The Cog Smalltalk Virtual Machine
writing a JIT in a high-level dynamic language

Eliot Miranda
Independent Consultant
San Francisco
eliot.miranda@gmail.com

Introduction

Cog is an extension of the Squeak VMMaker [1] virtual machine framework for implementing Smalltalk virtual machines. The framework implements virtual machines in a subset of Smalltalk and allows their incremental development and execution within the Smalltalk IDE, and then translates the code to C to produce a production VM.

Cog adds a just-in-time compiler for x86 to the existing interpreter. At this stage in its evolution Cog produces a VM that runs language-intensive (rather than library intensive) benchmarks from the computer language shootout [2] at around 5 times faster than the interpreter. While Cog's code generation and message send optimization techniques are not novel, its plumbing of an x86 simulator into its framework, which allows full simulation of generated machine code, is probably of some interest to the VMIL community. The VM is also well-modularised and extensible, as exemplified by recently being applied to Newspeak [6].

Cog is fully open-source, released under the MIT Squeak license. The author maintains a blog with detailed articles on various components of the system [7].

General Terms Virtual Machine Implementation, High-level language, dynamic language, just-in-time compiler.

1. Overview

Cog is implemented within the Squeak VMMaker package and translated to C using the package's Slang translator. Being a Smalltalk program, Cog is evaluable within a Smalltalk environment and hence allows incremental development and debugging of most of the system. But to evaluate generated x86 machine code the framework is extended with an interface to the core processor implementation in the Bochs [3] IBM PC emulator, which is married to the byte array object holding the heap and generated code, and to variables in the Smalltalk interpreter and JIT objects.

Cog was implemented in a commercial context, the Qwaq/Teleplace[1] [4] application of the Croquet [5] virtual worlds system to business communication. In this context, incremental delivery of increased performance over the base interpreter and minimisation of risk was a major requirement. Hence Cog evolved in several stages, and forethought was given to making major components pluggable to support that evolution.

This paper follows the above sketch. We first cover the Slang Smalltalk-to-C translator and its extensions, going on to describe the processor simulation system with Bochs at its core. The next two sections then look at different aspects of modularity and extensibility in Cog. The last section presents a couple

[1] The company was founded as Qwaq, Inc (see the left of a qwerty keyboard) and changed its name to Teleplace to avoid offense; try selling Qwaq Forums to a medical establishment.

of cool hacks made possible by using Smalltalk and then we conclude.

2. Slang

As described in [1] the VMMaker framework uses a subset of Smalltalk that is easily translated to C. The translator, called Slang, is a one-pass traversal over the parse trees of methods in the framework, straightforwardly writing out C code. This implies no rich data structures, no Dictionaries (hash maps), Sets or OrderedCollections (dequeues, stacks), only Arrays, simple to:by:do: loops, and no polymorphism; each message in the framework must have only one implementation.

The base Interpreter is monolithic, inheriting from the object representation/heap/garbage collector ObjectMemory. All messages are therefore to self. Elements of this architecture were far too restrictive for a JIT and the architecture was extended, by splitting the Interpreter from the ObjectMemory, and keeping the JIT separate from these two. In addition to this support for multiple classes, there is support for record and pointer-to-record types which are used extensively to implement data structures such as in the Interpreter, a stack page of Smalltalk method activations, or in the JIT, an abstract instruction, or an entry on the simulation stack used to map stack bytecode to register-based code.

The base Interpreter uses a ByteArray called memory to model the object heap, using integer addresses in the memory ByteArray as object references. In Cog this memory ByteArray is extended with space for stack pages holding method activations in the form of stack frames, and with space for generated code.

3. Bochs

To evaluate generated x86 machine code the framework is extended with an interface to the core processor implementation in the Bochs [3] IBM PC emulator. This accesses data in the memory ByteArray, and executes instructions in the generated code portion of the memory ByteArray. Using Bochs is both less work and doubtless provides faster execution than implementing an x86 simulator in Smalltalk. The Bochs x86 simulator is derived by taking the core processor definition from Bochs, changing the memory interface so that it fetches/stores from/to the memory ByteArray and wrapping this up in a Squeak VM plugin[2] with a primitive interface whose receiver is an Alien[3] for the Bochs C++ processor object and whose arguments are commonly the memory ByteArray and some read/write/execute limits. The primitive interface comprises 6 methods for execution, single stepping, disassembly, initialization and error message retrieval. The remaining methods access the register state of the x86 represented by the C++ object.

