# [DRAFT/WIP] The Pharo Virtual Machine Explained

The Pharo team

July 5, 2024

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

# Contents

**Illustrations** **iii**

**1 Preamble** **1**

**2 Object Representation** **3**
2.1 Background . . . 3
2.2 Object Layout . . . 6
2.3 Immediate Objects . . . 9
2.4 Object Header . . . 13
2.5 Conclusion . . . 16
2.6 References . . . 16

**3 Methods, Bytecode and Primitives** **19**
3.1 Compiled Methods . . . 19
3.2 Stack Bytecode and the Sista Bytecode Set Overview . . . 22
3.3 Primitive Methods . . . 24
3.4 Bytecode Encoding and Optimizations . . . 25
3.5 The Sista Bytecode . . . 28

**4 Calling conventions** **29**
4.1 Passing arguments . . . 29
4.2 Returning . . . 29
4.3 Shared state . . . 31

**5 The Spur Memory Manager Overview** **33**
5.1 Spur features . . . 34
5.2 Memory Structure Overview . . . 34
5.3 Memory Growing and Segments . . . 35
5.4 Memory Initialization . . . 35
5.5 Spur Generational Garbage Collection . . . 36
5.6 The Stack . . . 36
5.7 Conclusion . . . 36
5.8 The New Space . . . 37
5.9 New Space Memory Layout . . . 37
5.10 The Scavenger . . . 37
5.11 Example of a Scavenger pass . . . 39

5.12 to be continued . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.13 The Old Space . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.14 The Free List . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.15 Free cells in memory . . . . . . . . . . . . . . . . . . . . . . . . 41
5.16 Free list . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
5.17 Free Table . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

**6 Weak Objects** **43**
6.1 Strong and Weak references . . . . . . . . . . . . . . . . . . . . 43
6.2 Strong and Weak Objects . . . . . . . . . . . . . . . . . . . . . 43
6.3 Weak reference collection during scavenging . . . . . . . . . . . 44
6.4 Weak list structure . . . . . . . . . . . . . . . . . . . . . . . . . 45

**7 Ephemerons** **47**

**8 JIT Vocabulary** **49**

**9 Looking at Stack Structure** **51**
9.1 Stack/Context Terminology . . . . . . . . . . . . . . . . . . . . 51
9.2 Cog Stack Structure . . . . . . . . . . . . . . . . . . . . . . . . 51

**10 Adding Static Methods** **53**
10.1 Bytecode table . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
10.2 About method execution . . . . . . . . . . . . . . . . . . . . . 54
10.3 Study 128 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
10.4 A first version of sendStaticLiteralMethod . . . . . . . . . . . . 55
10.5 Compiling the VM . . . . . . . . . . . . . . . . . . . . . . . . . 56
10.6 Getting a compiled method . . . . . . . . . . . . . . . . . . . . 56
10.7 Fixing some Pharo logic . . . . . . . . . . . . . . . . . . . . . . 58
10.8 Extending the IRBuilder . . . . . . . . . . . . . . . . . . . . . . 58
10.9 Study the Translator . . . . . . . . . . . . . . . . . . . . . . . . 59
10.10 Translator extension . . . . . . . . . . . . . . . . . . . . . . . . 60
10.11 Testing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
10.12 The case of recursion . . . . . . . . . . . . . . . . . . . . . . . 61
10.13 Better sendStaticLiteralMethodBytecode . . . . . . . . . . . . . 62
10.14 Bench . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
10.15 Limits and conclusion . . . . . . . . . . . . . . . . . . . . . . . 65

**Bibliography** **67**

# Illustrations

2-1 Word size and Alignment on 32 and 64 bits architectures. . . . . . . . . . . 4
2-2 16rFEDCBA987654321 in 64-bits Little and Big-Endian. . . . . . . . . . . . . 6
2-3 Object Layout and Alignment on 32 and 64 bits Architectures. . . . . . . . . 8
2-4 References to heap-allocated objects are pointers to an object’s base header. 9
2-5 64 bits immediate objects. . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2-6 32 bits immediate objects. . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2-7 On 64 bits, SmallInteger are encoded on 61 bits. . . . . . . . . . . . . . . . 12
2-8 64 bits `SmallFloat` immediate. . . . . . . . . . . . . . . . . . . . . . . . 13
2-9 Base Object Header. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2-10 Extracting the number of data slots in an object . . . . . . . . . . . . . . 15

