Clément Béra ~ Smalltalk, Tips 'n Tricks

Editing Cog's JIT compiler

04 Wednesday Feb 2015

Posted by Clement Bera in Cog

≈ **4 Comments**

Today I'm going to discuss about editing Cog's JIT in order to increase the number of primitives which can be generated to machine code or to add new bytecodes and add their translation to machine code.

All these changes are done on an image with the VMMaker packages loaded. To have such an image, you can choose between two solutions:

- **The Pharo branch:** Clone the github repository and follow the instructions on the Readme.md. As of today, You have to run this script:
```
git clone --depth=1 https://github.com/pharo-project/pharo-vm.git
cd pharo-vm
cd image && ./newImage.sh
```
- **The Cog branch:** checkout the svn repository and follow the instructions on this page. As of today, you have to run this script:
```
svn co http://www.squeakvm.org/svn/squeak/branches/Cog/image
cd ./image
./buildsqueaktrunkvmmakerimage.sh
```

You now have your Pharo or Squeak VMMaker image to work with. The VMMaker trunk is interpreter-based only, so it is not discussed in this post.

To compile the VM from slang to C, then C to executable, the easiest way is to create an Ubuntu 32 bits VM on VirtualBox (or VMware). Then you need to run this script:

```
sudo apt-get install cmake zip bash-completion ruby git xz-utils debhelper
devscripts
sudo apt-get install libc6-dev:i386 libasound2:i386 libasound2-dev:i386
libasound2-plugins:i386 libssl-dev:i386 libssl0.9.8:i386 libfreetype6-dev:i386
libx11-dev:i386 libsm-dev:i386 libice-dev:i386
sudo apt-get install build-essential gcc-multilib g++
sudo apt-get install libgl1-mesa-dev libgl1-mesa-glx:i386
sudo apt-get install binutils:i386 g++-4.6:i386 gcc-4.6:i386
sudo ln -s /usr/lib/i386-linux-gnu/mesa/libGL.so /usr/lib/i386-linux-
```

```
gnu/libGL.so
sudo ln -s /usr/lib/i386-linux-gnu/mesa/libGL.so.1 /usr/lib/i386-linux-
gnu/mesa/libGL.so
sudo apt-get install uuid-dev:i386
sudo apt-get install subversion
```

uuid-dev is mandatory only in the Cog branch. svn is only used to checkout the Cog branch.

The reference documentation is for the Pharo branch the readme.md on the github repository, whereas it is the HowToBuild file in one of the build.* folder on the Cog branch.

To build the VM, you need first to generate the sources out of the VMMaker image, then compile the C files with gcc (it does not work with llvm / clang, it does not work either with gcc 4.8 nd more recent for me).

Generating the sources:

In the Pharo branch, run as a Do It in the image `PharoVMBuilder buildUnix32`. This code presets the build with the V3 memory manager, the V3PlusClosure bytecode set, the StackToRegisterMappingCogit JIT, no additionnal options and a set of plugins.

In the Cog branch, run as a Do It in the image `VMMaker generateConfiguration` and pick the settings you need for your build.

Compiling the C code:

In the Pharo branch, go to the build folder and run $ ./build.sh

In the Cog branch, go to the directory you need for your build. For example, if you need a 32bits intel linux VM with the V3 memory manager and StackToRegisterMappingCogit, go to build.linux32x86/squeak.cog.v3/build.itimerheartbeat (Ubuntu requires the itimerheartbeat build). Then run $ ./mvm

Results:

In the Pharo branch, the compiled VM is in the folder named results.

In the Cog branch, the compiled VM is in the current repository (for example, build.linux32x86/squeak.cog.v3/build.itimerheartbeat)

Introduction to the Cog's JIT classes

All of the JIT's classes are in the package named VMMaker-JIT.

Cog's JIT is mainly split into two part, the one that parses the bytecode and uses a DSL to generate machine code, and the one that from the DSL generates machine code instructions. This split is here to separate platform-dependent code from the rest.

