A bytecode set for adaptive optimizations

Clément Béra

RMOD - INRIA Lille Nord Europe

clement.bera@inria.fr

Eliot Miranda

Cadence Design Systems

eliot.miranda@gmail.com

Abstract

The Cog virtual machine features a bytecode interpreter and a baseline Just-in-time compiler. To reach the performance level of industrial quality virtual machines such as Java HotSpot, it needs to employ an adaptive inlining compiler, a tool that on the fly aggressively optimizes frequently executed portions of code. We decided to implement such a tool as a bytecode to bytecode optimizer, implemented above the virtual machine, where it can be written and developed in Smalltalk. The optimizer we plan needs to extend the operations encoded in the bytecode set and its quality heavily depends on the bytecode set quality.

The current bytecode set understood by the virtual machine is old and lacks any room to add new operations. We decided to implement a new bytecode set, which includes additional bytecodes that allow the Just-in-time compiler to generate less generic, and hence simpler and faster code sequences for frequently executed primitives. The new bytecode set includes traps for validating speculative inlining decisions and is extensible without compromising optimization opportunities. In addition, we took advantage of this work to solve limitations of the current bytecode set such as the maximum number of instance variable per class, or number of literals per method. In this paper we describe this new bytecode set. We plan to have it in production in the Cog virtual machine and its Pharo, Squeak and Newspeak clients in the coming year.

1. Introduction

The Cog virtual machine (VM) is quite efficient compared to popular language such as Python or Ruby, but is still far behind mainstream languages such as Java. This is because the VM does not have an adaptive Just-in-time (JIT) compiler, a tool that recompiles on the fly portion of code frequently executed to portion of code faster to run.

As we implement the adaptive optimizer as a bytecode to bytecode optimizer, we rely heavily on the bytecode set design. We need to adapt it to be suitable for optimizations and extend it with unsafe operations. The current bytecode set needs revising because of the lack of available bytecodes and the lack of unsafe operations (operations faster to run as they do not check any constraints) as well as the implementation of primitives, forbidding respectively to efficiently extend the bytecode and to inline primitive methods. In addition, the current bytecode set has well-known issues such as a maximum number of instructions a jump forward can encode and we took advantage of the bytecode set revamp to fix these problems, ending up with important simplification in the VM implementation.

We design a new bytecode set with the following improvements:

- It provides many available bytecodes to be easily and efficiently extendable.
- It features a set of unsafe operations for the runtime optimizer.
- It encodes the primitives in a way they can be inlined.
- It solves some well-known issues of the old bytecode set.

In this paper we describe the constraints we have to design a better bytecode set. We specify for each constraint if it applies for all the bytecode set designs or only in our case to design a bytecode set for adaptive optimizations.

After describing how the current bytecode is used in our virtual machine and Smalltalk clients, we discuss the current issues and missing features, then show how we solve the current issues. We also discuss some aspects of the new bytecode set and compare it to related work.

2. The Cog bytecode execution and memory model

Smalltalk has a runtime very similar to the Java Virtual Machine (JVM)[12], the Common Language Infrastructure (CLR)[6] and other common platform-independent object-oriented languages. To execute code, a high level compiler translates the Smalltalk source code into bytecode, a low level language. The bytecode is then executed by a virtual machine, being either interpreted or compiled down to na-

tive code through a just-in-time compiler. The virtual code is platform-independent and is encoded in bytes for compactness. Its byte encoding gives it the name bytecode.

A new memory manager, named Spur, has been recently introduced in the Cog VM[11]. All the figures and examples about memory representation we describe show the objects in the new Memory Manager format.

Two main bytecode sets are now supported in the Cog VM. One targets the Smalltalk clients, Squeak and Pharo, whereas the other one targets a research language named Newspeak. In this paper we focus on the bytecode set for Smalltalk clients, as the new bytecode set is for now exclusively for Smalltalk and would need to be adapted to be used with Newspeak.

2.1 Vocabulary

After looking at several bytecode sets and working on ours, we decided to describe bytecode operations by using three forms. Here is the definition of the three forms proposed, with examples from the new bytecode set:

- *single bytecode:* the instruction is encoded in a single byte. For example, the byte B0 means that the execution flow needs to jump forward by two instructions and the bytecode 4D means that the interpreter should push the boolean true on the stack.

- *extended bytecode:* the instruction is encoded in two bytes. The first byte defines the instruction and the second byte encodes an index relative to the execution of the instruction. For example, the byte E5 means that the interpreter should push a temporary variable on the stack, the index of the temporary variable being encoded in the next byte.

- *double extended bytecode:* the instruction is encoded in three bytes. The first byte defines the instruction and the second and third bytes encode an index relative to the execution of the instruction. For example, the byte FC means that the interpreter needs to push on the stack a variable in a remote array, the index of the remote array being encoded in the second byte and the index of the variable in the remote array is encoded in the third byte.

We call extended bytecode and double extended bytecode *argument bytecodes*, because they require extra byte(s) to encode the expected virtual machine behavior.

We always use the name bytecode to refer to the virtual code, i.e., the code understood by a virtual machine and not native code understood by a processor.

2.2 The Cog compiled method format

The bytecode, executable by the virtual machine, is saved in the heap - memory space reserved for object - in the form of compiled method[1]. A compiled method is an object encapsulating executable bytecode. In addition to the bytecode, a compiled method has, as shown in Figure 1:

- An object header (as every object in the system) to inform the virtual machine about basic properties such as its size, its class or its hash.

- A literal array that is used aside from the bytecode to fetch specific objects by the virtual machine to execute code.

- A compiled method header (specific to compiled method) to inform the virtual machine about basic properties such as its number of arguments or its number of temporaries.

- A source pointer to encode the way the IDE can get the method source code.

The format in memory of a compiled method is very specific. Common objects are word-aligned or byte-aligned on all their length. However, a compiled method is a mixed form of an array (its first fields in memory correspond to the compiled method header, encoded as a small integer, and its literals) and a byte array (its next fields correspond to its bytecode and its source pointer).

**Memory representation of
Compiled Method in 32 bits
with the new Memory Manager Spur**

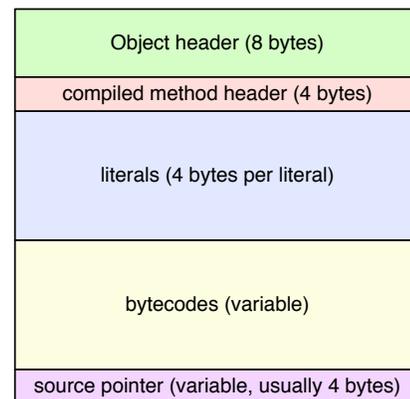| Object header (8 bytes) |
| compiled method header (4 bytes) |
| literals (4 bytes per literal) |
| bytecodes (variable) |
| source pointer (variable, usually 4 bytes) |

Figure 1. Memory representation of a Compiled method

Changes in the bytecode set design impacts both the memory zones for the compiled method header and for the bytecodes, but does not affect the compiled method object's header, its literal frame nor its source pointer encoding.

