method is activated, and is terminated when the method's execution is finished (the method has returned). This means if a method has been activated several times, several contexts exist (one per method activation).

In many languages, a method activation is defined as a stack frame. In Smalltalk, a context is different from a stack frame because it is manipulated as any object and on the contrary to stack frames, a Smalltalk context *can* edit its sender/caller (i.e. the context that activated it).

A *BlockClosure* is a reference to a function together with an environment. A BlockClosure is interesting because in addition to a method, it captures the environment in which it was created. By referencing this environment, it has two features that regular methods do not have:

- Access to non local variables: a block closure can access variables present in the environment it captured.
- Non local returns: if it wants to, a block closure can return to the sender of the environment it captured, instead of the sender of its activation.

Method's context

Methods are represented by CompiledMethod objects. Without going into details, a compiledMethod holds information about what code the virtual machine has to run in the form of bytecode instructions.

When a method is activated, a Context is used by the virtual machine to access the runtime state of the method. The virtual machine needs several information to be able to execute an instruction in the compiled method:

- *sender:* the sender references another Context, the one that activated this context.
- *pc:* pc stands for program counter. It can also be called sometimes ip for instruction pointer. The pc holds a number so the VM can know which bytecode instruction it is currently executing and which instruction is the next instruction to execute.
- *method:* we said that a context is a method activation. The method references the method that is activated by this context.
- *receiver:* when executing instructions such as 'self' or instance variable access, the VM needs a pointer towards the receiver object
- *arguments and temporary variables values:* All the values of the temporary variables and arguments are stored in the form of a stack. A Context has an instance variable, stackp, which represents the current depth of the stack, and a variable-sized zone to store all the values.

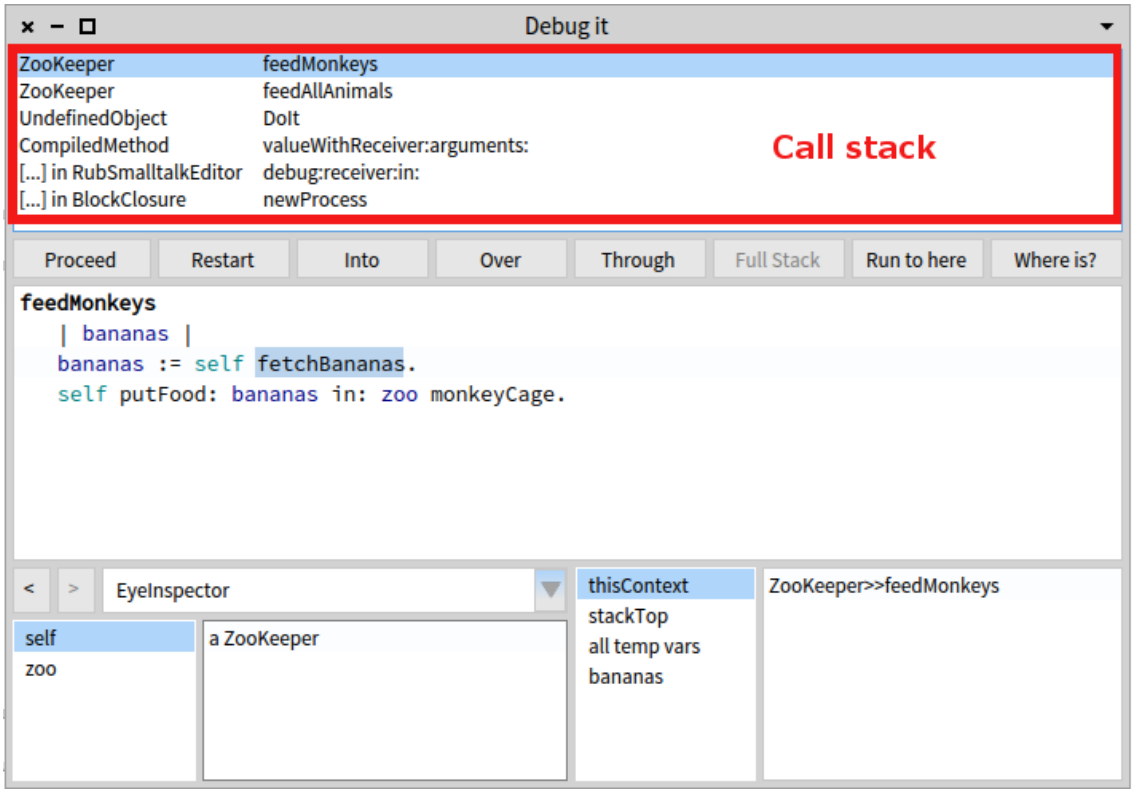
Example:

```
ZooKeeper>>feedAllAnimals
self feedSnowLeopard.
self feedLion.
"..."
self feedMonkeys.
"..."
self feedTiger.
self feedPanther.
```

