Clément Béra ~ Smalltalk, Tips 'n Tricks

Testing unsafe operations

03 Tuesday Mar 2015

Posted by Clement Bera in Cog

≈ 3 Comments

For the runtime optimizer (Sista) I am working on, Eliot Miranda and I added new operations (we call them *unsafe operations*) in Cog's interpreter and JIT compiler. Before testing them directly by running the optimizer and looking for crashes, I wrote tests to check that each unsafe operation works as expected. I'd like to discuss a bit the design of those tests in this post.

I'll discuss a specific operation in the post, but all the unsafe operations are handled in a similar way. The operation I chose to described is called USmiLessOrEqual. This operation corresponds to <= but assumes the 2 operands are SmallIntegers, else the behavior of the instruction is undefined. The optimizer uses it only when it can ensure that the two operands are SmallIntegers.

What to test

To test USmiLessOrEqual, one could think I'm going to take an important range of integers and check that the result of each integer against each other integer within the range would give the correct answer. This approach looks nice, but it's clearly not what I want.

If one looks into the implementation of USmiLessOrEqual in the interpreter or in the JIT compiler, one would see that this operation forwards to the C operation <= for integers in the interpreter, and to the CPU instructions compare then jumpBelowOrEqual/jumpGreater in the JIT compiler. Hence, by testing a range of integers, one would actually test if the C operation <= for integers or the processor opcodes for comparison and branching actually works. I don't want to test the processor nor the C operations, I already trust them.

What I do want to test, however, is that the interpretation of the code and the machine code generated by the JIT compiler are correct for USmiLessOrEqual. For coverage, I don't want to test only a single case but I want all the branches to be taken. That means, for example, that I want to test all the possible branches used to generate machine code for USmiLessOrEqual in the code of the JIT compiler as well as all the branches that can be taken at runtime in the generated machine code.

Let's discuss how many possible branches there are when executing the code that performs the operation USmiLessOrEqual with the two operands a and b. I'll discuss the different branches in the JIT compiler and machine code generated. I won't discuss about the interpreter, but basically it has only a subset of the cases I describe.

Pseudo-code:

```
a USmiLessOrEqual: b
```

Bytecode:

```
pushTemp: 0
pushTemp: 1
callPrimitive: #USmiLessOrEqual
```

This operation expects the two operands to be integers. This is ensured by the runtime optimizer, and the behavior when a or b is not an integer is undefined, so we do not need to test the cases where a or b is not a SmallInteger. We'll assume for the rest of the blog post that the two operands are SmallIntegers.

One could think there are actually two cases for this operation (we'll call them cases A):

Case A1: a <= b, so the operation should answer true.

Case A2: a > b, so the operation should answer false.

These two cases will definitely answer different results, they'll definitely take two different paths in the virtual machine (one path pushing true on stack, the other pushing false) so they need both to be tested. But that's more complex: if the operation USmiLessOrEqual is directly followed by a branchTrue or a branchFalse, the JIT compiler generates different machine code to quicken the branch execution.

The machine code for "a USmiLessOrEqual: b", if not followed by a branch, looks like that:

```
cpu compare: a with: b
cpu jumpBelowOrEqual:
cpu push: false
cpu jump:
cpu push: true
...
```

There's already a branch in the machine code, with one path pushing true and one path pushing false on the stack.

The machine code for "a branchFalse" or "a branchTrue" looks like that:

```
cpu compare: a with: false
cpu jumpEqual: (jump to false branch: )
cpu compare: a with: true
cpu jumpEqual: (jump to true branch: )
cpu callMustBeBooleanTrampoline
```

```
(code for false branch)
...
cpu jump: (jump over true branch)
(code for true branch)
...
```

There are three possible paths for a normal branch, depending on if a is true, false or a non boolean.

