Clément Béra ~ Smalltalk, Tips 'n Tricks

Simulating the Cog VM

30 - *Monday* - *May 2016*

Posted by Clement Bera in Cog

≈ **8 Comments**

Hi,

It's been a long time I wanted to discuss about the Cog VM Simulation infrastructure, which is critical for VM development. Recently I tried to help Ben Coman getting familiar with the VM simulator, and I thought I could compile down various things I wrote on the mailing list.

The Cog VM inherits directly from the Squeak VM wirtten by Dan Ingalls and detailled in this paper. Dan's idea was to provide interactive development for the VM -basically the Smalltalk debugger- while still compiling the production VM to machine code through C to keep the C advantages: platform-independency, the optimizing compiler, interaction with other C libraries.

To do so, the VM has two runtimes:

- the simulation runtime, used for debugging, that runs inside the Smalltalk runtime.
- the production runtime, used for production, that runs on major platforms.

The simulator is mainly used to debug:

- the JIT compiler
- the garbage collector
- the interpreter
- the VM plugins

I am not going to talk about VM plugin debugging in this post.

I. Getting a development environment

1. Development image

To simulate the VM one needs a development image. Two images are available:

- a Squeak image, instructions here.
- a Pharo image, instructions here.

The Squeak VM development image is provided with the scripts we will use for simulation, the Pharo image is not, so I will write down the scripts in the blog post for Pharo users. The scripts in the Squeak image are the reference, one needs to run them and not the one in the blog post as they may run out-of-date. For this reason, I recommend using the the Squeak image for VM simulation. For all the scripts discussed in the post, I detail where they can be found on the Squeak development image.

2. Development VM

One needs a VM with the processor simulator plugins installed. The VM provided on the Cog blog download page have the required plugins by default. The Pharo VM does not, so I'd recommend to use the Cog blog VM.

3. REPL image

To ease simulation, one may want to simulate a REPL (Read Eval Print Loop) image. The REPL image, if started from command-line, starts like an interactive commandline tool waiting for instructions. The REPL image understands chunk format, so one can write '3 + 4 !' to get a result. Once the result is displayed, the image waits for the next instructions to run.

If one uses a Squeak development image, on the image folder should be a script named *buildspurreaderimage.sh* that creates the desired image. On the Pharo side, as of today there are no convenience to do that, so one needs to build a similar image by oneself.

II. Simulating the Stack VM

The Stack VM is the VM *without* the JIT compiler. That VM is simulated quicker that the Cog VM. If one wants to debug the memory management, no need to simulate the full Cog VM, the Stack VM simulation is enough.

The script for Stack VM simulation are provided (for Squeak users) in the *VM simulation workspace*.

Here is, as of today, the script I use. Note that the script in the *VM simulation workspace* are the reference, these scripts may run out of date:

```
| cos |
cos := StackInterpreterSimulator newWithOptions: #(ObjectMemory
Spur32BitMemoryManager).
cos desiredNumStackPages: 8.
cos openOn: 'spurReader.image'.
cos openAsMorph; run
```

The simulator should open. After a dozen of second, maybe more according to your computer, the simulator should show the display screen of the image simulated.

There are multiple settings. The most important setting is *ObjectMemory*, where one can set one of the spur memory manager *Spur32BitMemoryManager* or *Spur64BitMemoryManager*, or the old V3 memory manager *NewObjectMemory*.

III. Simulating the Cog VM

The Cog VM is the production VM. That VM requires the processor simulator plugins to simulate correctly machine code.

The script for Cog VM simulation are provided (for Squeak users) in the *VM simulation workspace*.

Here is, as of today, the script I use. Note that the script in the *VM simulation workspace* are the reference, these scripts may run out of date:

```
| cos |
cos := CogVMSimulator newWithOptions: #(
Cogit StackToRegisterMappingCogit
ObjectMemory Spur32BitCoMemoryManager
ISA IA32).
cos desiredNumStackPages: 8.
cos openOn: 'spurReader.image'.
cos openAsMorph; run
```

The simulator should open. After a dozen of second, maybe more according to your computer, the simulator should show the display screen of the image simulated.

There are multiple settings. The most important settings are:

- *ObjectMemory*, where one can set one of the spur memory manager *Spur32BitCoMemoryManager* or *Spur64BitCoMemoryManager*, or the old V3 memory manager *NewCoObjectMemory*. Note the *Co* in the names (these are not the same memory managers that the StackVMSimulator).
- *Cogit*, where one can set one of the available JIT. In most case one wants *StackToRegisterMappingCogit* which is the production version.
- *ISA*, where one can specify the machine code use. The supported ISA are: *ARMv5*, *IA32*, *MIPSEL* and *X64*. Hopefully *ARMv8* will be available in the near future, but it's not the case right now.