For the machine code generation part, we have these classes:

```
CogAbstractInstruction
  CogARMCompiler
  CogIA32Compiler
```

Basically one plugs into the JIT a CogAbstractInstruction concrete subclass. CogARMCompiler generates ARMv5 instructions (at the exception of pld, an ARMv6 instruction) whereas CogIA32Compiler generates Intel x86 32 bits instructions. There is another one in development for Intel x86_64 64 bits instructions, which might be there when you will look at it (CogIA64Compiler ?).

Abstract instructions are used for two purposes:

- As the backend of the bytecode to abstract instruction part of the JIT
- As a DSL to generate machine code routines such as trampolines (a trampoline is a routine allowing to switch from the native runtime to the interpreter runtime)

I am not going to talk about these aspects. Basically you need to read the Intel / ARM manual and implements the correct binding to these instructions. One may also add an optimizing backend to generate optimized instructions there, but the current backend, even if it is quite simple, does a pretty good job for now.

The interesting part for this post is the bytecode to abstraction instructions part. Let's look at it:

```
Cogit
  SimpleStackBasedCogit
    StackToRegisterMappingCogit
      SistaStackToRegisterMappingCogit
```

Cogit is the abstract superclass of the JIT classes. There are three subclasses:

- **SimpleStackBasedCogit:** This JIT just generates machine code that does the same thing as the bytecode. For instance, if the bytecode says push this value on stack, it'll push the value on stack. Its performance boost comes from the removal of the bytecode decoding overhead and to the addition of advanced inline cache techniques to speed up message sends.
- **StackToRegisterMappingCogit:** In addition to the superclass, this JIT maps stack values to registers if possible. For instance, if the bytecode says push this value on stack, but the value is used only before the next interrupt point, it will put the value into a register instead of on stack. In addition, it generates quick paths for SmallIntegers in branches if the branch is just after a comparison. It also does some tricks such as passing the receiver and arguments of a message by registers and not on stack for primitive calls.
- **StackToRegisterMappingCogit:** This JIT, in addition to this superclass, adds for marked methods counters in machine code in order to detect methods frequently used. It also features extra machine code routines to trigger code when a method frequently used is detected or when an uncommon case happen in an optimized method, as well as an introspection primitive so compiled method can know their counters values as well as their inline caches values.

Before adding code in the JIT, pick the Cogit version you want to use. I recommend StackToRegisterMappingCogit if you're not working on the Sista.

Adding your new bytecode / new primitive in the JIT compilation process

Now that you've chosen which JIT you want to edit, let's change the way it compiles your bytecodes or your primitives.

Editing the way the JIT compiles the bytecode

Class-side, you have a method that describes your bytecode set for the JIT. Here is an example with the SqueakV3PlusClosure bytecode set:

```
initializeBytecodeTableForSqueakV3PlusClosures
"some code"
self generatorTableFrom: #(
(1 0 15 genPushReceiverVariableBytecode needsFrameNever: 1)
(1 16 31 genPushTemporaryVariableBytecode needsFrameIfMod16GENumArgs: 1)
(1 32 63 genPushLiteralConstantBytecode needsFrameNever: 1)
(1 64 95 genPushLiteralVariableBytecode needsFrameNever: 1)
(1 96 103 genStoreAndPopReceiverVariableBytecode needsFrameNever: -1)
(1 104 111 genStoreAndPopTemporaryVariableBytecode)
(1 112 112 genPushReceiverBytecode needsFrameNever: 1)
(1 113 113 genPushConstantTrueBytecode needsFrameNever: 1)
(1 114 114 genPushConstantFalseBytecode needsFrameNever: 1)
(1 115 115 genPushConstantNilBytecode needsFrameNever: 1)
(1 116 119 genPushQuickIntegerConstantBytecode needsFrameNever: 1)
(1 120 120 genReturnReceiver return needsFrameIfInBlock: isMappedInBlock 0)
(1 121 121 genReturnTrue return needsFrameIfInBlock: isMappedInBlock 0)
(1 122 122 genReturnFalse return needsFrameIfInBlock: isMappedInBlock 0)
(1 123 123 genReturnNil return needsFrameIfInBlock: isMappedInBlock 0)
(1 124 124 genReturnTopFromMethod return needsFrameIfInBlock: isMappedInBlock
-1)
(1 125 125 genReturnTopFromBlock return needsFrameNever: -1)
(1 126 127 unknownBytecode)
(2 128 128 extendedPushBytecode needsFrameNever: 1)
(2 129 129 extendedStoreBytecode)
(2 130 130 extendedStoreAndPopBytecode)
(2 131 131 genExtendedSendBytecode isMapped)
"some code"
```