ZooKeeper>>**feedMonkeys**
| bananas |
bananas := self fetchBananas.
self putFood: bananas in: zoo monkeyCage.

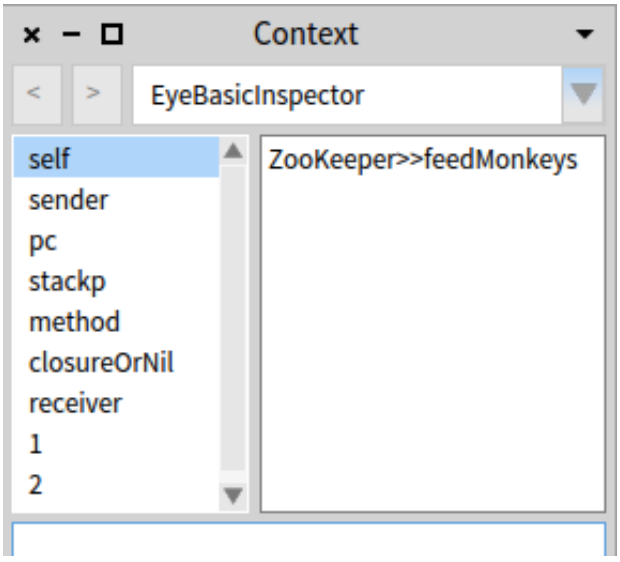
DoIt: ZooKeeper new feedAllAnimals

If you debug step by step the DoIt, you can see in the debugger the method `feedMonkeys` and its the call stack in the form of contexts.