The JIT quickens the branch by directly using the branches of USmiLessOrEqual and not generating the push booleans as well as the comparison between booleans and true and false. USmiLessOrEqual answers true or false, so there's no need to compile the mustBeBoolean fall back. The generated code for: "(a USmiLessOrEqual: b) branch" will then look like that:

```
cpu compare: a with: b
cpu jumpBelowOrEqual: (jump to true branch)
(code for false branch)
...
cpu jump: (jump over true branch)
(code for true branch)
...
```

We therefore needs to define three cases (we'll call them cases B), depending on what follows USmiLessOrEqual:

Case B1 (no branch):
a USmiLessOrEqual: b

Case B2 (branchTrue):
(a USmiLessOrEqual: b) ifTrue: ["some code"]

Case B3 (branchFalse):
(a USmiLessOrEqual: b) ifFalse: ["some code"]

Cases A combined with cases B means that for the operation USmiLessOrEqual, we have *6 (2 * 3) different possibles cases* to be tested.

In addition, the JIT compiler has to generate heavily optimized code (by carefully selecting the native instructions to generate) for unsafe operations as they're present in optimized methods which are frequently used.

We'll define two terms in the sense of Cog's JIT compiler to explain briefly how instruction selection is performed.

A *cpu constant* means that the operand is conceptually a literal. I say conceptually because some literals (such as true, false, nil) are literals but are not present in the literal array of methods.

A *cpu variable* is a value that is in a register, spilled on stack or a cpu constant.

Let’s look again at the bytecode for “a USmiLessOrEqual: b”:

```
pushTemp: 0
pushTemp: 1
callPrimitive: #USmiLessOrEqual
```

The pushTemp: operations have generated in the JIT compiler code to put the operands in registers or to spill them on stack as they’re temporary variable. If the variable was a cpu constant, it would have just remembered the value and will generate the code for the cpu constant only when the constant will be used. For this operation, both operands are necessarily cpu variables.

Instruction selection depends on if one of the two operands is a constant or not. The generated code will first put values on stack in registers, then, if one operand is a constant, the JIT will generate a native instruction comparing a constant versus a register, else it will generate a native instruction comparing two registers.

The JIT compiler distinguishes 3 different operations at compilation time for USmiLessOrEqual:

Case C1: a is a cpu constant and b is a cpu variable

Case C2: a is a cpu variable and b is a cpu constant

Case C3: a is a cpu variable and b is a cpu variable

The JIT compiler backend assumes that if both operands are cpu constants, the operation would have been statically computed by the optimizer at compilation time. The operation would still work, it would just be optimized as if one of the operand was a cpu variable (putting one of the constant into a register first).

This means that in the machine code for USmiLessOrEqual, one has to test 18 cases (Cases A combined with cases B combined with cases C).

How to test all the cases

For convenience, I’m going to test this instruction image-side and not VM-side.

To test each case, I need to have a compiled method with the bytecode sequence I want to test. For example, I may want a sequence such as the two operands are cpu variables (not cpu constants) and the operation USmiLessOrEqual is followed by a branchTrue.

To do that, I created a method to instrument. Let’s look at its code:

```
methodToInstrument
"This method is instrumented in the tests. Do *not* edit it.

The instrumentation starts after the last temporary assignment.
Temporaries are result of message sends, so they're in registers
or spilled on stack, whereas literals are constants.
```

```
Code after the last temporary assignment is here to generate the
required literals and to let enough room in the bytecode zone of
the method for instrumentation."

| t50 t3 tArray tByteArray |
t50 := self get50.
t3 := self get3.
tArray := self getArray.
tByteArray := self getByteArray.
5 + 10 + #(1 2 3 4 5 6) first + #[1 2 3 4 5 6 7 8] first.
^ t3 + t50 + t3 + t50 + t3 + t50 + t3 + t50 + t3 + t50
```

In different tests, I will need 2 different integer constants, a byteArray and an Array in register or spilled on stack. This is why the method start by getting into temporary variables such values. The byteArray and the array are used in other tests than the one of USmiLessOrEqual. They're used for tests on variable object access unsafe operations. I will not detail them here.

Let's look quickly into the methods called:

```
get3
^ 3

get50
^ 50

getArray
^ #( 1 2 3 4 5 6 7 8 9 10 11)

getByteArray
^ #[ 1 2 3 4 5 6 7 8 9]
```

As explained in the method comments, the method will be instrumented after the last temporary assignment, in order to be able to use the values in temporary in the generated code.