IV. Debugging the JIT

To debug the JIT, one can directly compile to machine code a method in the image. The resulting machine code is shown on transcript.

The scripts for in-image compilation are available (for Squeak users) in the *In-image compilation workspace*.

Here is, as of today, the script I use. Note that the script in the *In-image compilation workspace* are the reference, these scripts may run out of date:

```
StackToRegisterMappingCogit
genAndDis: Object>>#name
options: #(ObjectMemory Spur32BitCoMemoryManager)
```

The settings available are the same than the Cog VM simulator.

Running this code should print in the Transcript the machine code version of the method in parameter.

V. Simulator menu

The bottom right text morph of the simulator provides a right-click menu with all the features.

In the menu (in the Cog Simulator, the Stack Simulator has a subset of them), there are in order:
1) toggle Transcript (toggle between simulator and external Transcript the output stream)
clone VM (clone the simulator, to have guinea pigs to reproduce bugs, typically bugs hard to reproduce once you've reproduced them once or GC bugs)
2) things related to the stacks.
'printcallStack' is the main one which prints the current stack.
'print ext head frame' prints the current stack frame, very useful too.
These 2 are the most useful. Other entries are situational.
3) 'printOop:' expects parameter in hex, printing an oop, if non immediate the header and all the fields.
disassemble entries are very useful to disassemble where the VM has crashed or disassemble a method that looks suspicious based on its address.
4) inspect objectMemory or interpreter to look into the variables value, if crash in GC or interpreter
Or run the leak checked to look for memory leaks.
Or inspect cogit if a bug happened during JIT compilation
Or inspect the method zone, typically useful to analyze it
5) print cog methods and trampoline (similar to disassemble, used for debugging the machine code)
All the *break* things stop execution on a specific selector / pc / block / ..
If single stepping is enabled (you need to do that only on small portion of code, the machine code gets dog slow), then you can report recent instructions to see the register state at each machine instructions, etc).

VI. Warm-up exercises

1. First class table page inspection

Inspect the object memory, then look for the first class table page instance variable. It's an oop referencing an array, try in the simulator to "printOop:" the address of the first class table page that you found. It should print it in the Transcript, the first entries are immediate, in Spur32 SmallInteger/Character/SmallInteger.

2. Showing the active method

Print the active call stack. There is one line per stack frame. For example,
`16r101300 M MultiByteFileStream(StandardFileStream)>basicNext`
means that:

– the stack frame address in the stack zone is `16r101300`
– the machine code version of the method is executed in this frame (M and not I).
– the receiver has the type `MultiByteFileStream`
– the stack frame on top of the stack is the activation for the method `StandardFileStream>>basicNext`

Use one stack frame's address to print the stack frame. There is a field named method. If the method is jitted, two address are available, the machine code version and the bytecode version, else only the bytecode version is there.

Try to print one of the method's address as an oop, and if it tells you "address in the machine code zone", print the cog method and its machine code instead. Try using the *disassembleMethod/trampoline…* entry on a machine code method.

3. Debugging machine code

When the simulator has started and the REPL window has popped up, select *single step*. Then enter something in the REPL window and execute it. Once done, do *report recent instructions*. You should be able to see in the Transcript the last 100 machine instructions with the register state in-between each instruction.

Have fun !

thoughts on "Simulating the Cog VM"

1. ***said:*** **Stéphane Ducasse**

   May 30, 2016 at 5:49 pm

   Thanks clement. This is a great little ressources and litle rivers are making large …

   REPLY

2. *said:* Chris Cunnington

   May 31, 2016 at 2:27 am

   This post goes a long way to explaining how to begin to explore the simulator. Thanks.

   REPLY

   - ***said:*** **Clement Bera**

     May 31, 2016 at 5:58 am

Hi Chris. In this post I merged multiple mails I sent Ben Coman in the mailing list. I am still dying to watch the video you said you would do about the simulator the day we met. As soon as you publish it, post the link here I will refer to it from that post.

REPLY

- *said:* **Chris Cunnington**

  May 31, 2016 at 12:19 pm

  OK, I'll post one next week and alert you here. It will be useful if you make comments about it to further refine what I'm showing.

- *said:* **Clement Bera**

  June 1, 2016 at 1:49 pm

  Great ! I will have a look and give you feedback. Hopefully Eliot will do that too.

3. *said:* Chris Cunnington

   June 8, 2016 at 1:16 pm

   Hi Clement,
   I'm going to be a few days late with the video. This time next week for sure. I have not made a video since 2010 and it is taking longer than expected to make a new video making rig from scratch.
   Chris

   REPLY

   - *said:* **Clement Bera**

     June 8, 2016 at 2:38 pm

     As long as you make it I'm happy. It's ok if it takes you a month.

     REPLY

4. Pingback: Sista VM Screencast | Clément Béra

8