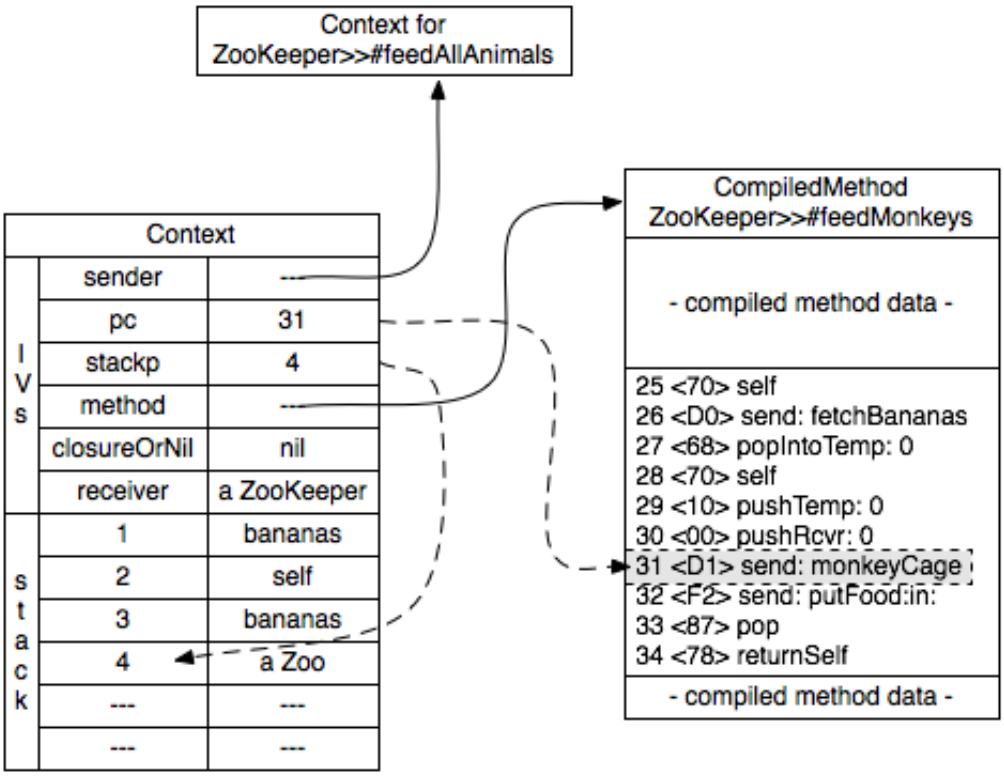


Each line in the debugger corresponds to a method activation (= a Context). Each context has a reference in its sender field to the next context in the list.

Let's *basic inspect* the top context.



As we can see in the inspector, the sender field references the context for ZooKeeper>>#feedAllAnimals that activated this context for ZooKeeper>>#feedMonkeys. The pc field represents the next bytecode instruction that will be executed, 'send monkeyCage'. The stackp field represents the current depth of the stack, which is 4 (the context can access 4 extra fields after its instance variables, represented in the basic inspector by the number 1 to 4). The method field references the compiled method for which this context was created. The closureOrNil field is always nil for method activations (we'll discuss closure activations later). The receiver field holds a pointer to the receiver.



Let's detail the stack zone (fields 1 to 4 in the figure). This zone has a variable size. For performance, the Cog VM preallocates room for the Context stack based on a flag in the Compiled method (it preallocates either 16 or 56 fields depending on the flag, see the largeFrame and SmallFrame class variable values of CompiledMethod). Thanks to stackp, the Context knows which fields it is allowed to access in its stack zone (other values on stack may not be safe).

In our case, the context can access 4 values on stack. For method activations, the stack is composed as follow:

- arguments values
- temporary variables values
- additional stack slots

The method studied, `ZooKeeper>>#feedMonkeys` has no arguments. It has however 1 temporary variable, bananas. This is why the first stack slots holds bananas, this is the value of the temporary variable (when executing bananas := self fetchBananas, the temporary was assigned to a collection of bananas. Before these instructions, it was nil).

The additional stack slots are there for runtime support. For example, when a message send is activated, the VM pushes on stack the receiver and the arguments of the message. In our case, we are about to send a nested message send. The elements on stack at position 2 and 3 are the receiver and first argument of the message send #putFood:in:, whereas the element at 4 on stack is the receiver of the #monkeyCage message.

Ok, we explained the basics, now let's move to the advanced cases: BlockClosure creation and activation.

BlockClosure creation

At creation time, a BlockClosure captures its enclosing environment. It captures:

- accesses to non local variables (non local temporary variables, non local arguments, enclosing environment receiver)
- direct access to the enclosing environment for non local return (If you don't know what's a non local return, please read the Block chapter in Deep into Pharo)
- access to the code that will be used to execute the closure (in the form of bytecode instructions

Example: (N.B.: This example is convenient as a showcase, it's not necessarily recommended code)

A zoo keeper plays with all the monkeys, and if there's an issue (a monkey is mad or he harms the zoo keeper while playing with him), he leaves the monkey cage without playing with the other monkey.

```
playWithMonkeysWith: toys
1  | issue |
2  issue := false.
3  zoo monkeyCage getMonkeys do: [ :monkey |
4    monkey isMad
5      ifFalse: [
6        self playWith: monkey with: toys.
7        self isHarmed ifTrue: [ issue := true ] ]
8      ifTrue: [ issue := true ].
9    issue ifTrue: [ ^ self leaveMonkeyCage ] ]
```