Let's look at what the description means:

```
(1 0 15 genPushReceiverVariableBytecode needsFrameNever: 1)
```

- The first 1 means these instructions are encoded in a single byte.
- The 0 15 means that bytecodes from 0 to 15 needs are described by this line. (Bytecode 0 to 15

are pushReceiverVariable, see the bytecode table).
- genPushReceiverVariableBytecode is the selector of the method that needs to be called to generate machine code when this bytecode is encountered.
- needsFrameNever: 1 is one of the additional bytecode description you can have. This one means, in any case, this bytecode does not require to create a stack frame for the method (Methods with only bytecodes like that, such as setter, do not require to build a stack frame which speeds up their execution). Another additional bytecode description is isMapped, which keeps a mapping from the machine code pc to the bytecode pc (used for debugging for example)

In the V3PlusClosure bytecode set, we can see that bytecode 126 and 127 are not used. So if you want to add another bytecode and compile in the JIT, this is the place to do it. Other bytecode sets, such as the SistaV1, have many more free slots.

Editing the way the JIT compiles primitive operations

There's a similar class side method that describe how the JIT compiles primitives based on the primitive number.

```
initializePrimitiveTableForSqueak
"come code"
self table: primitiveTable from:
#( "Integer Primitives (0-19)"
(1 genPrimitiveAdd 1 mclassIsSmallInteger:)
(2 genPrimitiveSubtract 1 mclassIsSmallInteger:)
(3 genPrimitiveLessThan 1 mclassIsSmallInteger:)
(4 genPrimitiveGreaterThan 1 mclassIsSmallInteger:)
(5 genPrimitiveLessOrEqual 1 mclassIsSmallInteger:)
(6 genPrimitiveGreaterOrEqual 1 mclassIsSmallInteger:)
(7 genPrimitiveEqual 1 mclassIsSmallInteger:)
(8 genPrimitiveNotEqual 1 mclassIsSmallInteger:)
(9 genPrimitiveMultiply 1 processorHasMultiplyAndMClassIsSmallInteger:)
(10 genPrimitiveDivide 1 processorHasDivQuoRemAndMClassIsSmallInteger:)
"some code"
```

The description for a line is as follow:

```
( 2 genPrimitiveSubtract 1 mclassIsSmallInteger:)
```

- 2 means this is the primitive number 2
- genPrimitiveSubtract is the selector of the method to call to generate code for this primitive operation
- 1 means this primitive has one argument
- As for bytecode, there are additional options, such as mclassIsSmallInteger:, which ensures that the class in which the primitive is implemented is SmallInteger (Else the primitive would always fail, that can be checked statically no need to postpone it to runtime)

Writing one's own machine code generation method

Let's say you've changed the bytecode table. You've added bytecode 126 which adds one to the top element on stack if it's a smallinteger, else does nothing, and push the result on stack. You changed in the bytecode set description

```
(1 126 126 genPlusOne)
(1 127 127 unknownBytecode)
```

=> 1 because it's a single byte instruction, encoded in byte 126, and we want to use genPlusOne for code generation.

You've reinitialized the bytecode description `StackToRegisterMappingCogit initializeBytecodeTableForSqueakV3PlusClosures`.