3. Challenges for a good bytecode set

3.1 Generic challenges

Any one who has to design a bytecode set faces some challenges. We enumerate the major ones we noticed in this subsection.

Platform-independent. Applications running on top of Cog are currently deployed on production on different operating systems: Linux, Mac OS X, Windows, iOS, Android and RISC OS and on different processors: ARM 32bits, Intel

x86 and Intel x86_64. Therefore, our bytecode set needs to be platform-independent, more specifically processor independent and operating system independent.

Compaction. To have the minimum memory footprint, a bytecode set needs to be able to encode all the compiled methods of system in the minimum number of bytes.

Easy decoding. The bytecode is mainly used by the interpreter to be executed, the JIT compiler to generate native code and in-image to analyze the compiled methods. Therefore, a good bytecode set needs to be easy to decode, decoding being used for analysis, interpretation and compilation.

Conflicts. One cannot have a very compact and very easy to decode bytecode set. Related work[9] has shown that a bytecode set can be compressed by 40% to 60% by sharing the bytecode in a Huffman table, but the bytecode becomes then harder to decode, which complicates the virtual machine interpreter, the JIT compiler and bytecode analysis. This extra difficulty also impacts performance (9% performance loss in their work). Therefore, one has to choose between easier decoding or extreme compaction. Our targets, such as the latest iPhone or the Raspberry pie have at least 256Mb of RAM. The Pharo memory footprint is usually around 20 Mb on production application. Therefore, we prefer to ease the decoding over compacting the bytecode. We want the new bytecode encoding to be at worst the same size as the old bytecode encoding but easier to decode.

Backward compatibility. Some frameworks and libraries may rely on the bytecode format to work. It includes compilers and virtual machines implementations that the designer of the new bytecode set has obviously to consider, but it also includes other frameworks such as serializers. A serializer typically reuses the bytecode encoding to serialize a compiled method. Changing the code set implies either to be backward compatible to have these tools working or fix all the frameworks and libraries.

3.2 Challenges specific to our goals

The runtime bytecode to bytecode optimizer we plan will perform classical dynamic language adaptive optimizations such as inlining based on type feeback[7] and bounds check elimination[2]. Here is a typical example:

```
MyDisplayClass>>example: anArray
      anArray do: [ :elem | self displayOnScreen: elem ].


Array(SequenceableCollection)>>do: aBlock
      1 to: self size do:
            [:index | aBlock value: (self at: index)]
```

Firstly, based on type-feed back the runtime optimizer notices that the argument of MyDisplayClass»example: is always an Array, and then inlines the message send to the array as well as the closure activation, adding a guard that triggers deoptimization if the assumption that anArray is an Array

is not valid any more. We represented the guard in pseudo Smalltalk code so the code is readable, but of course a guard is typically implemented in a very efficient way in native code.

```
MyDisplayClass>>OptimizedVersion1OfExample: anArray
      Guard: [ anArray class == Array
                    ifFalse: [ DynamicDeoptimization ] ].
      1 to: anArray size do: [ :index |
            self displayOnScreen: (anArray at: index) ].
```

The optimizer wants then to inline additional messages sent to anArray: size and at:. After inlining these two messages, it notices that index is always within the bounds of anArray because the to:do: selector sent on an integer enforces that the block argument is an integer between 1 and anArray size (This is typically inlined statically by the compiler). Therefore the optimizer edit the at: instruction to fetch the field of the objects at the index without checking that the index is within the bounds of anArray. In pseudo code, it would mean:

```
MyDisplayClass>>OptimizedVersion1OfExample: anArray
      Guard: [ anArray class == Array
                    ifFalse: [ DynamicDeoptimization ] ].
      1 to: anArray inlinedSize do: [ :index |
            self displayOnScreen: (anArray
                    inlinedNoBoundsCheckAt: index) ].
```

To be able to do this kind of optimizations, the bytecode set needs specific requirements and instructions that we detail in this subsection.

All methods should be inlinable. All methods and closure activations should be able to be inlined by the optimizer, including primitives. Inlining non primitive methods removes the overhead of a CPU call and allows the optimizer to have bigger portions of code to optimize. Inlining performance critical primitives allows the optimizer to perform additional critical optimizations such as bound check elimination. Due to inlining, optimized methods are bigger than regular methods: they may have jumps of thousands of instructions or thousands of literals. In addition, inlining a method may allow an object to access directly the instance variable of another object. Therefore, the bytecode needs to be able to encode compiled methods with:

- Inlined primitives
- Very large jump
- Very large number of literals
- Access to non receiver instance variable

Unsafe operations. Smalltalk primitives are type safe, which means that calling a primitive with inappropriate arguments will trigger a primitive failure but will not crash the execution. Therefore, each primitive needs to guarantee that the receiver and arguments have one of the expected

type before being performed. Guaranteeing the type of an object means additional CPU instructions to run. The runtime optimizer may want to encode unsafe operations, such as unchecked array access, if it can guarantee that the given primitives will not fail to avoid this type check overhead. The bytecode set needs to be extended to support a set of unsafe operations, which corresponds to inlined primitives that are optimized.

Extendable. While implementing more and more optimization passes in our optimizer, we may need to add extra unsafe operations or to encode new bytecode operations. The bytecode set needs to be easily extendable without compromising optimization opportunities and complicating the logic of the optimizer.

3.3 Current Bytecode issues

The current bytecode set could be improved compared to the generic requirements and is definitely not good enough for our requirements (specification available in Appendix A). In this subsection we describe the issues we noticed.

Argument bytecodes limits. The current bytecode set has limitations regarding the size of arguments. There is no pattern to indefinitely expand an argument bytecode. One typical example is the jump bytecode. This bytecode supports jump forward up to 1024 instructions, but no more. This is already an issue because several Pharo users ended up having a compilation error due to this problem. They had to fix it by editing their code, whereas a user should not need to understand nor see these compiler details. These issues make it impossible for some methods or closures to be inlined due to the code creating a jump too big for the encoding.

Few available bytes. One of our main concerns was the lack of available bytecodes. Only three bytecodes were available (not associated with an operation). This clearly limits the extensibility of the current bytecode set, which is required by our optimizer to easily add new instructions. We estimate that 10 available bytecodes are needed to be extended and manipulated in the future.

Primitive index implementation forbid inlining. Another issue is with the primitive index: by being encoded in the compiled method header, inlining primitives is not possible. However, the runtime optimizer needs to optimize critical primitives such as array access.

DoubleExtendedDoAnything bytecode. The old bytecode set had a bytecode named "double extended do anything" bytecode. This bytecode is able, with the proper encoding, to do almost any operation in the virtual machine. However, This bytecode creates extra difficulties in the JIT compiler to properly map the native code program counter to the bytecode program counter because you always need extra checks to know if it is a message send or something else. This bytecode does not fit with our definition of easy decoding. To

be easy to decode, a bytecode set should not encode several operations in the same byte, or if this is the case then encodes very similar operation, such as push new Array and pop into array, but not operations such as a push, a store and a message send in one bytecode.