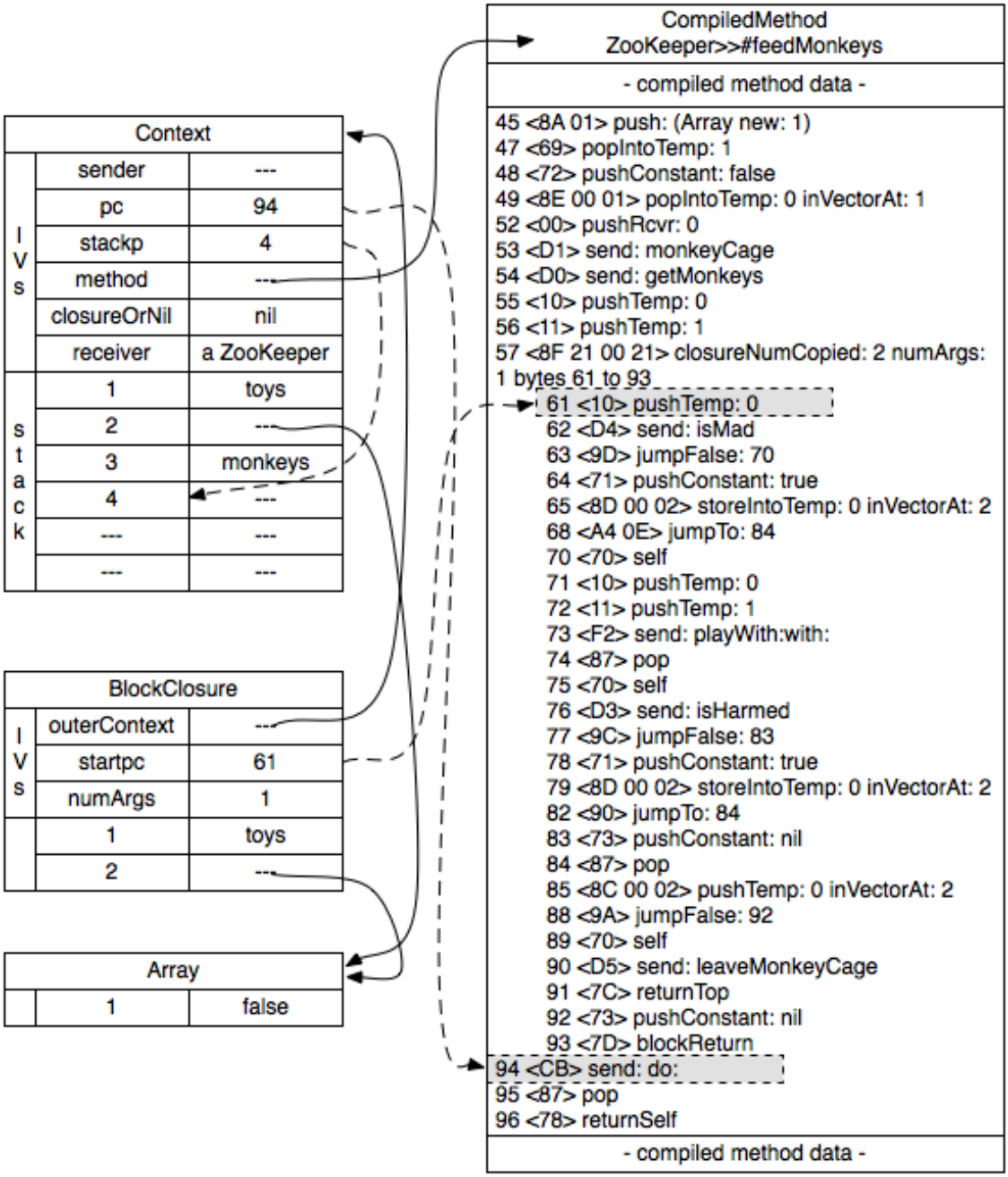
Let's look at the block created as an argument of the #do: message line 3,
`[ :monkey | "..." ]`.

This block has access to the variable `issue`, created in the method's context, and to the argument of the method `toys`. These variables are not defined in the block, so they're non local. This kind of variables typically do not exist in regular methods.

A block activation shares its receiver with the environment it captures. Therefore, code in a block closure can access self, as well as indirectly all the instance variables of self. In Pharo, the implementation allowing a block closure to reference self is done differently than other non local variables: the block closure captures the context where it was created and the captured context has a reference to self. When the blockClosure is activated, it copies the receiver from the captured context to its activation context receiver field. But conceptually the receiver is also a non local variable.

This block has also a non local return line 9: if there was an issue, the ZooKeeper leave the monkey cage without playing with the other monkeys.

Let's debug the BlockClosure and inspect it (step by step, then inspect the closure when it's on stack top, I'll show a schema because it's simpler to explain).