3-1 Source code with several literals . . . . . . . . . . . . . . . . . . . . . . 21
3-2 Pseudo-bytecode performing 2+7 . . . . . . . . . . . . . . . . . . . . . . . 23
3-3 The SmallInteger addition method is a primitive method . . . . . . . . . . . 24
5-1 Memory Map: a new space followed by an old space. . . . . . . . . . . . . . 34
5-2 A young object is an object located below the newSpaceLimit . . . . . . . . 35
5-3 The NewSpace structure composed of the eden, and two past and future regions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5-4 The Old Space with two segments. . . . . . . . . . . . . . . . . . . . . . . . 40
5-5 Free cells of size 5 in the memory. . . . . . . . . . . . . . . . . . . . . . . 41
5-6 Free list view is a table of linked-lists of free objects. . . . . . . . . . . . 42

# CHAPTER 1
# Preamble

Most of the design of the Pharo Virtual Machine has been made by E. Miranda and we are very grateful for this. On this basis, we are working on improving its design and create new generation virtual machine. This book is the result of an effort to understand, document, and structure the knowledge of the internals of the Pharo VM. Our objective is to increase the amount of people who understand and improve it in the future. This work re-used some previously existing materials such as blog posts of C. Béra.

Now, it may happen that we wrote something wrong. If you spot something wrong, please let us know.

Readers may also be interested in other booklets. The booklet on concurrent programming in Pharo also describes some VM primitives and provide some specific information. A booklet on the Pharo compiler is also a work in progress.

*Acknowledgements.* This work is supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020. The work is supported by I-Site ERC-Generator Multi project 2018-2022. We gratefully acknowledge the financial support of the Métropole Européenne de Lille. This work is also supported by the Action Exploratoire Alamvic led by G. Polito and S. Ducasse.

CHAPTER 2

# Object Representation

Before delving inside the internals of the VM execution, it is important to understand the data it manipulates, in our case, Pharo objects. This chapter presents in-depth how objects are represented in memory. This will allow us to understand in further chapters how objects are created and mutated, how dynamic type checks are performed, and the different memory optimizations employed.

The Pharo VM uses since a couple of years the Spur memory model, designed and implemented by Eliot Miranda for the OpenSmalltalk VM [2]. This model greatly improved the Garbage Collector (GC) and the complexity of JIT-compiled machine code.

## 2.1 Background

This section sets up some terminology necessary to understand this chapter, such as *word*, *nibble*, or *alignment*. It is important to set up a common vocabulary because some of these terms are used differently by different technologies, and the Pharo VM terminology is not an exception.

### Data Units: Words and Bytes

Objects are stored in memory, thus it is important to understand the basics of memory organization. Such an organization depends on the chosen computer architecture, which encompasses the memory and the processor. One trait that characterizes a computer architecture and strongly influences the memory organization is its *bit width i.e.,* the number of bits used to represent the main processing unit in a processor. For example, 64-bit machines are machines that have a bit width of 64. Since the most common machines nowadays are 64-bit machines, we will focus our presentation on them. However, the Pharo VM also supports 32-bit machines for compatibility with smaller devices.

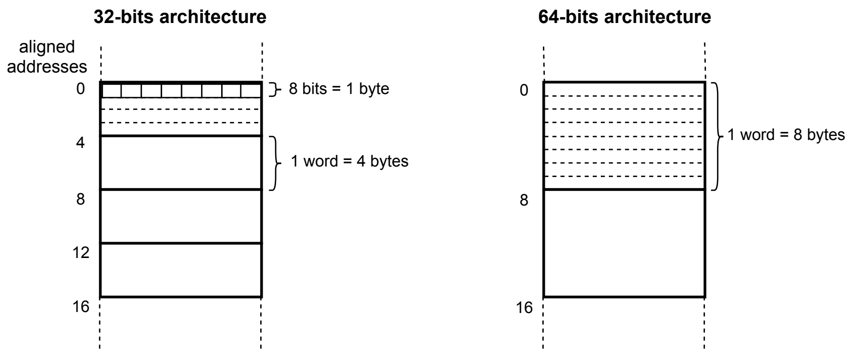

**Figure 2-1** Word size and Alignment on 32 and 64 bits architectures.

Memory is conceptually divided into cells of 1-byte length, each byte using 8 bits. Data is manipulated in units that group many bytes together. A *word* is a fixed unit of data with as many bits as the processor bit width. This means a word is 64 bits long –or 8 bytes long– in 64-bit processors, and 32 bits long –or 4 bytes long– in 32-bit processors. Figure 2-1 shows the memory layout on both 32 and 64-bit architectures. The main difference between these two architectures is their word size.

Each 1-byte memory cell has an address, a unique identifier that can be used to read and write into that memory cell. Addresses are typically restricted to fit in a word. Memory addresses form a sequence ranging from 0 to the highest integer that can be represented in a word. On 64-bits, for example, the maximum address is 2^64, thus it can address 2^64 different bytes. Memory cells are said to be contiguous if their addresses are contiguous. Note that Figure 2-1 only shows addresses that are multiple of a word, although each 1-byte cell also has an address. Representing memory addresses as data is what is commonly referred to as pointers.