Now the method genPlusOne will be called when the JIT wants to compile bytecode 126. To implement genPlusOne in StackToRegisterMappingCogit, you need to use abstract machine instructions. Let's look at them a bit.

The abstract instructions available are in the protocol "abstract instructions" of the class Cogit. Their names always start with an uppercase character, so you can know while writing code generation code which methods actually do generate machine code and which methods are evaluated at compile-time. The abstract instruction names may look a bit strange. Let's describe some of them.

Let's look for example at the Add instruction (addition). You have four available instructions:

```
AddCq:R:
AddCw:R:
AddR:R:
AddRd:Rd:
```

The beginning of the method name, Add, is used to specify what the instruction does. Here it's a Add instruction, which adds two numbers.

Then each argument is prefixed by a one or two letters describing the type of the argument. Here's the mapping for the instructions we are discussing:

- Cq -> constant
- Cw -> address where to read the value
- R -> general purpose register
- Rd -> SSE2 registers (XMM registers)

Therefore, in our case, the instructions means (I'll say int, which means int32 in 32 bits processor and int64 in 64 bits processors):

- AddCq:R: -> Adds an int constant (argument 1) to an int in a register (argument 2)
- AddCw:R: -> Adds an int at a memory location in argument 1 to an int in a register (argument 2)

- AddR:R: -> Adds an int in a register (argument 1) to an int in another register (argument 2)
- AddRd:Rd: -> Adds a double in XMM register (argument 1) to a double in another XMM register (argument 2)

Did you get that ? Well basically you've understood most of the DSL. There are a few more description letters for memory read and write, such as:

```
MoveXbr: indexReg R: baseReg R: destReg
MoveXwr: indexReg R: baseReg R: destReg
```

These two instructions are used to read a byte (Xbr) or a word (Xwr) at the address in register baseReg shifted by an index in register indexReg and write the result in register destReg.

Ok, now that you got the DSL, let's implement genPlusOne !

So the specification tells us this bytecode adds one to the top of stack if it's a SmallInteger, else does nothing.

Here's the method you want:

```
genPlusOne
<var: #jumpNotSmallInt type: #'AbstractInstruction *'>
| jumpNotSmallInt rcvrReg |
self ssTop type = SSRegister
  ifTrue: [rcvrReg := self ssTop register]
  ifFalse:
    [(rcvrReg := backEnd availableRegisterOrNilFor: self liveRegisters) ifNil:
      [self ssAllocateRequiredReg:
        (rcvrReg := optStatus isReceiverResultRegLive
          ifTrue: [Arg0Reg]
          ifFalse: [ReceiverResultReg])]].
self ssTop popToReg: rcvrReg.
self ssPop: 1.
jumpNotSmallInt := objectRepresentation genJumpNotSmallIntegerInScratchReg:
rcvrReg.
objectRepresentation genConvertSmallIntegerToIntegerInReg: rcvrReg.
self AddCq: 1 R: rcvrReg.
objectRepresentation genConvertIntegerToSmallIntegerInReg: rcvrReg.
jumpNotSmallInt jmpTarget: self Label.
self ssPushRegister: rcvrReg.
^ 0
```

Let's discuss it.

The pragma is needed only for C code generation (the simulator does not need it). It is used to enforce the type of the variable jumpNotSmallInt to be an AbstractInstruction*. Usually you need very few types when writing routines in the JIT. You'll get use to the few types very quickly, just look at how normal bytecodes and primitives are generated to understand which types are available and which ones you need.

The first lines, until ssPop:, are used to either get the receiver of plusOne in a register. To use registers instead of stack values, StackToRegisterMappingCogit uses a simulated stack. ssTop answers the top of this simulated stack. If it's already a register, then we've got the receiver at hand, else we allocate a register to hold the receiver.

The next line "self ssTop popToReg: rcvrReg" moves the top of the simulated stack in the rcvrReg we've allocated. popToReg: is lazy, so if the value is already in that register, it won't generate anything.