The BlockClosure is created with a certain number of copied variables, in our case, 2, `issue` and `toys`. This is why our BlockClosure has two extra variable fields.

The outerContext of the BlockClosure is the context that created it, it is the method activation record of `playWithMonkeysWith:`.

The method captured by the blockClosure is represented as bytecodes inlined in its enclosing method. Therefore, a blockClosure can access its bytecode by looking for its outerContext method and its bytecode starts at the program counter stored in its instance variable `startpc`. In the

figure, we indented the bytecodes of the blockClosure in the method. When the method is executed, after the block creation, it jumps over the block bytecodes. The block bytecodes are used only in the block closure activation.

The `numArgs` field of the BlockClosure holds 1, because the block has one argument `monkey`.

Now we have two copied variables, holding `toys` and `#(false)`. `toys` is the argument of the enclosing method. As the argument is not assigned in the block closure nor after the block creation, the value of toys will remain the same after the block creation. Therefore, a copy of the variable `toys` i written into the block closure, allowing the block closure to access it.

On the other hand, the variable `issue` is assigned in the blockClosure. As the variable is shared between the enclosing method and the closure, when `issue` is edited (i.e. the temporary variable is assigned), the variable needs to be edited in both place. To do that, the compiler automatically generates the creation of an array (See instruction 45, push (Array new: 1)). This array, allocated on heap, holds the variables that are shared between the closure and the method and that couldn't be passed as a copy because of the position of some assignments. We saw that the second copied value of the block was referencing a heap allocated array holding `false`.

Variables that are accessed through an indirection array are not accessed with the same bytecodes than regular variables. At instruction 55, we see a `pushTemp: 0`, which means it accesses the first value on the context stack, the argument of the method `toys`. At instruction 49, `popIntoTemp:0 inVectorAt: 1` is an access to a temporary in an indirection array. This bytecode means, access the first field of the array located at position 2 on stack.

Note: Arrays created automatically for variable shared between closures and methods are also called *tempVectors* or *vectors*.

Notes:
1) Typically, a blockClosure has access to only 1 temp vector that can have up to 255 variables and multiple copied temporary variables. However, in specific cases (multiple closures including nested closures and inlined closures), a blockClosure may have access to several temp vectors.

2) While debugging, the user can see in a blockClosure its temporaries including the copied temporaries. He can also edit their values. In this case, the debugger figures out that several fields need to be edited, the field in the blockClosure activation, the field in the blockClosure itself, the field in the enclosing activation, and change the value of the variable in all the references.

Non local returns

Yesterday I looked up the definition of a closure and I found this one:

In programming languages, a closure (also lexical closure or function closure) is a function or reference to a function together with a referencing environment — a table storing a reference to each of the non-local variables of that function. A closure — unlike a plain function pointer — enables a function to access those non-local variables even when invoked outside its immediate lexical scope.

The definition started well, until they described the referencing environment as “a table storing a reference to each of the non-local variables of that function”. Then the definition is not correct anymore because in some implementation the referencing environment is more than just a table. That’s the case in Smalltalk.

A blockClosure has a direct pointer to its outerContext, the context that created the blockClosure. With this variable, the blockClosure can access any temporary variables (even the ones it does not need), as well as perform non local return.

1) Why would the blockClosure want to access temporary variables it does not need ? One simple answer: to improve the debugger. In the debugger, the user can see the values of any temporary variables, disregarding if it’s a copied variable, indirect variable or unused variable (you don’t even have to understand such concept to debug your smalltalk code).

2) If the blockClosure can access the non local variables from the outerContext, why does it need to keep references to them in its variable fields ? For performance mainly. Using directly a context is difficult as the virtual machine maps the linked list of contexts to a C-like stack internally. Therefore one wants to limit the access to the contexts for performance.

3) What’s a non local return and do we care ?

Ok here’s the main point. A blockClosure can return either to its sender (see our example with false as argument) or to its homeContext sender (see our example with true as argument). This non local returns requires the virtual machine to walk up the stack until it finds the stack frame to return to. This can be done only using the outerContext field.

This outerContext field is therefore kept (as a blockClosure instance variable, set when the blockClosure is created) to be able to perform non local returns and to be able to debug a blockClosure seeing all the temporaries disregarding of their status with the blockClosure.