Primitive index is split. The compiled method header shown in Figure 2 has a strange primitive index field: this field is split in 2 fields, the main part of the primitive number and the high bit. This is here for backward compatibility with the 16bit Squeak version. However, as none of the Cog users has been using a Smalltalk dialects in 16bit for over multiple decades, and other recent changes have broken this compatibility. This split is now useless whereas it complicates the decoding.

Old compiled method header

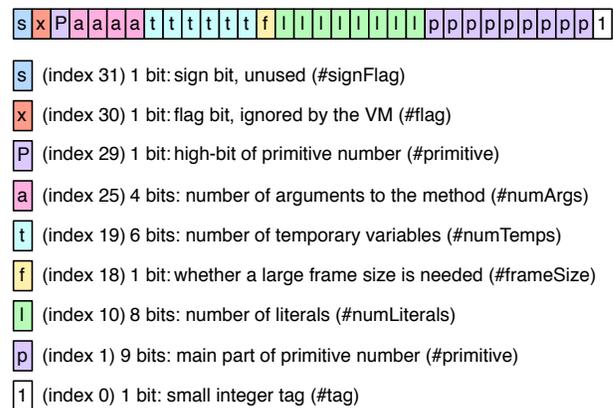|	
s	(index 31) 1 bit: sign bit, unused (#signFlag)
x	(index 30) 1 bit: flag bit, ignored by the VM (#flag)
P	(index 29) 1 bit: high-bit of primitive number (#primitive)
a	(index 25) 4 bits: number of arguments to the method (#numArgs)
t	(index 19) 6 bits: number of temporary variables (#numTemps)
f	(index 18) 1 bit: whether a large frame size is needed (#frameSize)
l	(index 10) 8 bits: number of literals (#numLiterals)
p	(index 1) 9 bits: main part of primitive number (#primitive)
1	(index 0) 1 bit: small integer tag (#tag)

Figure 2. The old compiled method header

Immediate objects waste memory. The bytecode set misses some compact bytecode for immediate objects. For example, encoding the SmallInteger 5 requires at least 5 bytes, 4 bytes to store 5 in the literal array and 1 to 3 bytes for the push literal bytecode. It is very simple to encode and decode immediate objects in the bytecode, as their virtual machine representation is also encoded with bytes, and it greatly improves the compactness of the bytecode.

Late addition of closure bytecodes. Real closures were added in the supported Smalltalk dialects after the old bytecode set design. There was not a lot of free space in the bytecode set, so the bytecode for BlockClosure creation was added in a way that the same bytecode is used to push nil on the stack and to allocate temporary slots on stack for the BlockClosure. This makes the decoding of a BlockClosure complex for the JIT compiler and the in-image analysis due to the difficulty to fetch the number of temporaries.

Outdated constraints. The current bytecode set has a string property: most instructions are 16 bits aligned. For example, instructions 0 to 15 are mapped to push receiver

variable and instructions 16 to 31 are mapped to push temporary. We asked the old bytecode set designer, Dan Ingalls, who is also the implementor of the original Smalltalk-80. It happens that Smalltalk-80 used to run on the Xerox D, and that this 16 bits alignment was there to easily dispatch the instructions on microcoded machines. As we do not run Pharo any more on this kind of machines, we removed this constraint for the bytecode set design.

4. New bytecode set features

In this section we describe the features of the new bytecode set, starting by the ones we added for our runtime optimizer to the other one that improved the bytecode set in general. Then we explain how we convert a Smalltalk image from the old bytecode set to the new one to validate the approach.

4.1 Adaptive optimization features

Extendable instructions. One of the most notable feature of the bytecode set (specification available in Appendix B) is the addition of an extension bytecode. This bytecode, as a prefix, extends the following byte by an index encoded in a byte. In the new bytecode set, each bytecode correspond to a single instruction. It is not possible, as for the "double extended do anything" bytecode of the old bytecode set, to encode different operation in a single byte. By being a prefix, the extension bytecode does not complicate the JIT compilation, as each bytecode still represent a single instruction, and an extension bytecode just requires to fetch the appropriate instruction in a fixed distance (2 bytes further).

This feature allows for example the compiler to generate jump forward bytecode to an infinite number of instructions. The jump forward bytecode is a single extended bytecode, therefore it can jump up to 255 instructions (255 being encoded in the argument byte). By being prefixed by one extension, it can encode a jump up to 65535 instructions. If it is prefixed by two extensions, it can encode a jump up to 16777215 instructions. As one can add as many extensions as one wants, each instruction accepting extensions can be extended to the infinite.

Example: Extended conditional jump instruction
(Numbers in hexadecimal)

Byte number	EF + next byte
Name	jumpFalse:
Bit values	11101111 iiiiiiii
Explanation	Pop value on stack and jump if the value is false by a distance of distance := iiiiiiii+(ExtensionB*256)

Byte number	E1 + next byte
Name	ExtensionB (ExtB)
Bit values	11100001 bbbbbbbb
Explanation	ExtensionB ExtB := ExtB*256+bbbbbbbb

bytecode sequence	Explanation
EF 12	jumpFalse: 18
E1 05 EF 12	jumpFalse: 50D 50D = (5*FF)+12
E1 AF E1 05 EF 12	jumpFalse: ADA7BC ADA7BC = (AF*FF*FF)+(5*FF)+18

Inlined primitives and unsafe operations. The new bytecode set moved the primitive index of a method from the compiled method header to the beginning of the compiled method's bytecode (these two zones are shown in Figure 1). The primitive call is now a double extended bytecode, with the two argument bytes encoding the primitive number. The first bit of the first byte argument determines if the primitive is inlined or not. If the primitive is not marked as inlined, it automatically fails if it is not the first bytecode of the method. This change does not slow down the virtual machine due to the different method caches. Inlined primitives cannot fail, therefore one needs to be very careful while inlining a primitive to properly handle inlined primitive failure and its fall back code with control flow operations (for example, a flag on top of stack can force the execution flow to jump on a specific branch that handles the fall back code of the inlined primitive).

Byte number	F9 + next byte (> 127) + next byte
Name	callPrimitive
Bit values	11111001 0iiiiiii jjjjjjjj
Explanation	call primitive at iiiiiii+(jjjjjjjj*FF) fails if not the first method's bytecode On success: triggers a hard return On failure: executes fallback code

Byte number	F9 + next byte (<= 127) + next byte
Name	callInlinedPrimitive
Bit values	11111001 1iiiiiii jjjjjjjj
Explanation	call inlined primitive at iiiiiii+(jjjjjjjj*FF) Inlined primitives cannot fail, may be unsafe, never triggers a return, may or may not push a value on stack

To encode the primitives, we noticed that we needed to support at least 1000 primitives and 1000 inlined primitives to support all the operations we might want to implement in the next decade. We could have made CallPrimitive a single extended bytecode taking an extension according to the extendable instruction model introduced in the last paragraph, but that would complicate the VM's determination of the primitive number and the primitive error code store since the extension, being optional, would make the sequence variable length. So we decided to make it a double extended bytecode. Therefore the new representation of primitive allows 32768 primitives and 32768 inlined primitives which is more than enough.

Some of the inlined primitives are unsafe, which means that if you send them on incorrect objects you may corrupt the memory or crash the virtual machine. However, the optimizer can guarantee at compile-time that this cannot happen. The first unsafe operations we want to support are direct access to the field of an objects to optimize indexable objects access by removing bounds checks.