Based on the original method's code, I know that at the point where method instrumentation will start:

- temporary variable number 0, t50, holds 50
- temporary variable number 1, t3, holds 3
- temporary variable number 2, tArray holds #(1 2 3 4 5 6 7 8 9 10 11)
- temporary variable number 3, tByteArray holds #[1 2 3 4 5 6 7 8 9]
- literal number 0 holds 5
- literal number 1 holds 10
- literal number 2 holds #(1 2 3 4 5 6)
- literal number 3 holds #[1 2 3 4 5 6 7 8]

The difference between values held by the literals and the ones held in temporary variables is that values held by the literals will be compiled into cpu constants, whereas values held by the temporary variables will be compiled either into an assignment to a register or spilled on stack.

Based on my knowledge on the current state of the method, I am going to use specific methods to generate the code I want. For example, if I want to generate the constant 5, I'm going to use:

```
genCst5: encoder
encoder genPushLiteral: 4
```

Now that I have everything at hand, let's instrument the method. First I need to generate the bytecodes I want. I use a simple pattern (which may not be very well designed) where I hold in an array all the possible cases and the expected value. For USmiLessOrEqual, it's cases description gives you:

1) Possible pairs of operands that answers true to USmiLessOrEqual (cases C, operands are cpu constants or not, I added a cases with 2 cpu constants as operands just in case):

```
{ self blockGenVar3 . self blockGenVar50 } .
{ self blockGenVar3 . self blockGenCst5 } .
{ self blockGenCst5 . self blockGenVar50 } .
{ self blockGenCst5 . self blockGenCst10 }
```

2) This is combined with the possible results answered (Cases A). Basically I'll check that operand1 USmiLessOrEqual: operand2 answers true, and the opposite, operand2 USmiLessOrEqual: operand1 answers false:

```
{ self blockGenSuccessively: truePair . true } .
{ self blockGenSuccessively: (self invertedPair: truePair) . false }
```

3) Lastly I combined them with possibility of being followed by branches. For this purpose I used branches that push different integers on stack to check the result. For instance:

```
jumpFalse10Else5Block
^ [ :encoder |
encoder genBranchPopFalse: 2.
self genCst5: encoder.
encoder genJump: 1.
self genCst10: encoder. ]
```

Here, if value on stack is true, there will be 5 on stack after the branch, else there will be 10. This bytecode sequence is equivalent to `ifTrue: [ 5 ] ifFalse: [ 10]` with the receiver of ifTrue:ifFalse: on stack.

Here are the different branch cases to combine (cases B), first arg is the block that generates or not the branch, the other is the expected result:

```
int := self jumpBlockResult: boolean.
```

```
{ self emptyBlock . boolean } .
{ self jumpFalse10Else5Block . int } .
{ self jumpTrue5Else10Block . int }
```

Ok. So for USmiLessOrEqual, we're going to generate 24 (2 * 3 * 4) bytecode sequences to test. Let's generate the first one:

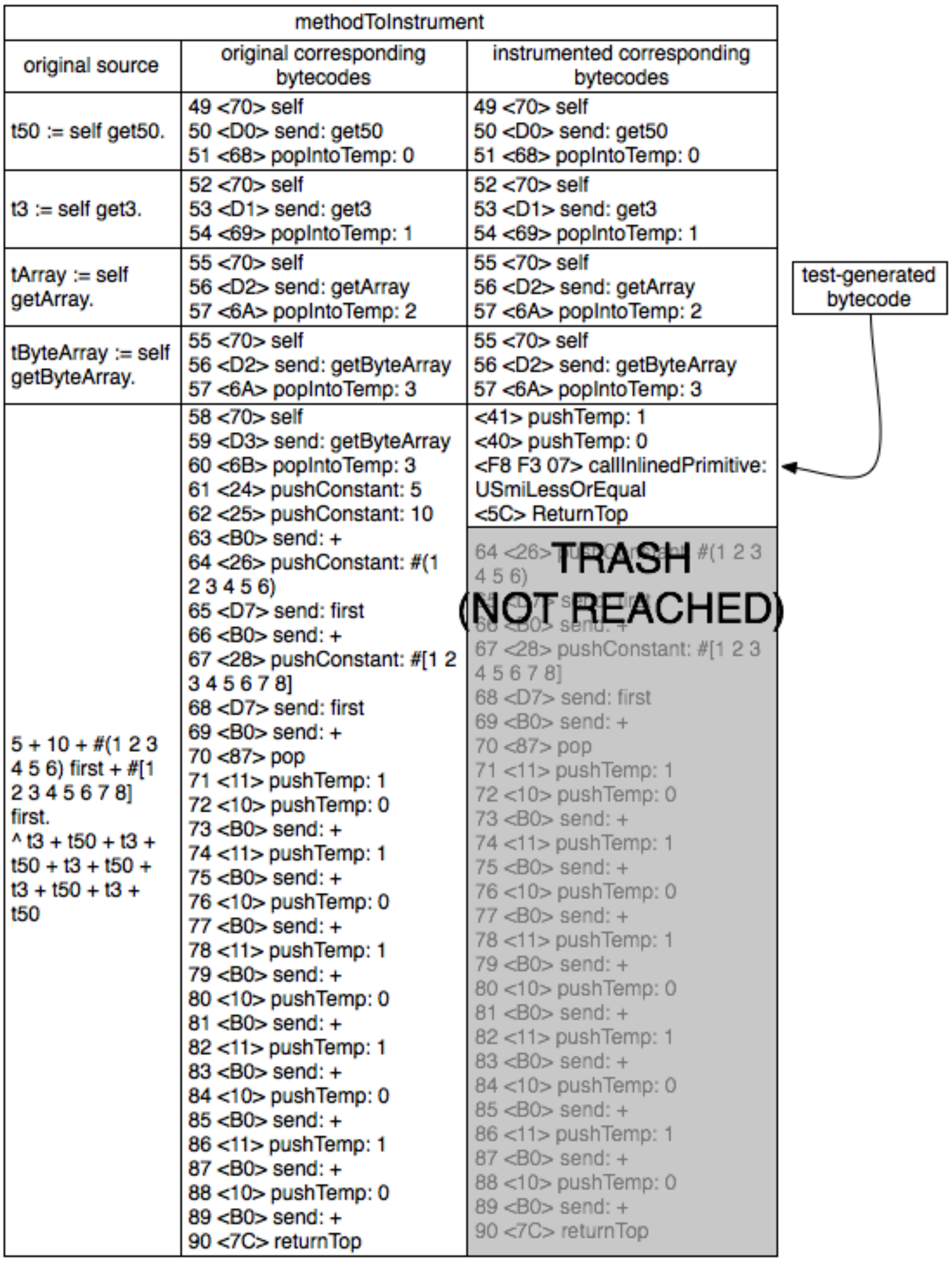
- The two operands are in registers or spilled on stack`{ self blockGenVar3 . self blockGenVar50 }`
- The result of the operation will be true (3 <= 50)
- The operation is not followed by a branch

I get the bytecode sequence:

```
65 pushTemp: 1 "holds 3"
64 pushTemp: 0 "holds 50"
248 243 7 callInlinedPrimitive: USmiLessOrEqual
92 ReturnTop
```

Note that we added a returnTop at the end to be sure that the method will return the result to test afterwards. We can't let the original bytecode that remains in the compiled method be executed after our hand-written bytecodes.

Let's now instrument the method:



methodToInstrument		
original source	original corresponding bytecodes	instrumented corresponding bytecodes
t50 := self get50.	49 <70> self 50 <D0> send: get50 51 <68> popIntoTemp: 0	49 <70> self 50 <D0> send: get50 51 <68> popIntoTemp: 0
t3 := self get3.	52 <70> self 53 <D1> send: get3 54 <69> popIntoTemp: 1	52 <70> self 53 <D1> send: get3 54 <69> popIntoTemp: 1
tArray := self getArray.	55 <70> self 56 <D2> send: getArray 57 <6A> popIntoTemp: 2	55 <70> self 56 <D2> send: getArray 57 <6A> popIntoTemp: 2
tByteArray := self getByteArray.	55 <70> self 56 <D2> send: getByteArray 57 <6A> popIntoTemp: 3	55 <70> self 56 <D2> send: getByteArray 57 <6A> popIntoTemp: 3
5 + 10 + #(1 2 3 4 5 6) first + #[1 2 3 4 5 6 7 8] first. ^ t3 + t50 + t3 + t50 + t3 + t50 + t3 + t50 + t3 + t50	58 <70> self 59 <D3> send: getByteArray 60 <6B> popIntoTemp: 3 61 <24> pushConstant: 5 62 <25> pushConstant: 10 63 <B0> send: + 64 <26> pushConstant: #(1 2 3 4 5 6) 65 <D7> send: first 66 <B0> send: + 67 <28> pushConstant: #[1 2 3 4 5 6 7 8] 68 <D7> send: first 69 <B0> send: + 70 <87> pop 71 <11> pushTemp: 1 72 <10> pushTemp: 0 73 <B0> send: + 74 <11> pushTemp: 1 75 <B0> send: + 76 <10> pushTemp: 0 77 <B0> send: + 78 <11> pushTemp: 1 79 <B0> send: + 80 <10> pushTemp: 0 81 <B0> send: + 82 <11> pushTemp: 1 83 <B0> send: + 84 <10> pushTemp: 0 85 <B0> send: + 86 <11> pushTemp: 1 87 <B0> send: + 88 <10> pushTemp: 0 89 <B0> send: + 90 <7C> returnTop	<41> pushTemp: 1 <40> pushTemp: 0 <F8 F3 07> callInlinedPrimitive: USmiLessOrEqual <5C> ReturnTop TRASH (NOT REACHED) 64 <26> pushConstant: #(1 2 3 4 5 6) 65 <D7> send: first 66 <B0> send: + 67 <28> pushConstant: #[1 2 3 4 5 6 7 8] 68 <D7> send: first 69 <B0> send: + 70 <87> pop 71 <11> pushTemp: 1 72 <10> pushTemp: 0 73 <B0> send: + 74 <11> pushTemp: 1 75 <B0> send: + 76 <10> pushTemp: 0 77 <B0> send: + 78 <11> pushTemp: 1 79 <B0> send: + 80 <10> pushTemp: 0 81 <B0> send: + 82 <11> pushTemp: 1 83 <B0> send: + 84 <10> pushTemp: 0 85 <B0> send: + 86 <11> pushTemp: 1 87 <B0> send: + 88 <10> pushTemp: 0 89 <B0> send: + 90 <7C> returnTop

On the figure, on the left is the original source code and bytecodes of the method. After the last temporary assignment, we override the bytecodes and write down our bytecode sequence. We use an instrumented method that is long enough so there's enough room to correctly write down the test-generated bytecode.

Then, we need to void the method Cog VM state. The method may already be compiled to machine code with another bytecode sequence (from the previous tests for example, as they're typically all run in a row). Voiding the Cog VM state asks the VM to flush its machine code state for the method to flush this kind of dependencies.

```
method voidCogVMState
```

Now we can run the method. I run it several times to have both the interpreter and the machine code results:

```
runInstrumentedMethod
| res |
res := { nil . nil }.
res at: 1 put: self runMethodToInstrument. "interpreter result"
1 to: 5 do: [ :i | self runMethodToInstrument ]. "heat up the JIT"
res at: 2 put: self runMethodToInstrument. "jitted result"
^ res
```

Lastly, I can compare the two results I collected from the runtime to the expected result. The operation USmiLessOrEqual is fully tested !

Now, one needs to understand that such tests are not safe: if the test fails, sometimes one will just have an assertion failure, but in most cases one has a segmentation fault to debug in the VM simulator. I could run the test directly in the VM simulator to avoid such issues, but the VM simulator takes a while to start-up and to run any code. There is no perfect solution, but at least I have a way to test that the code I wrote in the JIT compiler is correct (though the real coder doesn't test, only the ones who fear are testing😉)

thoughts on "Testing unsafe operations"

1. Pingback: Opérations dangereuses | L'Endormitoire

2. *said:* **Stéphane Ducasse**

   April 15, 2015 at 7:42 pm

   This is great that you focus on testing such hidden parts of the JIT compiler.
   Great work!

   REPLY

3. *said:* **Stéphane Ducasse**

   April 15, 2015 at 7:42 pm

   Reblogged this on Weekly news about Pharo.

   REPLY

3