We then pop the simulated stack by one because the operation consumes the operands.

Then, we need to branch to manage the case where the receiver is a SmallInteger and when it's not. This can't be done directly by the JIT, because it depends on the object representation. The JIT delegates to the object representation for that feature. Here are available object representation (Non final classes are abstract):

```
CogObjectRepresentation
  CogObjectRepresentationForSpur
    CogObjectRepresentationFor32BitSpur
    CogObjectRepresentationFor64BitSpur
  CogObjectRepresentationForSqueakV3
```

In our case, we dispatched on the SqueakV3 object representation.

Then we want to add one. Unfortunately, the value in the register is a SmallInteger, which means it's a pointer encoding directly in its value the integer value. Again, the way the integer is encoded as a pointer depends on the object representation. For a simple implementation, we therefore ask the object representation to convert the SmallInteger to an integer, add 1, then convert back to a smallInteger.

We then push the register holding the result on stack.

Lastly, we return 0 to tell the JIT that everything went fine while compiling this bytecode.

Validating your code

Ok, you're now generating your own code. Is it correct ? The first thing to do is to check the machine code generated for a method holding the code you edited.

Let's use that method:

```
examplePlusOne
^ 4 plusOne
```

We assume, as you added the bytecode, that you have a syntax element or something generating the bytecode 126. We assume in this case that the bytecode compiler generates bytecode 126 instead of a message send when it encounters the selector #plusOne.

Here's the bytecode generated:

21 pushConstant: 4
22 plusOneOperation
23 returnTop

Then we open a Transcript and run:
`StackToRegisterMappingCogit genAndDis: MyClass>>#examplePlusOne`

In the Transcript we got the Intel x86 machine code version of the method:

```
1000
objhdr: 1003
nArgs: 0 type: 2
blksiz: 78
method: 100040
mthhdr: 1001
selctr: 100048=#examplePlusOne
blkentry: 0
stackCheckOffset: 59/1059
cmRefersToYoung: no
00001018: xorl %edx, %edx : 31 D2
0000101a: call .+0xfffff5e9 (0x00000608=ceMethodAbort0Args) : E8 E9 F5 FF FF
0000101f: nop : 90
entry:
00001020: movl %edx, %eax : 89 D0
00001022: andl $0x00000001, %eax : 83 E0 01
00001025: jnz .+0x0000000f (0x00001036) : 75 0F
00001027: movl %ds:(%edx), %eax : 8B 02
00001029: andl $0x0001f000, %eax : 25 00 F0 01 00
0000102e: jnz .+0x00000006 (0x00001036) : 75 06
00001030: movl %ds:0xfffffffc(%edx), %eax : 8B 42 FC
00001033: andl $0xfffffffc, %eax : 83 E0 FC
00001036: cmpl %ecx, %eax : 39 C8
00001038: jnz .+0xffffffe0 (0x0000101a) : 75 E0
noCheckEntry:
0000103a: movl %ss:(%esp), %eax : 8B 04 24
0000103d: movl %edx, %ss:(%esp) : 89 14 24
00001040: pushl %eax : 50
```

```
00001041: pushl %ebp : 55
00001042: movl %esp, %ebp : 89 E5
00001044: pushl $0x00001000 : 68 00 10 00 00
IsAbsPCReference:
00001049: movl $0x00100000=nil, %ebx : BB 00 00 10 00
0000104e: pushl %ebx : 53
0000104f: pushl %edx : 52
00001050: movl %ds:0x80000='stackLimit', %eax : A1 00 00 08 00
00001055: cmpl %eax, %esp : 39 C4
00001057: jb .+0xffffffbf (0x00001018) : 72 BF
HasBytecodePC bc 20/21:
>>> 00001059: movl $0x00000009, %eax : B8 09 00 00 00
>>> 0000105e: andl $0x00000001, %eax : 83 E0 01
>>> 00001061: jz .+0x0000000a (0x0000106d) : 74 0A
>>> 00001063: sarl $1, %eax : D1 F8
>>> 00001065: addl $0x00000001, %eax : 83 C0 01
>>> 00001068: shll $1, %eax : D1 E0
>>> 0000106a: addl $0x00000001, %eax : 83 C0 01
0000106d: movl %eax, %edx : 89 C2
0000106f: movl %ebp, %esp : 89 EC
00001071: popl %ebp : 5D
00001072: ret $0x0004 : C2 04 00
startpc: 20
16r1049 IsAbsPCReference (16r1077)
16r1059 HasBytecodePC (16r1076, bc: 20)
```