Processors usually define also concepts such as *half-word* and *double-word*. We believe that such notations are confusing because they require context (Am I in a 32-bit architecture? 64-bit?), and thus we will not use them in this book. Instead, when referring to a sub-word unit, we will use the exact number of bits. For example, we will use *16-bit integer* instead of short or half-word.

Moreover, when useful we will use the term *nibble* to refer to half a byte, or 4 bits. A byte's high nibble and low nibble are the most and least representative halves of a byte respectively. Interestingly, when a byte is written in hexadecimal it gets a two-digit representation, where each digit represents the value of each nibble. For example, the byte 193 is represented as 16rC1 in hexadecimal. Its high nibble has a value of 16rC (12 in decimal). The low nibble has a value of 1.

## Alignment

A processor's ISA (Instruction Set Architecture) generally provides instructions that read or write data from an address with different granularity. For example, there are instructions to read/write individual bytes or entire words. Although ISAs give a lot of freedom to developers, modern architectures and micro-architectures (how CPUs are implemented internally) behave better when data-access are consistent and predictable. Particularly, address alignment, *i.e.,* its relative position in memory, is a property exploited by micro-architectures, compilers and programming languages for optimization.

A read/write to an address is said to be aligned when the address is a multiple of the accessed element size, in bytes. 1-byte reads are always aligned because all addresses are multiples of 1. 8-byte reads are aligned when the address is multiple of 8.

We will see later in this chapter how the Pharo VM exploits alignment to implement tagged pointers, and optimize the read of object header meta-data.

## Most and Least Significant Bytes and Bits

We mentioned before that memory cells are ordered. Moreover, the individual bits within a single cell are ordered too.

We say that the *most significant* bit in a byte is the bit that has most value in a byte, and conversely for the *least significant* bit. We can define the most and least significant bytes in a word in the same way. For example, the bit string 00000101 represents the number 5 in binary, and its least significant bit is represented by the rightest bit with value 1.

While we usually represent bytes from left-to-right (most-to-least significant), this is only the case with some architectures. The order in which individual bytes of a word are stored in memory is again a trait of the computer architecture: the endianness. An architecture is said to be little-endian if data bytes are stored from the least significant to the most significant, and big-endian otherwise. Understanding endianness is important when reading and writing values smaller than a word.

Nowadays, the most popular architectures out there are little-endian. This means that a 64-bit word with the value `16rFEDCBA987654321` is stored backward. Let's imagine that the word is stored at address a.

- The lowest address -a- contains the least significant byte -16r21-.
- The highest address -a+7- contains the most significant byte -16r0F- (see Fig. 2-2).

If we wanted to read the bytes in most-to-least significance order, then we need to iterate it backward: from a+7 to a.

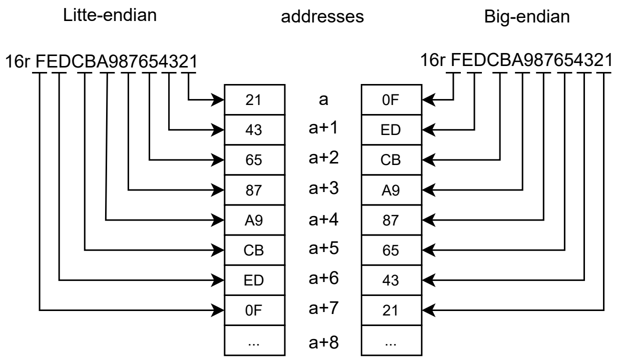

**Figure 2-2** 16rFEDCBA987654321 in 64-bits Little and Big-Endian.

## 2.2 Object Layout

Pharo programs are made of objects which are, for the most part, allocated in memory and occupy space. We call these objects *heap-allocated* because they reside in a memory region managed by the VM called the *heap*, that we will explore in later chapters.

### Object Formats

Pharo objects contain *slots* that store the object's data. Objects come in different kinds, determining the number of slots they contain and how their slot contents is interpreted. The following table summarizes the most common types of objects and their variations.

| Type/Format | # slots | Slot type | Slot size | Variations |
|---|---|---|---|---|
| Fixed | fixed | reference | word | Ephemeron |
| Variable | variable | reference | word | Weak |
| Byte indexable | variable | byte | 1/2/4/8 bytes | - |
| CompiledMethod | variable | reference+byte | word + 1 byte | - |

**Fixed and variable slots.** The number of slots in an object is either fixed, variable or a combination of both. Fixed slots are those decided statically. For example, an instance of class `Point` declaring variables `x` and `y` has two fixed slots. Variable slots are those determined at allocation time. The simplest example of variable slots are arrays, whose number of slots is specified as argument of the method `new:`. Some objects may contain a combination of fixed and variable slots, as it is the case of the instances of `Context`.