Machine code is generated into the "method zone", located low in the memory ByteArray. Correspondingly there are structures in the memory object, such as machine-code methods, that must be accessed from the Smalltalk side of things, and hence proxy objects are used to wrap addresses in the memory object (see Figure 1 on page 6). These wrapper classes are auto-generated from the classes that define the relevant ADTs, e.g. CogMethod defines the ADT for a machine-code method header, generates the C typedef in the Slang translation and generates CogMethodSurrogate32/64 to access CogMethod instances in the simulation memory (see appendix).

Accessing methods and variables in the interpreter, such as the bulk of primitives that are not implemented in machine code (window, file and operating system interfaces, FFI, save/restore and complex execution primitives such as perform: (Smalltalk's apply)), requires a way of mapping machine code

[2] A Squeak VM plugin is a class implementing a set of primitives that either implement or provide an interface to some useful functionality. It can be either statically or dynamically linked into the VM and is hence a modular extension to the VM providing encapsulation and platform independence over an FFI based implementation. See [12].

[3] An Alien is a wrapper for some external address along with a set of primitives such as longAt:put: for accessing external memory relative to that address.

accesses to Smalltalk message sends. This is done using illegal addresses. Each method and/or variable that needs to be accessed from machine code is given a unique illegal address and entered in a read and a write dictionary mapping illegal address to evaluable (either a Symbol or a Block) that can be used to either fetch/store a variable or call a method. Bochs is invoked to execute machine code until it hits an illegal address, at which point the invocation primitive fails and maps the Bochs exception into a Smalltalk ProcessorSimulationTrap Exception subclass that records the offending instruction, next instruction address, the illegal address, and register source or destination. The system simulator then handles ProcessorSimulationTrap exceptions by looking up the illegal address in the relevant Dictionary, evaluating it and continuing execution.

Execution of the simulator (see Figure 1 on page 6) starts in Smalltalk, the interpreter being pure Smalltalk. But once the CoInterpreter has asked the JIT to compile a method execution may continue in machine code, surfacing back up to Smalltalk on encountering an illegal address, or the VM being interrupted. Use of single-stepping allows capturing machine instructions so that one may conveniently examine the previous N instructions on hitting a bug.

4. Modularity, Evolution and Mixed-Mode Execution

Cog was implemented in a commercial context, the Qwaq/Teleplace [4] application of the Croquet [5] virtual worlds system to business communication. In this context, incremental delivery of increased performance over the base interpreter and minimisation of risk was a major requirement. One unique challenge in implementing a high-performance Smalltalk virtual machine is efficient access to contexts, Smalltalk's first-class activation records [8][9]. Such a machine must map activation records to stack frames, to be able to apply conventional inline cacheing techniques, which are focussed on call instructions. But the mapping must also allow modification of the activation records, causing modification or destruction of the associated stack frame as appropriate. Such code is complex and Qwaq/Teleplace management rightly wanted to speed delivery of improved performance by requiring that the context-to-stack mapping machinery was realised first in the context of a modified interpreter, without having to wait for the development of the JIT. This context-to-stack mapping StackInterpreter is typically about twice as fast as the original context-based Interpreter for Smalltalk-intensive code. Further, Qwaq/Teleplace negotiated a naive initial JIT code generator for early delivery. This code generator has a one-to-one correspondence between bytecodes and generated code, hence every bytecode that pushes an operand results in machine code that also pushes that operand on the real stack.

Given the StackInterpreter it was natural to use it via the CoInterpreter subclass[4] which married the context-to-stack mapping machinery, bytecode interpreter and primitive set to Cog's JIT (called the Cogit), providing a mixed execution model where code may either be interpreted or compiled to machine code and freely inter-called. Hence Cog evolved in five stages, an extended bytecode set and new Smalltalk bytecode compiler that implements full closures and hence enables context-to-stack mapping, a faster context-to-stack mapping interpreter, a naive code generator, a threading model supporting a non-blocking FFI and a sophisticated code generator, and is currently being extended with a more efficient object representation and a garbage collector that supports pinning to serve the threaded FFI. This evolution has been enabled by keeping the VM modular with forethought given to making things like the object representation and bytecode set pluggable.

