Clément Béra ~ Smalltalk, Tips 'n Tricks

CogMethod's Maps

19 - *Monday* - *Sep 2016*

Posted by Clement Bera in Cog

≈ Leave a comment

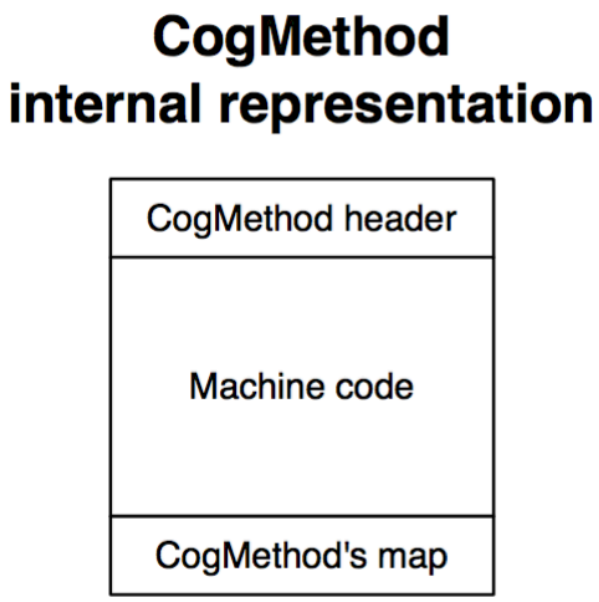
Hi everyone,

Today I am going to discuss the little map at end of cog methods. This map is used by the VM to get information about the machine code during various operations such as send's site relinking and garbage collection.

Cog methods

In Cog, a cog method is the machine code version of a Smalltalk (bytecoded) compiled method. It is located at the different place than the bytecode version, as the machine code needs to be in an *executable* memory zone. On the contrary, the bytecode version needs only to be in readable, and potentially writable, memory zone. In addition, the cog method requires specific care by the garbage collector (GC) as the GC cannot scan the machine code as it would scan objects.

A cog Method looks like this:



The cog method header contains metadata related to the cog method. One information stored here, for example, is the size of the cog method. The size is used to be able to iterate over all the cog methods in the machine code zone. Another information is the machine instruction address for the stack overflow check, which is one of the first machine instruction of the machine code of the method.

At the end of the cog method, a small map is present. This map is used by the garbage collector, the machine code zone compactor and various other utilities such as relinking inline caches. The map contains annotations of the machine code.

Machine code annotations

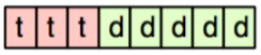
While generating the machine code version of a bytecoded method, cogit (cogit is Cog's JIT compiler) annotates different machine instructions. We will divide the annotations of the machine code in 5 main categories:

- *Object references:* When an object is directly referred from machine code, its address is written as a full machine word and is annotated as an object reference. This annotation is used typically for references to the method literals: on some machine back-ends such as x86, literals are inlined in the machine code and their addresses are annotated as object references. This annotation is used by the GC: if one of the literals is moved in memory by the GC, the GC uses the annotation to know the location of the literal's address in the machine code and update it to its new address.
- *Absolute PC references:* This annotation is used exclusively for cog method references. In the stack frame creation code, at the beginning of the machine code of frameful methods, there is machine code which upon execution writes down in the active stack frame the address of the cog method. This address is an absolute address in the machine code zone. It needs to be annotated because when the machine code zone compactor compacts the machine code zone, it moves cog methods and therefore needs to update their addresses.
- *Relative calls:* As for absolute addresses, relative calls needs to be updated when the machine code zone is compacted. In most architecture, relative calls are used to call trampolines.
- *Bytecode PCs:* At each interrupt point, the VM needs to be able to recreate the interpreter version of the frame for the programmer to debug the code. This annotation is therefore used for each interrupt point, except sends which are implicitly mapped with the send call annotation (explained just below).
- *Send calls:* Each send's site needs to be annotated for the inline cache logic to be able to link and unlink the send and to have a bytecode pc annotation. There are different kinds of sends (normal, super send, directed super send, newspeak sends, etc.), and each of them has a specific annotation.

Encoding

The challenge is to encode the annotations in a very compact way. To do so, the annotation are encoded as an annotation type and the distance in the machine code from the previous annotation. There is no additional information (no annotation metadata or anything like that).

The annotation scheme used in Cog method maps is as follow:



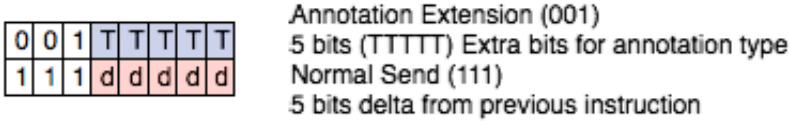
Each annotation is encoded in a byte. In each byte, 3 bits are used to precise the annotation type and 5 bits are used to encode the distance from the previous annotation.

The annotation types described in the previous sections are encoded as follow (type – annotation name):

- 2: Object reference
- 3: Absolute PC reference
- 4: Relative Call
- 5: Bytecode PC
- 6: Newspeak send
- 7: Normal send

The encoded values 0 and 1 are used to extend the other annotations:

- 0: Displacement. If the distance from the previous annotation is greater than 31 (depending on machine architectures, 31 bytes or 31 words), this annotation is used to extend the distance.
- 1: Annotation extension. This annotation is used as a prefix to extend the normal sends (type 7). This allows the VM to support more types than 7. The additional types, from 8 to 255, can be used to encode uncommon sends. Using an extra byte only for uncommon send allows not to waste too many space. The following figure details the extended byte encoding.



Thanks to the annotation extension, 5 other kind of sends are implemented:

- 08: Super send
- 09: Directed super send
- 10: Newspeak self send
- 11: Newspeak dynamic super send
- 12: Newspeak implicit receiver send

Reading the annotations

The GC only needs to read object references while the machine code zone compactor only needs to read relative calls and absolute PC references. This is relatively easy: the first annotation is always the stack overflow check, which address is known by the Cog method header. From the first instruction, the VM can iterate over the annotations, computing at each step the new machine instruction address. Each instruction can be patched this way. I won't go into details on how this happens as this is back-end specific.

Reading the bytecode PC annotations to map the machine code instruction pointer to the bytecode program counter is slightly more complicated. The VM needs to iterate at the same time both over the machine code, computing at each step the machine instruction address, and the bytecode of the method, computing at each step the bytecode PC of the current mapped instruction. While iterating over the bytecode, the VM stops at each instruction which is mapped from bcpc (bytecode PC) to mcpc (machine code PC). When iterating over the nth annotation mapped to a PC in machine code, the VM also knows the nth annotated bytecode as well as its bytecode PC. This way, the bytecode PC annotation does not even need to encode the bytecode PC, it is found out each time by iterating over both versions of the method.

I hope you enjoyed the post.