Access to non receiver instance variable. Instance variable access in Smalltalk does not require any specific checks, because an object cannot access any instance variable of any other objects than itself, and at compilation time the structure of the receiver is well-known. In 2008, for our efficient BlockClosure implementation, we added an extra bytecode to quickly allocate an Array on the heap and quickly access to its fields to be able to efficiently share some variables between a closure and its enclosing environment. These operations were also unchecked both for performance and because you know the size of the array at compilation time. We reused these accessing bytecodes to access the instance variables of non receiver objects.

Extendable. The new bytecode set has 15 available bytecodes. This allows us to extend it easily and efficiently.

Maximum number of literals increased. The maximum number of literal extension is quite specific. To overcome the previous limitation, we needed two changes. Firstly, we reused the free bits in the compiled method header from the primitive index to encode more literals. Secondly, we allowed the literal access bytecode to take an extension, allowing the bytecode to encode access to literals at a position over 255 in the literal frame of the compiled method.

4.2 Generic features

Overall bytecode size. In the Pharo 3.0 release (version 30848), we installed the new compiler back-end for the new bytecode set support. The system reached then 75190 compiled methods (closures are included in compiled methods). As explained in Section 2.2, the new bytecode set impacts only the compiled method header and the bytecode zone of a compiled method. However, the compiled method header has a fixed size of a word (4 bytes in 32 bits, 8 bytes in 64bits). All the bytecode zones of the compiled methods in the image used to be encoded in 2,285,892 bytes with the old bytecode set, and are now encoded in 1,960,187 bytes. The new bytecode set is therefore more compact than the old one by 14.2%. However, the difference of around 325kb is hardly noticeable on typical Pharo application that are around 20Mb.

Immediate objects compaction. Immediate objects are now encoded in the bytecode instead of in the literal array. An object in the literal array always uses a word, which corresponds to 4 bytes in 32bits. Most immediate objects can be encoded in a single byte, such as integer from 0 to 255, or the 255 most common Characters. We use that property

to reduce the encoding size of most immediate objects. We added two single extended bytecode to support the encoding of SmallInteger and Character, the argument byte encoding the immediate object instead of a literal in the literal frame. These bytecodes support extension if they require extra bytes to be encoded. We have also reserved a double extended bytecode for SmallFloat, but we have not implemented it as our 64bits VM version is not stable enough. Depending on its memory manager and on its 64 bits or 32 bits form, Cog has from 1 to 3 immediate objects: SmallInteger, Character and SmallFloat. On the long term, all three will be always used, this variable number of immediate objects being due to the migration to the new Memory Manager Spur and to 64 bits.

Platform-independent. We have not introduced any platform dependent instruction, so the new bytecode set remains platform-independent.

Easy decoding. A massive improvement is related to the bytecode decoding. The bytecodes are now arranged with two simple rules:

- The bytecode are sorted by the number of bytes they need to encode their functionality: single byte bytecode are encoded between 0 and 223, extended bytecode are encoded between 224 and 248, double extended bytecode are encoded from 249 to 255.

- Within a number of byte categories, bytecodes are sorted by their functionalities: push bytecode are next to each other, send bytecodes are next to each others, ...

Easier closure decoding. A new bytecode was introduced to encode the number of temporaries in closures. We now do not need any more to walk over part of the closure bytecode to find out the number of temporaries.

About backward compatibility. Some mainstream language can hardly change their bytecode set. For example, when Java added the extra bytecode for invokeDynamic [15], the process to get it included in all virtual machines executing the Java bytecode was really tedious, and they didn't even have to edit all the bytecode compilers because this extra bytecode is provided for other languages on top of the JVM and not for Java itself. However, the Cog VM clients have two different production compilers. In addition, the most widely used serialization framework of our clients, Fuel[5], serialize the sources of a compiled method instead of its bytecode to support debugging and code edition of materialized methods in environments without decompilers. Therefore, by changing the virtual machine and the two compilers, we fixed most backward-compatibility issues.

4.3 Switching between bytecode sets to validate our approach

Difficulties with offline converters. One of the main concern, in a Smalltalk runtime, when implementing a new byte-

code set, is how to switch a snapshot from one bytecode set to another one. One solution is to implement an offline converter, that can translate the compiled method of an image from a bytecode set to another one. This solution has a big disadvantage: as long as the bytecode set implementation both image-side with the compiler back-end and VM-side with the interpreter and JIT front-end are not stable, the Smalltalk runtime crashes almost immediately at start-up and it is very hard to debug.

Multiple bytecode set support. When the Newspeak support was added to the Cog VM, the virtual machine was improved to support multiple bytecode set in the same run-time. In some Smalltalk virtual machines, such as the one of Smalltalk/X and Visual Age, the support of multiple byte-code set was implemented a while ago to be able to execute both the smalltalk bytecode and the Java bytecode. Claus Gittinger has worked on the Visual Age implementation and helped for this Cog VM improvement.

We decided to use the multiple bytecode set feature and we did not implement an offline converter. This approach has a big advantage, we can debug our bytecode compiler back end in Smalltalk on top of a VM that supports the old and the new bytecode set.

Encoding support for multiple bytecode sets. The support of multiple bytecode set is implemented part in the VM and part in the compiled method header format. The sign bit of the SmallInteger encoding the compiled method header is used to mark the method as using one or the other bytecode set, as shown in Figure 3. However, we discourage from using multiple bytecode set on top of the Cog VM for anything else than image conversion from a bytecode set to another or experiments. This is because different bytecode set have different limitations, and it may be that a method can be compiled in a bytecode set and not in another one. For example, the number of literals is limited to 255 in the old bytecode set and 65535 in the new one by assuming a CallPrimitive bytecode. This means that a method with 500 literals can be compiled n the new bytecode set but not in the old one.

Validation. Our bytecode set was validated by the implementation of a compiler back-end to generate the new byte-code out of the source code, the implementation of the inter-preter front-end and the implementation of the JIT compile front-end. The whole infrastructure is running with the new bytecode set with similar performance, validating the design. This was easy as the compiler, the interpreter and the JIT compiler were designed with an abstraction layer over the bytecode set to easily change it. Moving an image from one bytecode set to another one was also not very difficult with the multi-bytecode set support feature of the VM.

New hybrid compiled method header

Depending on the first bit, the new compiled method header is encoded in one of the two format described below.