There are distinct advantages to the mixed execution model. The Cogit can refuse to compile certain

[4] Slang was extended to allow methods to be redefined in subclasses, with super sends being expanded during translation.

methods, especially extremely large ones that consume too much code space. Such methods are typically class initializers and rarely run[5]; interpreting them once typically consumes much less space and is much faster than compiling and running them once. The Cogit has a fixed size machine code zone, which has good performance characteristics due to its small working set size, and is also quick to scan and update when, for example, redefinition of a Smalltalk method in the IDE necessitates flushing of associated machine code and inline caches. The CoInterpreter implements an extremely simple policy, only compiling a method if it has less than a certain number of literals and has been found in the first-level method lookup cache. Hence Cog typically compiles a bytecoded method to machine code on its second invocation[6], avoiding compiling any method used only once (e.g. for initialization).

When the code zone fills up a reclamation is scheduled which throws away least recently used methods up to a quarter of the space. In a pure JIT in order to continue execution the VM may have to reclaim code while attempting to edit that code (e.g. to add a method to a polymorphic inline cache) which can be extremely tricky, given that reclamation involves compacting the space, relocating all methods in it. But with the mixed mode model reclamation is performed only at suspension points at the next available opportunity, providing significant simplification.

5. Modularity and Extensibility

The first major extension of the VM was the implementation of a threading model, similar to that in the Python VM, but using an extremely efficient locking mechanism [10], where any number of native threads can share the VM with only one owning the VM at any one time. This is implemented in the CoInterpreterMT subclass of the CoInterpreter, itself a subclass of StackInterpreter.

The second major extension was the implementation of a stack-to-register mapping code generator that avoids reifying operands on the actual stack by maintaining a simulation stack during compilation and only generating code to access an operand when compiling a bytecode that consumes operands such as a send or instance variable assignment. This simple technique may have been pioneered by Peter Deutsch in the ObjectWorks 2.4 Smalltalk VM, but was not described in the literature. It is similar to the technique used in the Intel VTune Java JIT [11]. Also developed is a code generator that provides support for adaptive optimization and speculative inlining, by adding performance counters to code generated for conditional branch bytecodes (which provides basic block frequency information), and code to reify inline cache state as Smalltalk objects. Hence the Cogit hierarchy is

```
Cogit
   SimpleStackBasedCogit
      StackToRegisterMappingCogit
         StackToRegisterMappingCogitChecker
         SistaStackToRegisterMappingCogit
```

where StackToRegisterMappingCogitChecker as a test class that checks stack depths are correct at suspension points, and Sista stands for Speculative Inlining Small-talk Architecture.

The bytecode decode and generation scheme is pluggable; a bytecode set is defined by an Array of CogBytecodeDescriptor instances that contain a message selector to generate the bytecode (mapped to a function pointer in C), the number of bytes in the bytecode, etc. Hence there is no monolithic central switch statement in the compiler, and bytecode sets should be easily changed, although to date this has

[5] of course during development class initializers may be run often, but they may be evaluated only after redefinition, and being evaluated on demand by the programmer their performance is typically not an issue.

[6] The CoInterpreter will also compile to machine code if a backward branch bytecode (which occurs at the end of all loops) is evaluated a consecutive number of times in the same method.

only been exercised in adding bytecodes for the Newspeak implementation.

The object representation is also pluggable, all code for generating object access being abstracted into a separate hierarchy, currently with only two classes, an abstract CogObjectRepresentation and its sole subclass CogObjectRepresentationForSqueakV3 that implements the current Squeak object format. I am just starting work on a new object representation based on a scheme the author used in the 64-bit VisualWorks implementation where object headers contain indices into a class table rather than a full-width pointer to a class object. This has space advantages but also advantages in inline caches, which now contain constant class indices instead of class pointers subject to movement by the garbage collector.

6. Clever Tricks

The utility of being able to use Smalltalk, a scriptable dynamic language, to run queries during simulation, for example to gather statistics from inline caches, should not be underestimated.

Preceding sections have been glib about the details of simulation. One issue is integer overflow in C and its absence in Smalltalk (which has bignum support). Hence the framework uses simulator subclasses to implement certain methods that reconcile Smalltalk and C's peculiarities and provide simulation-only debugging code. For example,

```
  ObjectMemory methods for interpreter access
  isIntegerValue: intValue
     ^ (intValue bitXor: (intValue << 1)) >= 0

  ObjectMemorySimulator methods for simulation
   isIntegerValue: valueWord
      ^ valueWord >= 16r-40000000 and: [valueWord
<= 16r3FFFFFFF]
```