**Slot type.** Slots contain either object references or plain data bytes. Object references are pointers that reference other objects forming a graph, further explained in 2.6. Plain data is stored as raw bytes in a slot, typically representing low-level data-types such as integers or floats.

**Slot size.** The different slots in an object have a size that limits their contents. Reference slots store an address, and thus are a word long. Byte slots store a sequence of bytes, and thus element size can be 1, 2, 4 or 8 bytes. All fixed slots in an object are of type reference. All variable slots in an object are of the same type and are defined by its class. For example, instances of the class `ByteArray` have 1-byte slots, and instances of `FloatArray` have 8-byte slots containing IEEE-754 double precision floating-point numbers.

**Weak and Ephemeron.** Weak and Ephemeron object formats are variations of the types described above, extending them with special semantics for the memory manager. Weak objects are objects whose variable slots work as weak references (in contrast with strong references). That is, they don't prevent the garbage collection of the referenced object. Ephemeron objects are fixed objects representing a key-value mapping whose value is referenced strongly as long as the key is referenced by objects other than the ephemeron. These special objects will be further discussed in the chapters about memory management.

**CompiledMethod.** Compiled methods are variable objects that do not follow the conventions above. They contain a word sized variable part storing object literals, followed by a 1-byte variable part storing *bytecode* instructions.

## Representing Objects in Memory

Objects in Pharo are represented as a contiguous memory region with space for a header and data slots. The header contains meta-data used for decoding the object internals, such as the size, its type and its class. The data slots contain the object slots. Figure 2-3 illustrates the layout of a 3-slot object in both 32 and 64 bits.

Each object has a mandatory base header that contains common information such as its class, its size and mutable bits for the Garbage Collector. When objects are more than 254 words long, they are considered large, and their actual size is stored in an overflow header that precedes the base header. The base and overflow headers have each a fixed size of 8 bytes (64 bits). Headers are discussed in-depth in Section 2.4.

Data slots contain the different slots in an object. However, there is not a one-to-one mapping between an object slots and its underlying data slots. Data slots are always 1 word long each and their number is chosen to accommodate all the slots of the object. Each reference slot occupies one data slot.

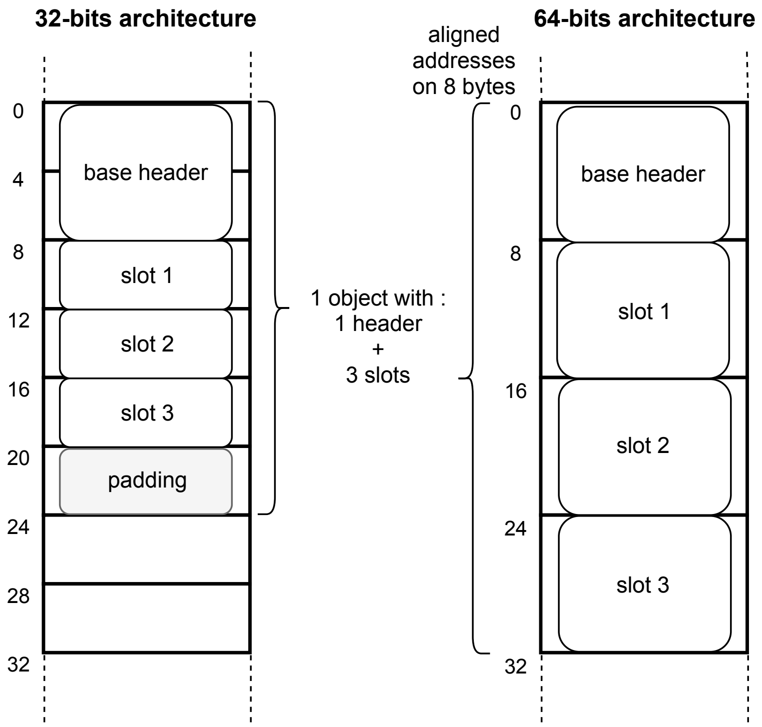

**Figure 2-3** Object Layout and Alignment on 32 and 64 bits Architectures.

Byte slots, however, may occupy less than a data slot. For example, in a 64-bit system, a data slot can accommodate 8 1-byte-long slots, 4 2-byte-long slots, 2 4-byte-long slots or 1 8-byte-long slots.