0 (index 31) 1 bit: sign bit, 0 selects the old bytecode set (#signFlag)

x (index 30) 1 bit: flag bit, ignored by the VM (#flag)

P (index 29) 1 bit: high-bit of primitive number (#primitive)

a (index 25) 4 bits: number of arguments to the method (#numArgs)

t (index 19) 6 bits: number of temporary variables (#numTemps)

f (index 18) 1 bit: whether a large frame size is needed (#frameSize)

l (index 10) 8 bits: number of literals (#numLiterals)

p (index 1) 9 bits: main part of primitive number (#primitive)

1 (index 0) 1 bit: small integer tag (#tag)

1 (index 31) 1 bit: sign bit, 1 selects the new bytecode set(#signFlag)

x (index 30) 1 bit: flag bit, ignored by the VM (#flag)

(index 29) 1 bit: unused

a (index 25) 4 bits: number of arguments to the method (#numArgs)

t (index 19) 6 bits: number of temporary variables (#numTemps)

f (index 18) 1 bit: whether a large frame size is needed (#frameSize)

p (index 17) 1 bit: has primitive (#hasPrimitive)

l (index 1) 16 bits: number of literals (#numLiterals)

1 (index 0) 1 bit: small integer tag (#tag)

Figure 3. The new hybrid compiled method header

5. Discussion

5.1 Compaction of message sends

To improve the compactness of the bytecode set, some Smalltalk dialects as Visual Works have a specific byte-code for common message send. This bytecode is a single extended bytecode, the argument byte being the index of the common selector in a common selector array. This way, sends with common selectors are most of the time encoded in 2 bytes instead of 5 bytes because the selector is not in the literal frame, at the cost of an indirection array.

A similar feature is present in the old and new bytecode set. A list of 16 arithmetic selectors and 16 common selec-tor have a quick encoding form. Our representation could have been extended to have this single extended bytecode, so we would have many more selectors encoded in a com-pact way. However, this approach has issues. For instance, common selector are not always the same. Therefore, on a regular basis, the team maintaining the programming lan-guage needs to update this common selector array depend-

ing on the new common selectors. This is problematic for some tools, such as some serializer which needs to version the serialized methods to know what common selector array it needs to use for serialization and materialization. We kept the exact same compact selectors from the old bytecode set to the new one to avoid this kind of issue. In addition, we had already reached our compaction goals, so we didn't need to add extra complexity for more compaction.

5.2 Register-based bytecode set

Our bytecode set is stack based. This design date from the time where there were both register based and stack based CPU. However, modern common CPU are now all register-based, so one can wonder if a register-based bytecode set may not be better. At Google, the Dalvik VM team chose to rely on a register-based bytecode for their Android applications[4].

We didn't move to a register-based bytecode for different reasons. The main reason is that to generate register-based bytecodes, you need to give to the compiler the number of available registers. This generates extra complexity when running the application on different platforms which may have a different number of general purpose registers. For example, on our targets, intel x86 has 8 general purpose registers whereas intel x86_64 has 16 general purpose registers. Moving from one architecture to another one requires to loose performance by using less registers than available, to recompile all the code base with the new number of general purpose registers or to have a non trivial mapping from a limited number of register to an extended one in the JIT compiler.

5.3 Threaded FFI

The Cog VM now starts to support a threaded foreign function interface (FFI) to be able to call C function without blocking the virtual machine execution. The current process implementation requires the user to fetch a global variable and to execute a message send to access the current process. This implementation was good enough but does not scale for the new multithreaded FFI. Therefore, we are considering an extra bytecode to have a direct access on the active process.

We are not sure exactly if this bytecode will be used and how. The original idea was to extend the language with an extra reserved keyword, *thisProcess*, which will push on the stack the active process the same way the active context is pushed with the reserved keyword *thisContext*. But adding an extra reserved keyword adds extra constraints to the language, so we need to study other solutions.

6. Related Works

6.1 Bytecode and primitive operations

The main difference between most bytecode sets and a Smalltalk bytecode set is that we do not encode any primitive operations in the bytecode. For example, addition is implemented as the message send named "+", and can be sent to any Object in the system. Integer addition will be performed only if the receiver is a SmallInteger. However, there are no integer addition encoded in the bytecode set that requires the operands to be SmallIntegers. Classic bytecode sets, such as the Java one[12], encodes primitive operations. For example the operation iadd in Java expects both operands to be Integers. All Smalltalk instructions expect objects of any type.

With the new bytecode set, we added encoding for inlined primitives at bytecode level, which includes typed operations. However, this encoding will only be used only by the runtime optimizer or very specific low-level tool such as an ABI compiler, and is not present by default in the Smalltalk semantics.

6.2 Bytecode set with superoperators

Several teams designed a bytecode set with superoperators to optimize the interpreter speed[3, 8, 13]. Superoperators are virtual machine operations automatically synthesized from smaller operations to avoid costly per-operation overheads. This is typically done statically at compile-time. The compiler detects common bytecode patterns and then extends the bytecode set with superoperators performing the common patterns. For an interpreter, this technique drastically improves the performance by reducing the overhead of bytecode fetching. However, in our case we would need to adapt many in-image tools, as well as the interpreter and the JIT to support it. In addition, this optimization speeds up only the interpreter, which is in most case not performance critical as our VM heavily relies on the JIT for performance. So we concluded that this optimization would cost too much time to implement compared to its benefits.

6.3 Bytecode extensions

To add infinite argument values to our argument bytecodes, we added the extension prefix bytecodes in the new bytecode set as explained in Section 4.1. We designed the extension this way to be able to encode one instruction per bytecode (only the argument is variable) and because we needed a limited number of different instructions.

Other systems have needed many more instructions, typically more than 255 which is the maximum number of different instructions you can encode in a byte. An example is the Z80 CPU bytecode[16] which needed many graphical instructions (the Z80 is the processor of the GameBoy and the SuperNintendo). In this case, they decided to use bytecode prefix to indicate to the processor to fetch the next bytecode in another bytecode table encoding other instructions. This is a convenient trick when you want an important number of instructions, but as we have much less different instructions that 255 in our virtual machine, we felt the extra complexity of this encoding didn't worth it.

6.4 Visual Works bytecode set

The Cog virtual machine is very different from the Visual-Works VM. However, they both run Smalltalk runtime, so the comparison is interesting.

BlockClosure bytecodes. One difference is that Visual-Works has another model to activate BlockClosure. When a BlockClosure is activated, it is present in the receiver slot of the context, because the BlockClosure received the block activation message. We instead have two fields in a context, one for the method activation's receiver and one for the closure. VisualWorks' model seems very pure and cleaner for the user. However, in term of the bytecode set, it means that accessing the receiver or the receiver fields in method has to be encoded differently between a method and a BlockClosure, because in one case it access the receiver of the active context whereas in the other case it access a variable from the lexically enclosed environment. In our implementation, the receiver slot of a BlockClosure context has the receiver of the homeContext, therefore accessing the receiver and the receiver's fields is the same whichever activation you are. Both implementation has pros and cons. We considered the other approach, but we preferred to simplify the virtual machine implementation at the cost of complicating a bit the model for the end user.

Loop encoding. Another difference is the support in the VisualWorks bytecode set to encode the beginning of a loop. This is convenient to be able in the JIT to generate native code in a single pass. However, loops are not very frequent in Smalltalk dialects: 5% of the compiled methods have a loop in the Pharo 3.0 release image and adding in our JIT implementation some code to handle loops didn't increase a lot the JIT complexity, therefore we preferred no to introduce this bytecode.

Common selector array. VisualWorks features a common selector array as it is described in Section 5.1. We didn't choose to add that in the Cog VM as explained in this subsection.