So CoInterpreter started life with a substantial subclass CogVMSimulator. When CoInterpreterMT was added, I wanted to avoid maintaining two simulator subclasses with essentially the same code. So CogVMSimulator inherits from CoInterpreterMT and at start-up, using reflection, arranges that all methods implemented in CoInterpreter and overridden in CoInterpreterMT have an override in CogVMSimulator that invokes either the CoInterpreter version or the CoInterpreterMT version depending on CoInterpreterMT's cogThreadManager instance variable being nil or non-nil. For example:

```
  CogVMSimulator methods for simulation switch
  initializeInterpreter: bytesToShift
     "Auto-generated by CogVMSimulator>>
ensureMultiThreadingOverridesAreUpToDate"

       ^self perform: #initializeInterpreter:
            withArguments: {bytesToShift}
            inSuperclass: (cogThreadManager
                              ifNil: [CoInterpreter]
                              ifNotNil: [CoInterpreterMT])
```

In adding threading the system needed to simulate multiple processors. This was implemented by wrapping the Bochs Alien with a proxy that essentially implements only doesNotUnderstand:. The doesNotUnderstand: method then checks the current process and if it has changed, saves the register set and switches it to that of the current process.

7. Conclusion

Cog is a new implementation of the Smalltalk-80 virtual machine for Squeak. It is a Smalltalk program, providing incremental development and high-level debugging, while it is a JIT, depending on external processor simulators to efficiently simulate machine code execution. It is well modularised and extensible and as such provides a sound basis for the evolution of Smalltalk execution engines. It is in active development and is the standard execution engine for the Squeak and Pharo Smalltalk dialects and for the Newspeak programming language.

Acknowledgements

I wish to thank Andreas Raab and Greg Nuyens ex of Qwaq/Teleplace, where Cog started and matured, and Yaron Kashai at Cadence Design Systems, Inc, where I have the opportunity to continue Cog's evolution and apply it to Newspeak.

