Clément Béra ~ Smalltalk, Tips 'n Tricks

Introducing Immutability in the Cog VM

24 *Sunday* *Jan 2016*

Posted by Clement Bera in Cog, Spur

≈ **1 Comment**

Hi everyone,

I have not written on my blog for a long time and I am sorry about that. There is a good reason though. I have been working for a while on a book which teach to Smalltalk developers how to implement a simple object virtual machine. One part of the book, dealing with call stack management, has moved to a closed beta and a few testers are trying to do the tutorials. Hopefully this part of the book will be available in the mid-january to everyone (as open beta).

During the last month, I tried to work from time to time on introducing Immutability in the Cog VM. The Cog VM was finally compiled with full immutability support last wednesday. I did most of the work myself, though as always, **Eliot Miranda** was very helpful. This post discusses the implementation of immutability in the Cog VM.

Immutability design

As argued on the virtual machine mailing list, we are talking about a write-barrier more than immutability itself.

My work was directly inspired from the immutability experiment implemented on the Newspeak interpreter which was used a few years ago (now the Newspeak interpreter and the Smalltalk interpreter are merged in the Cog VM) and is somehow similar to VisualWork's immutability.

Two main new primitives are introduced. One (*isImmutable*) allows to check if an object is immutable. The other one (*setImmutabilityTo:*) makes an object mutable or immutable based on the boolean argument.

Once an object is set as immutable, any primitive attempting to change the object's state and instance variable stores fail.

Failing a primitive in case of immutability is easy to implement as the Cog VM already support primitive failures. The in-image primitive fall-back code may require to be edited to check if the primitive failed because of immutability and have a different behavior if so (for example raising a NoModification error). There is nothing much to say about that.

VM Callback

The main issue lied with instance variable store failure. The generic idea is to trigger a virtual machine callback, similarly to #doesNotUnderstand: , when an instance variable store is performed on an immutable object. The call-back notifies the programmer that the program is attempting to mutate an immutable object before the mutation has happen. Then, the program can do different things, such as raising an error or change the object to mutable state to perform the store.

VM callback are already supported, but none of them are similar to the new one we introduced for instance variable store failure, #attemptToAssign:withIndex: . This VM callback happens in an instance variable store instruction.

One problem is that the context's pc will be after the instance variable store when the call back happens, with a given stack state, and that given stack state does not expect anything to be pushed on stack before the next instruction. The call back #attemptToAssign:withIndex:, as any message send, should return a value that would be pushed on stack, but if that happens, the sender's stack would be messed up.

To solve this problem, I designed the #attemptToAssign:withIndex:: callback as a method returning no value. This can be done hacking the active process. Here is an example code of #cannotAssign:withIndex: :

```
attemptToAssign: value withIndex: index
"Called by the VM when assigning an instance variable of an immutable
object.
Upon return, executing will resume after the inst var assignment. If
the inst var had to be performed, do it manually here in the call back
with instVarAt:put: .
This method has to return *no* value. I do it by hacking the process
(as for sista callbacks) until we provide a better solution."
| process |
"self do something here if you want."
process := Processor activeProcess.
[ process suspendedContext: process suspendedContext sender ]
    forkAt: Processor activePriority + 1.
Processor yield.

"CAN'T REACH"
```

As you can see, the method returns no value as the sender does not expect any value to be returned. The sender could, for example, have an empty stack and expect the next instruction to be performed on an empty stack.

Another problem lies with the machine code version of the method. Because of the call-back, the execution can be interrupted on any instance variable store. I will discuss later in the article the implications.

Immutability exception

I wanted the immutability design to be simple to hit production as fast as possible. For this purpose, I needed to select objects that can't be immutable because they require a lot of additional work. In the future, I could change the VM to allow these objects to be immutable if someone can show me a production application that requires it. It is just harder to get these objects immutable, so I didn't do it.

I distinguish two kind of objects that can't be immutable for now:

- *Context*: Contexts represent method and closure activation records. There are difficult to make immutable as they are mapped to stack frames in the VM for performance.
- *Objects related to Process scheduling:* The virtual machine switches from a Process to another one from time to time while executing Smalltalk code. These switches implies that the *Processor* (global object) has to be mutated in the virtual machine. It can be tricky to execute an in-image callback when such objects are mutated as the VM is in the middle of a Process switch. For this purpose, I forbid Semaphores, the Processor, the linked list of Processes and the Processes themselves to be immutable. This decision was quite aggressive, but as I said before, I could consider letting more objects to be immutable if one shows me a good production application use-case.

Immutability in the memory manager

The first thing to do was to change the Memory Representation so each object keeps a specific bit to mark if they are immutable or not.

Fortunately, the Spur Memory Manager was designed with immutability in mind. In each object's header, a bit was already reserved for immutability. Most of the code related to this bit (is the object immutable, set the object mutability) was already implemented, and I've just had to extend and bug-test it. Once this was ready, I had just to integrate the use of the immutability bit checks in the rest of the VM.