Non immediate entries in machine code. VisualWorks has support for non-immediate entries in machine code methods. This means that if you can guarantee that an object is not an immediate object, i.e., not a Character, a SmallInteger nor a SmallFloat, you can speed up monomorphic sends sites by targeting the method after the immediate entry. This is encoded in the bytecode with a specific send targeting the non-immediate entry in the machine code. This implementation leads to lots of complication, such as if the user change a temporary variable in the debugger leading to an immediate object being sent a message send with a non immediate entry, as well as VM-side complication as we needed to add an extra entry in the native methods, aside from the class check entry and unchecked entry.

7. Future work and Conclusion

The reason why we needed a new bytecode set was to be able to implement a runtime optimizer in the JIT compiler to optimize methods in a bytecode to bytecode fashion. This optimizer will therefore have some similarities with Soot[14], the Java bytecode to bytecode optimizer, but will be used at runtime and not statically. As the bytecode set is now ready, one needs to design and implement the runtime optimizer.

Reportedly, it is very difficult to produce correct marshaling code for FFI calls on some architectures, especially on x86_64bits[10], which is now very common. Generating marshaling code in the virtual machine is tedious, as debugging the virtual machine has always been much more complex than debugging high level languages, even with very good dedicated tools. One could implement the marshaling code of FFI calls by generating compiled methods encoding low level instructions with the unsafe operations of the new bytecode set in order to simplify this process.

In this paper, we showed how we designed a bytecode set suitable for runtime bytecode to bytecode optimizations for the Cog VM and its Smalltalk clients.

This new bytecode set encodes an extendable set of unsafe operations to provide information to the JIT compiler for producing better machine code, encodes primitives in a way the optimizer can inline them, has many available bytecodes to be easily extended and fixes the old bytecode set issues we described.

References

[1] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[2] R. Bodík, R. Gupta, and V. Sarkar. Abcd: Eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 321–333, New York, NY, USA, 2000.

[3] P. Bonzini. Implementing a high-performance smalltalk interpreter with genbc and genvm.

[4] D. Bornstein. Dalvik virtual machine internal talk, google i/o, 2008.

[5] M. Dias, M. Martinez Peck, S. Ducasse, and G. Arévalo. Fuel: A fast general-purpose object graph serializer. *Journal of Software: Practice and Experience*, 2012.

[6] ECMA. *ECMA-334: C# Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, dec 2001.

[7] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 326–336, New York, NY, USA, 1994.

[8] A. E. Kevin Casey and D. Gregg. Optimizations for a java interpreter using instruction set enhancement. Technical report, Trinity College Dublin, sept 2005.

[9] M. Latendresse and M. Feeley. Generation of fast interpreters for huffman compressed bytecode. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, IVME '03, pages 32–40, New York, NY, USA, 2003. ACM.

[10] A. J. Michael Matz, Jan Hubicka and M. Mitchell. System v application binary interface amd64 architecture processor supplement.

[11] E. Miranda. Cog blog. speeding up croquet and squeak with a new open-source vm from qwaq, 2008.

[12] Oracle. The java virtual machine specification, java se 8 edition.

[13] Proebsting and T. A. Optimizing an ansi c interpreter with superoperators. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 1995, pages 322–332, New York, NY, USA, 1995. ACM.

[14] P. L. Raja Vall'ee-Rai, P. P. Clark Verbrugge, and F. Qian. Soot (poster session): a java bytecode optimization and annotation framework. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (Addendum)*, OOPSLA 2000, page 113–114, New York, NY, USA, 2000. ACM.

[15] J. R. Rose. Bytecodes meet combinators: Invokedynamic on the jvm. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, pages 2:1–2:11, New York, NY, USA, 2009.

[16] ZiLOG. Z80 cpu user's manual.