A special case arises when byte slots do not entirely fill an object data slots. For example, a 3-slot byte array occupies 3 bytes in a word-long data slot. In such a case, the Pharo VM introduces padding *i.e.,* filling space. Such unused filler is used to guarantee that the next object is aligned to the 8-byte boundary, a property that can be exploited for both performance and the representation of immediate objects explained in Section 2.3

## References and Ordinary Object Pointers

Objects reference each other forming a directed graph. Nodes in the graph are objects themselves, edges in the graph are usually called *object references*. In the Pharo VM, object references are called *ordinary object pointers*, or *oops* for short. There are two kinds of oops: object pointers to other objects and immediate objects. Pointers work as normal pointers in low-level languages.

Immediate objects are objects encoded in invalid object pointers using a technique called tagged pointers that takes advantage of pointer alignment.

Every object in Pharo has an address in memory, which is the memory address of its base header. An object A references an object B with an absolute pointer to B's base header stored in one if As reference slots. Figure 2-4 shows two objects forming a cycle. Each object has a single reference slot pointing to the other. References point to the object base header.

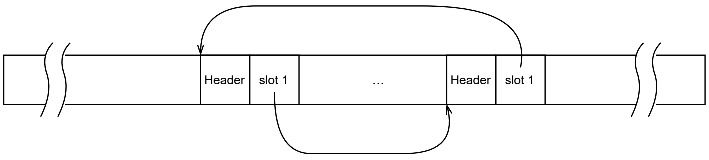

**Figure 2-4** References to heap-allocated objects are pointers to an object's base header.

## 2.3 Immediate Objects

The object representation presented so far imposes a non-negligible overhead on small objects, because of the space taken by its header. This problem becomes more visible with types that tend to have a large number of instances. Integers, for example, are in theory infinite and are used very often by even the simplest programs to drive the execution of loops. Representing integer objects as heap-allocated objects very fast becomes a bottleneck in an application. Instead, Pharo uses a common optimization called tagged pointers to represent integers and other common simple-valued immutable objects of the like.

### Alignment and Padding

Pointer tagging exploits the alignment property of pointers. In the Pharo VM, all objects are stored in memory aligned to 8 bytes. That is, an object address, and thus its header, is stored always at the 8-byte boundary, regardless of the architecture. Note that since an object header is always 8 bytes long, this means that the first data slot of an object is also always aligned. Further, since all data slots are a word long, each subsequent data slot is also aligned too.

To guarantee that objects always are aligned to the 8-byte boundary, the allocator inserts a padding at the end of an object filling it up to the next 8-byte boundary. This happens in two cases: byte objects in general and potentially all objects in 32-bit architectures. Byte objects may contain a number of slots that does not entirely fill a data slot, as shown in Section 2.2, thus requiring padding to fill a data slot. Moreover, data slots in 32-bit architectures are 4 bytes long, and thus an odd number of data slots requires 4-bytes of padding. Figure 2-3 shows an example of how an object with 1 header and 3 slots in laid out both in 32-bits and 64-bits architectures.

- in 64-bit architectures, the object occupies 4 words, for a total of 32 bytes. 1 base header of 8 bytes, 3 slots of 1 word each. The next free address (32 on the Figure) is aligned and thus an object can start there.
- in 32-bit architectures, the object occupies 5 words, for a total of 20 bytes: 1 base header of 2 words (8 bytes / 4 bytes per word), 3 slots of 1 word each. The next free address (20 on the Figure) is not aligned. In this case, the allocator inserts a 4 byte padding to align the following object to the 8-byte boundary.

Padding represents wasted memory and could be avoided in 32-bit architectures by requiring an alignment to a word. However, enforcing the same alignment in all architectures allows an overall code simplification by unifying the 64-bit and 32-bits implementations.

## Pointer Tagging

Pointer tagging is a technique to represent a set of values without the need to perform an allocation. Pointer tagging works by encoding (and disguising) such values within the pointer. Such a technique is possible thanks to object alignment.

Since all Pharo objects are aligned to 8 bytes, all object references will be multiple of 8 and have the form `xxx...xxx000` in binary, where its 3 lower bits are zero. Pointer tagging exploits this property by encoding data within the least significant bits of a pointer. With these three bits, we can encode up to 7 different tags (111, 110, ... 001) that tell us how to interpret the most significant bits.

The main advantage of this technique is to save storage space for common objects, which indirectly improves on data locality and CPU cache behavior. Its main drawback is the overhead incurred by runtime type checks: we need to verify if a pointer is tagged or not before operating on data.

Another consideration for tagged pointers is that they significantly reduce the number of values that can be represented. For example, tagged integers in the described schema have a maximum precision of 61 bits (+ 3 bits of tag = 64 bits). In the following sections we explain variable-sized tags and boxed values. Variable-sized tags help us mitigate this problem in 32-bit architectures, in which the loss is of 10% (3 bits out of 32). Moreover, a combination of pointer tagging and boxed values allows us to represent larger numbers by assuming that such larger numbers will be less common.