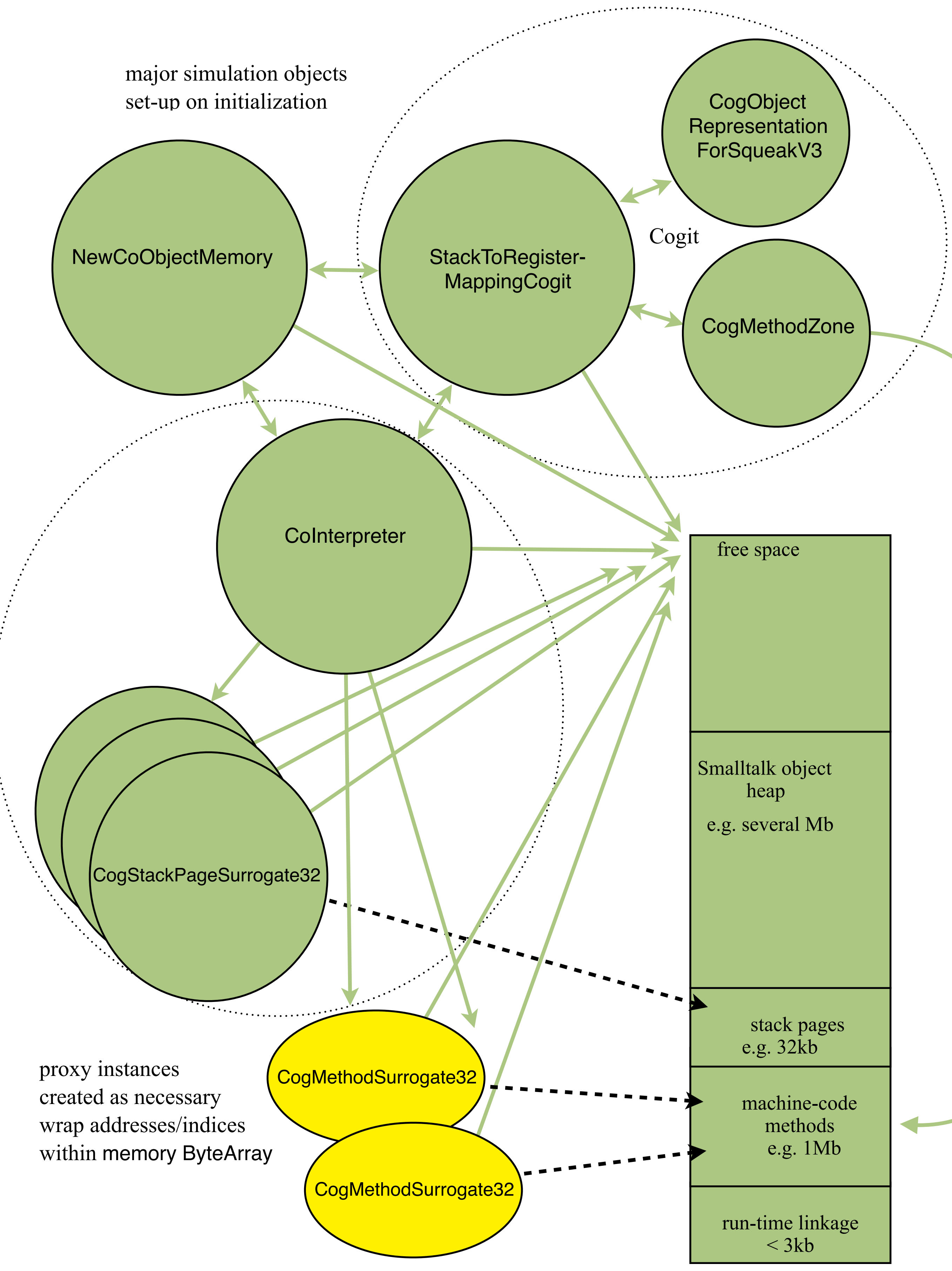

Figure 1. Cog simulation architecture

References

[1] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay , “Back to the future: the story of Squeak, a practical Smalltalk written in itself”, Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications, pp 318–326, ACM, 1997

[2] computer language benchmarks game http://shootout.alioth.debian.org/

[3] Bochs, http://en.wikipedia.org/wiki/Bochs, http://bochs.sourceforge.net/

[4] Qwaq/Teleplace, http://www.teleplace.com/

[5] Croquet, http://en.wikipedia.org/wiki/Croquet_Project

[6] Newspeak, http://newspeaklanguage.org/

[7] Cog blog, http://www.mirandabanda.org/cogblog

[8] http://www.mirandabanda.org/cogblog/2009/01/14/under-cover-contexts-and-the-big-frame-up

[9] Eliot Miranda, “Context management in VisualWorks 5i”, OOPSLA '99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design, November 1999, Denver, CO. http://www.esug.org/data/Articles/misc/oopsla99-contexts.pdf

[10] David Simmons, private communication.

[11] Ali-Reza Adl-Tabatabai, Michael Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, James M. Stichnoth, "Fast, Effective Code Generation in a Just-In-Time Java Compiler", In Proceedings of the SIGPLAN `98 Conference on Programming Language Design and Implementation, pp 280-290. Published as SIGPLAN Notices 33(5), May 1998.

[12] Tim Rowledge, “A Tour of the Squeak Object Engine”, http://www.google.com/search?q=%22A+Tour+of+the+Squeak+Object+Engine%22+filetype%3Apdf

Appendix - CogMethod code

```
VMStructType
    subclass: #CogBlockMethod
    instanceVariableNames:
        'objectHeader homeOffset startpc padToWord
         cmNumArgs cmType cmRefersToYoung
         cmIsUnlinked cmUsageCount stackCheckOffset'

CogBlockMethod
    subclass: #CogMethod
    instanceVariableNames:
        'blockSize blockEntryOffset methodObject
         methodHeader selector'
```

CogMethod auto-generated typedef:

```
typedef struct {
      sqInt        objectHeader;
      unsigned     cmNumArgs : 8;
      unsigned     cmType : 3;
      unsigned     cmRefersToYoung : 1;
      unsigned     cmIsUnlinked : 1;
      unsigned     cmUsageCount : 3;
      unsigned     stackCheckOffset : 16;
      ushort       blockSize;
      ushort       blockEntryOffset;
      sqInt        methodObject;
      sqInt        methodHeader;
      sqInt        selector;
  } CogMethod;
```

Sample auto-generated CogMethodSurrogate32 methods:

```
cmNumArgs
   ^memory unsignedByteAt: address + 5

cmType
   ^(memory unsignedByteAt: address + 6) bitAnd: 16r7

cmType: aValue
   self assert: (aValue between: 0 and: 16r7).
   memory
       unsignedByteAt: address + 6
       put: ((memory unsignedByteAt: address + 6)
                bitAnd: 16rF8) + aValue.
      ^aValue

cmRefersToYoung
   ^(((memory unsignedByteAt: address + 6)
       bitShift: -3) bitAnd: 16r1) ~= 0

methodObject: aValue
   ^memory unsignedLongAt: address + 13 put: aValue
```