I didn't implement immutability in the SqueakV3 Memory Manager. Immutability requires a Spur VM. We have not dropped support for this old Memory Manager, but there won't be any development there anymore. Everyone should migrate to the Spur Memory Manager at some point.

Immutability in the interpreter

As stated before, this work was directly inspired from the Newspeak interpreter experiment. I used the code of the experiment as a reference for my work.

Two main things were done:

- Instance variable stores: bytecode execution of instance variable stores were changed to check if the object mutated is immutable. If the object is indeed immutable, then the new callback I introduced in the interpreter (#attemptToAssign:withIndex:) is called.
- Primitives: I went carefully through each primitive and fail them if they mutate an immutable object, at the exception of objects that can't be immutable. In this case, a regular primitive failure happens though the error code may be different from other failures. I might have forgotten a few primitives, we'll see if I get some bug reports.

Immutability in the JIT

Here we come to the real deal. How to make the JIT immutability compliant ?

As for the interpreter, I started with the abstraction over the memory representation of objects. I added the generation of machine instruction checking if an object was immutable or not. This was fairly easy as one just needs to check a bit in the object's header.

Then I needed to change the primitives generated to machine code by the JIT to fail if they attempt to mutate an immutable object. There were only two primitives that needed to be changed: #at:put: and #stringAt:put: . Other primitives mutating objects are not compiled to machine code by the JIT. The change consisted in checking if the object mutated was immutable at the beginning of the primitive, in which case, the execution fallbacks to the C version of the primitive.

The second thing to do was to change the instance variable stores to check if the receiver was immutable. This was quite tricky.

The first problem there was that instance variable stores did not flush the register state. Registers used to be able to be live across an instance variable store. This does not work any more as an instance variable store can now trigger an in-image callback (basically a message send) which requires the registers to be flushed.

Unfortunately, there are no good solutions around it. Either the JIT flushes all the register and they can't be live across the store, generating lots of spills on the main execution path, or the JIT restores the register state after the callback, which implies the generation of many more machine instructions per instance variable store, which can be even slower than flushing the register state.

I decided to flush every register at each instance variable store but the one holding the receiver. Hence, only the receiver register needs to be restored after the callback (one extra machine instruction). This allows the most common register, the one holding the receiver, to be live across instance variable stores.

The second problem lied with the callback. I needed to add a trampoline from machine code to the C runtime to trigger the #attemptToAssign:withIndex: callback. By convention, trampolines in the Cog always pass all their arguments by register. Instance variable stores used to put the value to store in any register available. That was not possible any more, as the trampoline needs a fixed

register. I changed all the code related to the stores to fix the object mutated in a register (%EDX in x86) and the argument in another one (%ECX in x86). Then I added the trampoline, expecting the values in specific registers.

Lastly, each instance variable store instructions now needs to be mapped, i.e., the JIT needs to remember how to map the machine code program counter (mcpc) to their bytecode program counter (bcpc). This is important as, for example, the debugger can be opened in the #attemptToAssign:withIndex: callback requiring the context failing to perform the instance variable store to be displayed correctly. It can also be required for other operations, such as the divorce of all the stack page's frames if the stack page needs to be freed.

This last point was easy to do in the NewspeakV4 and the SistaV1 bytecode set. However (and *unfortunately*), the default bytecode set of Squeak and Pharo is still SqueakV3PlusClosures. This bytecode set is always problematic. The main problem I had was that all the extended stores (stores over 16 encoding such as pushTemp: 20) are compiled in a single extendedStore instruction or with the DoubleExentedDoAnythingBytecode. This implies that if I map the bcpc to mcpc of the extended store instruction, all the store temp would be mapped too. Not only it implies mapping lots of instructions, but it also implies the mapping of the extendedStore instruction used specifically in case of primitive error code and compiled differently in the JIT… And I am not even talking about the DoubleExtendedDoAnythingBytecode, which is the craziest bytecode of this set.

Well, this was messy, but Eliot and I built some work around and it's up and running by now. I hope soon some clients as Pharo will use by default the SistaV1 bytecode set, it will solve many issues (such as objects with over 256 instance variables, long jumps and many more) and simplify the VM work.

Conclusion

Well, now that immutability is up and running, I hope some of the Cog clients will use it. To try it, compile a VM with IMMUTABILITY set to true (it's a C Compiler setting) and try to load the package here in one of the Smalltalk client. For the newspeak support, I guess one has to port it form the Smalltalk version. I have also started to discuss how to integrate Immutability in Pharo here. If someone is interested in making compiled method literals immutable, I will be happy to help.

thought on "Introducing Immutability in the Cog VM"

1. Pingback: Smalltalk en vrac (22) | L'Endormitoire

Create a free website or blog at WordPress.com.

1