Currently, Pharo supports integers, characters and floating-point numbers as immediate objects. In 64 bits they use the tags `001`, `010` and `100` respectively, as shown in Figure 2-5. In 32 bits floats are not represented as immediate objects, integers present a 1-bit tag `1` while characters are represented with the 2-bit tag `10`, as shown in Figure 2-6.

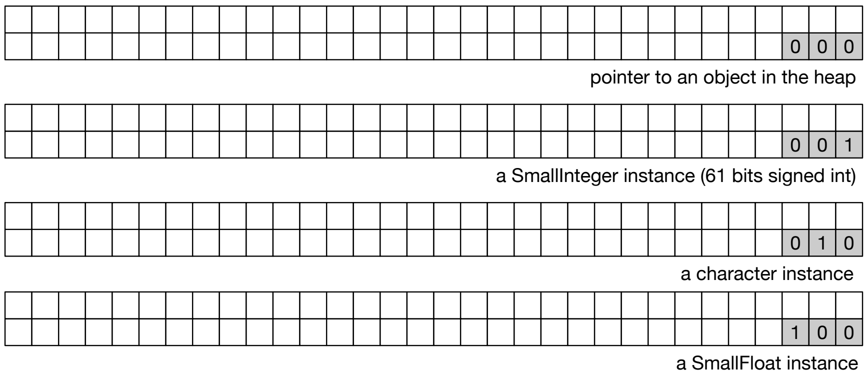
0 0 0
pointer to an object in the heap
0 0 1
a SmallInteger instance (61 bits signed int)
0 1 0
a character instance
1 0 0
a SmallFloat instance

**Figure 2-5** 64 bits immediate objects.

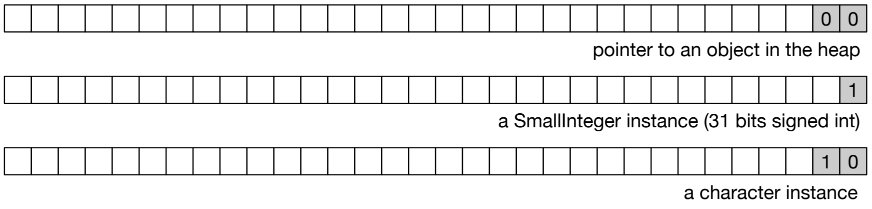
0 0
pointer to an object in the heap
1
a SmallInteger instance (31 bits signed int)
1 0
a character instance

**Figure 2-6** 32 bits immediate objects.

## Immediate Characters and Integers in 64-bits

In 64 bits, all immediate objects are represented as 61 bits of value and 3 bits of tag. In Pharo, immediate integers are instances of the class `SmallInteger`, and immediate characters are instances of the class `Character`. `SmallInteger` immediate objects range is between [-2^60,2^60-1] and are represented in two's complement. For example, the binary value `2r1010001` represents the untagged value `2r1010` which has the decimal value `10`.

Immediate characters encode in the 61 value bits the character's Unicode codepoint. This is, so far, enough to represent all Unicode codepoints: the maximum valid codepoint nowadays is 16r10FFFF, which requires only 21 bits.

**Listing 2-7** On 64 bits, SmallInteger are encoded on 61 bits.

```
SmallInteger minVal == (2**60) negated
>>> true

SmallInteger maxVal == (2**60-1)
>>> true
```

## 32-bit Immediate Integers and Variable Tags

In 32-bit architectures using 3 bits of tag would leave 29 bits left to represent integers. Instead of choosing this fixed tag representation, the 32-bit VM uses variable tagging. That is, different values use a different number of bits for tagging. Thus, tagging is carefully designed to avoid conflicts and ambiguities.

Immediate integers are tagged with a single bit and use the remaining 31 bits to encode a signed integer in two's complement. The range of immediate integers is [-2^30,2^30-1]. For example: `2r10101` represents untagged binary number `2r1010` which has the decimal value `10`.

Figure 2-6 illustrates the entire 32-bit tagging schema, with tag bits grayed out:

- `00` is an aligned address and therefore an object pointer in the heap.
- `*1` tag immediate integers.
- `10` tag immediate characters.

## 64-bit Immediate Floats

In 64-bit architectures, the Pharo VM represents floats as immediate objects with the tag `100`. The tagged value is an IEEE-754 64-bit double-precision floating-point number accommodated in 61 bits. However, to accommodate the 64 bits into 61 bits, immediate floats give up 3 bits in the exponent offset, storing only 8 out of 11 bits of exponent. The VM verifies that only immediate floats that do not lose information in this format are encoded as immediates. For floats that do not satisfy this constraint, floats use a boxed representation as explained in Section 2.3.