I highlighted with '>>>' the machine code generated for the plusOne instruction. The machine code is shown as follow: instruction address, ':' , instruction name, operands separatedBy: ',' , opcodes for the instructions.

In our case, the receiver is 4, a constant. This value, once encoded in a pointer, is `0x00000009`, because SmallInteger are tagged with a 1 as the last bit.

The next two instructions corresponds to the genJumpNotSmallIntegerInScratchReg:. It ends with jz, jumpZero instruction.

The next four instruction corresponds to the SmallInteger untagging, the addition of 1, the SmallInteger tagging.

The instruction right after the code generated by genPlusOne is the code of the returnTop instruction. The JIT knew that our result was in register %eax because we pushed the result on the simulated stack, therefore it moved %eax to %edx, which is the register used to store the result of a message send (based on the calling convention in Cog on Intel x86).

Next steps

So you've implemented your new method in the JIT and you've checked that the generated code for an example method using your code looks correct. Great.

I assume that you've already added the code for your new bytecode / new primitive in the StackInterpreter.

The two next steps are to try to run your example method in the Cog VM simulator and for real.

Typically, if the code you added is complex, you start up the simulator with a script looking like that:

```
cos := CogVMSimulator newWithOptions: #(Cogit StackToRegisterMappingCogit).
cos desiredNumStackPages: 8.
cos openOn: 'Squeak-4.5-All-in-One.app/Contents/Resources/Squeak4.5-13680.image'.
false ifTrue:
[cos systemAttributes
at: 2 put: '-doit';
at: 3 put: 'ShootoutTests runAllToTranscript. Smalltalk quitPrimitive'].
cos openAsMorph; run
```

In the script, you may put in the 'doit' parameter some code to run in the simulator some code which uses your example method and ensure that everything is allright (for example, that it answers 5 in our case for our example method).

If your code is not that complex and that based on the machine code version of the example methods you compiled you're 100% sure your code is correct, then skip the simulator.

Lastly, you can compile your VM, and enjoy your new feature, as explained at the beginning of the post.

If you get a compile time error, most probably you've misstyped some variables in a pragma (for instance I always put AbstractInstruction instead of AbstractInstruction* …).

If you get a segmentation fault at runtime, it means your code were not correct in some cases. Then, the best thing to do is to recreate an image with the segmentation fault triggered at start-up and debug it from the VM simulator.

I hope you enjoyed the post. It is difficult to explain how to write code in the JIT because it implies lots of knowledge on the object representation, on assembly code and on stack / register machines but I hope you learnt something out of the post🙂 .

thoughts on "Editing Cog's JIT compiler"

1. *said:***Stéphane Ducasse**

February 4, 2015 at 7:02 pm

super je dois finir de lire ton autre blog. Argh je dois bosser en VisualWorks 😉

Stef

REPLY

2. Pingback: Smalltalk en vrac (7) | L'Endormitoire

3. *said:*Ben Coman

   December 15, 2015 at 11:31 am

   Nice read. Thanks Clement.
   btw, this heading is duplicated… StackToRegisterMappingCogit

   REPLY

   - *said:***Clement Bera**

     December 15, 2015 at 12:05 pm

     I am not sure I understand. What do you mean this heading is duplicated ?

     REPLY

Blog at WordPress.com.

4