A. The old bytecode set

Old Bytecode set

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Push Receiver Variable 0 (0)	Push Receiver Variable 1 (1)	Push Receiver Variable 2 (2)	Push Receiver Variable 3 (3)	Push Receiver Variable 4 (4)	Push Receiver Variable 5 (5)	Push Receiver Variable 6 (6)	Push Receiver Variable 7 (7)	Push Receiver Variable 8 (8)	Push Receiver Variable 9 (9)	Push Receiver Variable 10 (10)	Push Receiver Variable 11 (11)	Push Receiver Variable 12 (12)	Push Receiver Variable 13 (13)	Push Receiver Variable 14 (14)	Push Receiver Variable 15 (15)
Push Temp 0 (16)	Push Temp 1 (17)	Push Temp 2 (18)	Push Temp 3 (19)	Push Temp 4 (20)	Push Temp 5 (21)	Push Temp 6 (22)	Push Temp 7 (23)	Push Temp 8 (24)	Push Temp 9 (25)	Push Temp 10 (26)	Push Temp 11 (27)	Push Temp 12 (28)	Push Temp 13 (29)	Push Temp 14 (30)	Push Temp 15 (31)
Push Literal Constant 0 (32)	Push Literal Constant 1 (33)	Push Literal Constant 2 (34)	Push Literal Constant 3 (35)	Push Literal Constant 4 (36)	Push Literal Constant 5 (37)	Push Literal Constant 6 (38)	Push Literal Constant 7 (39)	Push Literal Constant 8 (40)	Push Literal Constant 9 (41)	Push Literal Constant 10 (42)	Push Literal Constant 11 (43)	Push Literal Constant 12 (44)	Push Literal Constant 13 (45)	Push Literal Constant 14 (46)	Push Literal Constant 15 (47)
Push Literal Constant 16 (48)	Push Literal Constant 17 (49)	Push Literal Constant 18 (50)	Push Literal Constant 19 (51)	Push Literal Constant 20 (52)	Push Literal Constant 21 (53)	Push Literal Constant 22 (54)	Push Literal Constant 23 (55)	Push Literal Constant 24 (56)	Push Literal Constant 25 (57)	Push Literal Constant 26 (58)	Push Literal Constant 27 (59)	Push Literal Constant 28 (60)	Push Literal Constant 29 (61)	Push Literal Constant 30 (62)	Push Literal Constant 31 (63)
Push Literal Variable 0 (64)	Push Literal Variable 1 (65)	Push Literal Variable 2 (66)	Push Literal Variable 3 (67)	Push Literal Variable 4 (68)	Push Literal Variable 5 (69)	Push Literal Variable 6 (70)	Push Literal Variable 7 (71)	Push Literal Variable 8 (72)	Push Literal Variable 9 (73)	Push Literal Variable 10 (74)	Push Literal Variable 11 (75)	Push Literal Variable 12 (76)	Push Literal Variable 13 (77)	Push Literal Variable 14 (78)	Push Literal Variable 15 (79)
Push Literal Variable 16 (80)	Push Literal Variable 17 (81)	Push Literal Variable 18 (82)	Push Literal Variable 19 (83)	Push Literal Variable 20 (84)	Push Literal Variable 21 (85)	Push Literal Variable 22 (86)	Push Literal Variable 23 (87)	Push Literal Variable 24 (88)	Push Literal Variable 25 (89)	Push Literal Variable 26 (90)	Push Literal Variable 27 (91)	Push Literal Variable 28 (92)	Push Literal Variable 29 (93)	Push Literal Variable 30 (94)	Push Literal Variable 31 (95)
PopInto Receiver Variable 0 (96)	PopInto Receiver Variable 1 (97)	PopInto Receiver Variable 2 (98)	PopInto Receiver Variable 3 (99)	PopInto Receiver Variable 4 (100)	PopInto Receiver Variable 5 (101)	PopInto Receiver Variable 6 (102)	PopInto Receiver Variable 7 (103)	PopInto Temp 0 (104)	PopInto Temp 1 (105)	PopInto Temp 2 (106)	PopInto Temp 3 (107)	PopInto Temp 4 (108)	PopInto Temp 5 (109)	PopInto Temp 6 (110)	PopInto Temp 7 (111)
Push Receiver (112)	Push true (113)	Push false (114)	Push nil (115)	Push -1 (116)	Push 0 (117)	Push 1 (118)	Push 2 (119)	Return receiver (120)	Return true (121)	Return false (122)	Return nil (123)	Return (124)	Block Return (125)	(126)	(127)
(0) Push LongForm (128)	(0) Store LongForm (129)	(0) PopInto LongForm (130)	(1) Single Extended send (131)	(3) Double extended (132)	Super send literal selector (133)	(2) Second extended send (134)	Pop Stack top (135)	Duplicate Stack top (136)	Push this Context (137)	(4) Push or PopInto Array (138)	(139)	(5) Push Temp in temp Vector (140)	(5) Store Temp in temp Vector (141)	(5) Pop into Temp in temp Vector (142)	(6) Push Closure (143)
Jump 1 (144)	Jump 2 (145)	Jump 3 (146)	Jump 4 (147)	Jump 5 (148)	Jump 6 (149)	Jump 7 (150)	Jump 8 (151)	JumpFalse 1 (152)	JumpFalse 2 (153)	JumpFalse 3 (154)	JumpFalse 4 (155)	JumpFalse 5 (156)	JumpFalse 6 (157)	JumpFalse 7 (158)	JumpFalse 8 (159)
Long Jump -1024 to -769 (160)	Long Jump -768 to -513 (161)	Long Jump -512 to -257 (162)	Long Jump -256 to -1 (163)	Long Jump 0 to 255 (164)	Long Jump 256 to 512 (165)	Long Jump 513 to 768 (166)	Long Jump 768 to 1024 (167)	Long JumpTrue 0 to 255 (168)	Long JumpTrue 256 to 511 (169)	Long JumpTrue 512 to 768 (170)	Long JumpTrue 769 to 1024 (171)	Long JumpFalse 0 to 255 (172)	Long JumpFalse 256 to 511 (173)	Long JumpFalse 512 to 768 (174)	Long JumpFalse 769 to 1024 (175)
+ (176)	- (177)	< (178)	> (179)	<= (180)	>= (181)	= (182)	~= (183)	* (184)	/ (185)	\\ (186)	@ (187)	bitShift: (188)	// (189)	bitAnd: (190)	bitOr: (191)
at: (192)	at:put: (193)	size (194)	next (195)	nextPut: (196)	atEnd (197)	== (198)	class (199)	Block Copy: (200)	value (201)	value: (202)	do: (203)	new (204)	new: (205)	(Point) x (206)	(Point) y (207)
Send 0 args selector at 0 (208)	Send 0 args selector at 1 (209)	Send 0 args selector at 2 (210)	Send 0 args selector at 3 (211)	Send 0 args selector at 4 (212)	Send 0 args selector at 5 (213)	Send 0 args selector at 6 (214)	Send 0 args selector at 7 (215)	Send 0 args selector at 8 (216)	Send 0 args selector at 9 (217)	Send 0 args selector at 10 (218)	Send 0 args selector at 11 (219)	Send 0 args selector at 12 (220)	Send 0 args selector at 13 (221)	Send 0 args selector at 14 (222)	Send 0 args selector at 15 (223)
Send 1 args selector at 0 (224)	Send 1 args selector at 1 (225)	Send 1 args selector at 2 (226)	Send 1 args selector at 3 (227)	Send 1 args selector at 4 (228)	Send 1 args selector at 5 (229)	Send 1 args selector at 6 (230)	Send 1 args selector at 7 (231)	Send 1 args selector at 8 (232)	Send 1 args selector at 9 (233)	Send 1 args selector at 10 (234)	Send 1 args selector at 11 (235)	Send 1 args selector at 12 (236)	Send 1 args selector at 13 (237)	Send 1 args selector at 14 (238)	Send 1 args selector at 15 (239)
Send 2 args selector at 0 (240)	Send 2 args selector at 1 (241)	Send 2 args selector at 2 (242)	Send 2 args selector at 3 (243)	Send 2 args selector at 4 (244)	Send 2 args selector at 5 (245)	Send 2 args selector at 6 (246)	Send 2 args selector at 7 (247)	Send 2 args selector at 8 (248)	Send 2 args selector at 9 (249)	Send 2 args selector at 10 (250)	Send 2 args selector at 11 (251)	Send 2 args selector at 12 (252)	Send 2 args selector at 13 (253)	Send 2 args selector at 14 (254)	Send 2 args selector at 15 (255)

(0) jjkkkkkkk, jj = Receiver Variable, Temporary Location, Literal Constant / Illegal, Literal Variable, kkkkkk the index
(1) 32 first lit as selector and up to 7 args
(2) 64 first lit as selector and up to 3 args
(3) iiiiiii kkkkkkkkk, iii = Send, Send Super, Push Receiver Variable, Push Literal Constant, Push Literal Variable, Store Receiver Variable, Store-Pop Receiver Variable, Store Literal Variable, index = kkkkkkkk, for sends jjjjjj = numArgs
(4) jkkkkkkk, Push (Array new: kkkkkkkk) (j = 0) or Pop kkkkkkkk elements into: (Array new: kkkkkkkk) (j = 1)
(5) kkkkkkkk jjjjjj, tempAt kkkkkkkk in tempVectAt jjjjjj
(6) lllkkkk jjjjjj iiiiiii, Push Closure Num Args kkkk Num Copied llll BlockSize jjjjjjiiiiiii