Figure 2-8 shows the structure of a `SmallFloat`. The sign bit is moved to the lowest bit of the tagged value, and the highest 3 bits of the exponent are lost.

## Boxed Native Objects

Numbers that cannot be encoded as *immediates* need to either gracefully fail or implement a fallback mechanism. After arithmetics, if a number does not fit in the 61 bits of a tagged pointer the runtime creates a boxed object with

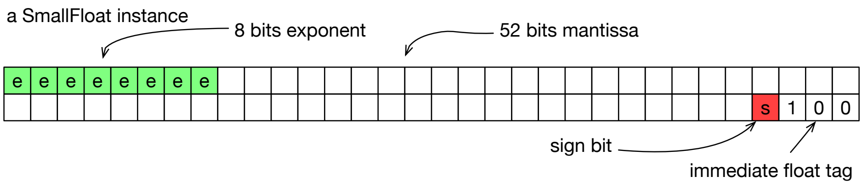

**Figure 2-8** 64 bits `SmallFloat` immediate.

the result. Boxed numbers are byte objects that contain the native number encoded in their byte slots.

Boxed numbers in Pharo include large integers (instances of `LargePositiveInteger` and `LargeNegativeInteger`) and boxed floats (instances of `BoxedFloat64`). Large integers implement variable-sized integers and represent arbitrary large integers as a string of bytes. Boxed floats are instead of fixed size: they represent IEEE-754 double-precision floating-point numbers, and store 8 bytes the corresponding float.

## 2.4 Object Header

Each *heap-allocated* object has a header that describes it. The header is made of a mandatory base header, and in the case of large objects, an overflow header that includes the object size. This section explains the overall design of the object header and each of its fields in detail.

### Base Object Header

The base object header contains meta-data that the Virtual Machine uses for several purposes such as decoding an object contents, maintaining garbage collection state, or even doing runtime type checks. Regardless of the architecture, the base header is 64 bits length, which means it is 2 words in 32 bits and 1 word in 64 bits as shown in Figure 2-9.

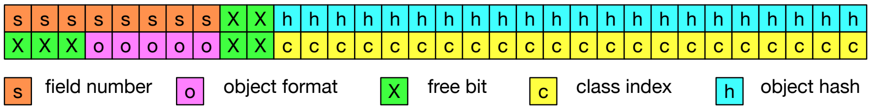
s s s s s s s s X X h h h h h h h h h h h h h h h h h h h h h h
X X X o o o o o X X c c c c c c c c c c c c c c c c c c c c c c
s field number
o object format
X free bit
c class index
h object hash

**Figure 2-9** Base Object Header.

This header is composed of several fields, marked with different colors in the figure. From the most significant to the least significant bits, the fields are as follows:

- **Object size (s).** This field contains the number of data slots in the object, padding included. For example, the object size of a byte array of 14 byte slots is 2 data slots. Padding is computed from the **object format** field below. Objects with more than 254 slots are considered large, are given 255 as object size and an overflow header as explained in Section 2.4.
- **Object hash (h).** This field contains 22 bits representing the identity hash of the object.
- **Object format. (o)** This field contains an enumerated value that identifies the format of the object as described previously in Section 2.2. The exact values of this field are explained in Section 2.4.
- **Class index (c).** This field contains the index at which the object's class is found in the class table.
- **Miscellaneous (x).** The remaining 7 bits illustrated with a green X are reserved for different reasons:
  - 1 bit is reserved for immutability.
  - 1 bit is reserved to mark the object as pinned. Basically, a pinned object is an object that cannot be moved in memory by the GC.
  - 3 bits are reserved for the GC: isGray (for tri-color marking), isRemembered (for the remembered table from old space to young space) and isMarked (for the GC mark phasis).
  - 2 bits are free.

Notice that the fields of the header are not all contiguous: miscellaneous bits are interleaved in between them. The header has been designed so commonly-accessed fields are aligned to a byte or 2-byte boundary. This design largely simplifies the decoding of the header, which boils down to a `load` and a `bit-and` instruction sequence. This simplifies the JIT compiler and generates better-quality machine code.

## Large Objects and the Overflow Header

The object size field is 8 bits long and cannot store values larger than 255. It is, however, desirable to have large arrays or strings with thousands of elements. For this purpose, large objects contain an extra header, namely the overflow header, preceding the base header *i.e.,* it is placed contiguous to the base header but in a lower address. The address of an object is always the one of its base header regardless if it has an extra header or not. The overflow header is 8 bytes long and contains the object size. It allows for very large objects with sizes of up to 2^64 words, which is largely sufficient. When an object has an overflow header, the object size field in the base header is marked with the value 255. Listing 2-10 pseudo-code shows how to obtain the number of data slots of an object.