We care about non local returns because in smalltalk, conceptually, all control structures (conditions, loops) are messages sends with blockClosures as arguments. If we wouldn’t have non local returns, we wouldn’t be able to write a return in a branch or in a loop.

ex:
MyClass>>foo
self isPlague ifTrue: [^ self].
“some code…”

This method conceptually requires a non local return.

Failing non local returns

Non local returns can fail in two specific cases. We detail them here.

1) Non local return to dead home context

A dead context is a context that has been terminated, which means that its execution has finished (it has encountered a return).

If a block holds a non local return, the execution flow will returns to its home Context sender. However, it can happen that the block's home Context is already dead.

Example:

```
exampleBlockCannotReturn
self getBlock value

getBlock
^ [ ^ 42 ]
```

In this case, while executing the blockClosure (`value` message), the blockClosure outerContext, the activation of `getBlock`, is already dead. Therefore an exception is raised (BlockCannotReturn).

2) Sideway return

Sideway returns is one of the trickiest aspect of non local returns. It is not specified in the Smalltalk specifications, so it is the choice of the VM implementors to allow them or not. In the Cog VM, they are strictly forbidden.

A sideway return happens when a block performs a non local return, with it home Context alive but not on stack.

Example:

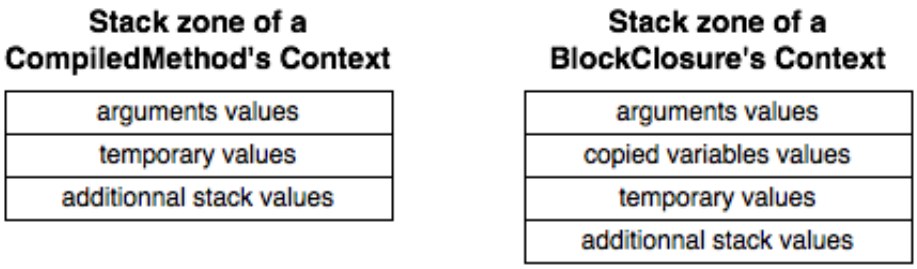
```
exampleSidewayReturn
[ ^ 42 ] forkAt: Processor activePriority + 1.
Processor yield.
```

Here, when the non local return of the block is performed, its outerContext (the activation of `exampleSidewayReturn`) is alive, but on another Process stack. This is a sideway return and also raises a BlockCannotReturn on the Cog VM.

Activating a BlockClosure

When a BlockClosure is activated, conceptually, a context is created as for method activations. There are 2 main differences in blockClosures activations:

- a reference to the blockClosure is held by the context in the field named `closureOrNil`. This field is nil for method activations, and references the closure in case of closure activation. This field is used, for example, to find out the home context of the blockClosure activation using the block closure outerContext.
- In the stack zone, we can find first the arguments of the block, then the copied variables of the block, then the temporary variables for the block before the additional slots (see figure below)



Performance details

When a blockClosure is created, up to three objects are created:

- the blockClosure to hold the outerContext, the references to non local variables, the number of arguments of the closure and the start pc to know what bytecode to execute while running the closure
- the outerContext (i.e. the mapping between the outer stack frame and a context object): this is not needed if the outerContext has already been created, for example by another closure creation
- the tempVector to store indirect temporaries

To improve performance, one has to reduce the number of objects created. One solution is to try not to have tempVectors, by rewriting the blockClosure differently to avoid the tempVector creation. Another solution is the adaptive optimization approach we are currently working on, which aims to inline the blockClosure in its homeContext, in order not to create any of these objects.

Some other smalltalks, such as VisualWork smalltalk, decided to keep the outerContext field of blockClosures only if a non local return is present in the closure. This makes it harder to debug, because the user cannot see from the blockClosure which activation created the closure nor unneeded temporary variables from the enclosing environment. This optimization would be tricky in Cog because Cog's blockClosure relies on the outerContext also to find out where is the method holding the bytecode to execute for a blockClosure (the blockClosure bytecode is inlined in the enclosing method bytecode).

thought on "Context and BlockClosure implementation"

1. Pingback: Smalltalk en vrac (6) | L'Endormitoire

1