Figure 4. The old bytecode set

B. The new bytecode set

New Bytecode set

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Push Receiver Variable 0 (0)	Push Receiver Variable 1 (1)	Push Receiver Variable 2 (2)	Push Receiver Variable 3 (3)	Push Receiver Variable 4 (4)	Push Receiver Variable 5 (5)	Push Receiver Variable 6 (6)	Push Receiver Variable 7 (7)	Push Receiver Variable 8 (8)	Push Receiver Variable 9 (9)	Push Receiver Variable 10 (10)	Push Receiver Variable 11 (11)	Push Receiver Variable 12 (12)	Push Receiver Variable 13 (13)	Push Receiver Variable 14 (14)	Push Receiver Variable 15 (15)
Push Literal Variable 0 (16)	Push Literal Variable 1 (17)	Push Literal Variable 2 (18)	Push Literal Variable 3 (19)	Push Literal Variable 4 (20)	Push Literal Variable 5 (21)	Push Literal Variable 6 (22)	Push Literal Variable 7 (23)	Push Literal Variable 8 (24)	Push Literal Variable 9 (25)	Push Literal Variable 10 (26)	Push Literal Variable 11 (27)	Push Literal Variable 12 (28)	Push Literal Variable 13 (29)	Push Literal Variable 14 (30)	Push Literal Variable 15 (31)
Push Literal Constant 0 (32)	Push Literal Constant 1 (33)	Push Literal Constant 2 (34)	Push Literal Constant 3 (35)	Push Literal Constant 4 (36)	Push Literal Constant 5 (37)	Push Literal Constant 6 (38)	Push Literal Constant 7 (39)	Push Literal Constant 8 (40)	Push Literal Constant 9 (41)	Push Literal Constant 10 (42)	Push Literal Constant 11 (43)	Push Literal Constant 12 (44)	Push Literal Constant 13 (45)	Push Literal Constant 14 (46)	Push Literal Constant 15 (47)
Push Literal Constant 16 (48)	Push Literal Constant 17 (49)	Push Literal Constant 18 (50)	Push Literal Constant 19 (51)	Push Literal Constant 20 (52)	Push Literal Constant 21 (53)	Push Literal Constant 22 (54)	Push Literal Constant 23 (55)	Push Literal Constant 24 (56)	Push Literal Constant 25 (57)	Push Literal Constant 26 (58)	Push Literal Constant 27 (59)	Push Literal Constant 28 (60)	Push Literal Constant 29 (61)	Push Literal Constant 30 (62)	Push Literal Constant 31 (63)
Push Temp 0 (64)	Push Temp 1 (65)	Push Temp 2 (66)	Push Temp 3 (67)	Push Temp 4 (68)	Push Temp 5 (69)	Push Temp 6 (70)	Push Temp 7 (71)	Push Temp 8 (72)	Push Temp 9 (73)	Push Temp 10 (74)	Push Temp 11 (75)	Push Receiver (76)	Push true (77)	Push false (78)	Push nil (79)
Push 0 (80)	Push 1 (81)	Push context Push process (82)	Duplicate Stack Top (83)	(84)	(85)	(86)	(87)	Return Receiver (88)	Return true (89)	Return false (90)	Return nil (91)	Return Top (92)	BlockReturn Nil (93)	BlockReturn Top (94)	Nop (95)
+ (96)	- (97)	< (98)	> (99)	<= (100)	>= (101)	= (102)	~= (103)	* (104)	/ (105)	\\ (106)	@ (107)	bitShift: (108)	// (109)	bitAnd: (110)	bitOr: (111)
at: (112)	at:put: (113)	size (114)	next (115)	nextPut: (116)	atEnd (117)	== (118)	== (119)	class (120)	value (121)	value: (122)	do: (123)	new (124)	new: (125)	(Point) x (126)	(Point) y (127)
Send 0 args selector at 0 (128)	Send 0 args selector at 1 (129)	Send 0 args selector at 2 (130)	Send 0 args selector at 3 (131)	Send 0 args selector at 4 (132)	Send 0 args selector at 5 (133)	Send 0 args selector at 6 (134)	Send 0 args selector at 7 (135)	Send 0 args selector at 8 (136)	Send 0 args selector at 9 (137)	Send 0 args selector at 10 (138)	Send 0 args selector at 11 (139)	Send 0 args selector at 12 (140)	Send 0 args selector at 13 (141)	Send 0 args selector at 14 (142)	Send 0 args selector at 15 (143)
Send 1 args selector at 0 (144)	Send 1 args selector at 1 (145)	Send 1 args selector at 2 (146)	Send 1 args selector at 3 (147)	Send 1 args selector at 4 (148)	Send 1 args selector at 5 (149)	Send 1 args selector at 6 (150)	Send 1 args selector at 7 (151)	Send 1 args selector at 8 (152)	Send 1 args selector at 9 (153)	Send 1 args selector at 10 (154)	Send 1 args selector at 11 (155)	Send 1 args selector at 12 (156)	Send 1 args selector at 13 (157)	Send 1 args selector at 14 (158)	Send 1 args selector at 15 (159)
Send 2 args selector at 0 (160)	Send 2 args selector at 1 (161)	Send 2 args selector at 2 (162)	Send 2 args selector at 3 (163)	Send 2 args selector at 4 (164)	Send 2 args selector at 5 (165)	Send 2 args selector at 6 (166)	Send 2 args selector at 7 (167)	Send 2 args selector at 8 (168)	Send 2 args selector at 9 (169)	Send 2 args selector at 10 (170)	Send 2 args selector at 11 (171)	Send 2 args selector at 12 (172)	Send 2 args selector at 13 (173)	Send 2 args selector at 14 (174)	Send 2 args selector at 15 (175)
Jump 1 (176)	Jump 2 (177)	Jump 3 (178)	Jump 4 (179)	Jump 5 (180)	Jump 6 (181)	Jump 7 (182)	Jump 8 (183)	JumpTrue 1 (184)	JumpTrue 2 (185)	JumpTrue 3 (186)	JumpTrue 4 (187)	JumpTrue 5 (188)	JumpTrue 6 (189)	JumpTrue 7 (190)	JumpTrue 8 (191)
JumpFalse 1 (192)	JumpFalse 2 (193)	JumpFalse 3 (194)	JumpFalse 4 (195)	JumpFalse 5 (196)	JumpFalse 6 (197)	JumpFalse 7 (198)	JumpFalse 8 (199)	PopInto Receiver Variable 0 (200)	PopInto Receiver Variable 1 (201)	PopInto Receiver Variable 2 (202)	PopInto Receiver Variable 3 (203)	PopInto Receiver Variable 4 (204)	PopInto Receiver Variable 5 (205)	PopInto Receiver Variable 6 (206)	PopInto Receiver Variable 7 (207)
PopInto Temp 0 (208)	PopInto Temp 1 (209)	PopInto Temp 2 (210)	PopInto Temp 3 (211)	PopInto Temp 4 (212)	PopInto Temp 5 (213)	PopInto Temp 6 (214)	PopInto Temp 7 (215)	(216)	(217)	(218)	(219)	(220)	(221)	(222)	(223)
Extension A (224)	Extension B (225)	Extended Push Receiver Variable (226)	Extended Push Literal Variable (227)	Extended Push Literal (228)	Extended Push Temporary Variable (229)	Push N Closure Temps (230)	Push Integer (231)	Pop Stack Top (232)	Push or PopInto Array (233)	Extended Send Literal selector (234)	Send To superclass Literal selector (235)	Trap on Behavior/ Array of Behavior (236)	Extended Jump (237)	Extended Pop And Jump True (238)	Extended Pop And Jump False (239)
Extended PopInto Receiver Variable (240)	Extended PopInto Literal Variable (241)	Extended PopInto Temporary Variable (242)	Extended Store Receiver Variable (243)	Extended Store Literal Variable (244)	Extended Store Temporary Variable (245)	(246)	(247)	Push Character (248)	Call Primitive / Call inlined Primitive (249)	Reserved for Push Float (250)	PushClosure Copy (251)	Push Temp in temp Vector (252)	Store Temp in temp Vector (253)	Pop into Temp in temp Vector (254)	(255)

Figure 5. The new bytecode set