**Listing 2-10** Extracting the number of data slots in an object

```
numSlotsOf: objOop
  numSlots := self baseNumSlotsOf: objOop.
  ^numSlots = 255
    ifTrue:  [ self readOverflowHeaderOf: objOop]
    ifFalse: [ numSlots ]
```

## Class References and Class Table

Each object includes a reference to its class in its header. However, for space reason, an object does not store the absolute address. An arbitrary address requires an entire word, which would add a non-negligible memory overhead. Instead, classes are stored in a table, and objects store the index of the class in the class table. The only exception to this is immediate objects that do not contain a header: the object tag is used as its class index.

Since programs are expected to have a low number of classes, class indexes are limited to 22 bits. 22 bits of class index support a maximum of 4 million of classes, which will be largely sufficient for most applications for a long time.

The class table is organized into 4096 pages of 1024 elements. The 12 most significant bits in the class index indicate the page index. The 10 least significant bits in the class index indicate the index of the class within the page.

Each class stores its own index as its hash. This allows the VM to get the index of a class without iterating the entire class table, and to guarantee a unique identity hash per class.

## Encoding of the Object Format Field

The object format field contains 5 bits that are used to identify the object's format explained in Section 2.2. An object's format is encoded as a 5-bit integer:

- `0`: 0 sized object *e.g.,* `nil`, `true`, `false`
- `1`: fixed size object *e.g.,* `Point`
- `2`: variable-sized object with no instance variables *e.g.,* `Array`
- `3`: variable-sized object with instance variables *e.g.,* `Context`
- `4`: weak variable sized object with instance variables *e.g.,* `WeakArray`
- `5`: ephemeron object
- `7`: forwarder object
- `9` : 64 bits indexable
- `10 - 11`: 32 bits indexable
- `12 - 15`: 16 bits indexable

- `16 - 23`: 8 bits indexable
- `24 - 31`: compiled methods
- `6` and `8` : unused

From the list above notice that zero-sized objects, instances of classes that define no instance variables, have their own format identifier. Variable objects with instance variables are marked separately from those without instance variables.

Special attention needs to be given to byte objects, where format `9` identifies byte objects with 64-bit slots, formats `10` and `11` identify byte objects with 32-bit slots, and so on. All byte objects having slots smaller than a word encode their padding in the format: the first format in each category (`10`, `12`, `16`, `24`) is used for objects that require no padding. Subtracting the format to the base format returns the number of padding bytes. For example, a byte array with format 21 has 21-16 = 5 bytes of padding.

The object format and its padding are necessary to compute the number of actual slots in the object. For example, given a byte array with format 21 and slot size 10, we can compute its size as 10 data slots * 8 bytes - 5 padding bytes = 75 bytes.

CompiledMethods are similar to byte arrays in terms of padding. However, they use a different format so the runtime can differentiate them from normal byte arrays.

## 2.5 Conclusion

In this chapter, we explored how Pharo objects are represented in memory using the Spur memory model supporting both 32 and 64 bits. From a user perspective, objects have a set of fixed and variable slots. Slots contain references to objects or plain byte data.

Under the hood, most objects are allocated in the heap and possess a header with meta-data. Object slots are stored in word-large data slots, and padding is inserted so that all objects are aligned to the 8-byte boundary. We exploit object alignment to implement integers, floats, and characters as tagged pointers. Tagged pointers use the least significant bits of a pointer to encode a type, and the most significant bits to encode a value. Thus, tagged pointers help us represent objects without the overhead of a memory header.

## 2.6 References

- Spur http://www.mirandabanda.org/cogblog/2013/09/05/a-spur-gear-for-cog/
- https://clementbera.wordpress.com/category/spur/

- https://clementbera.wordpress.com/2018/11/09/64-bits-immediate-floats/
- https://clementbera.wordpress.com/2014/01/16/spurs-new-object-format/
- https://clementbera.wordpress.com/2014/02/06/7-points-summary-of-the-spur-memory-
- http://www.mirandabanda.org/cogblog/category/spur/page/3/

CHAPTER 3

# Methods, Bytecode and Primitives

This chapter explains the basics of Pharo execution: methods and how they are internally represented. Methods execute one after the other, and *call* each other by means of message send operations. Methods execute under the hood using a stack machine. A stack holds the current calls and their values. Bytecode instructions and primitives manipulate this stack with push and pop operations.

In this chapter we explain in detail how methods are modelled using the sista bytecode set[**?**]. We explain how bytecode and primitive instructions are executed using a conceptual stack. The bytecode interpreter and the call stack are introduced in a following chapter.

## 3.1 Compiled Methods

Pharo users write methods in Pharo syntax. However, Pharo source code is just text and is not executable. Before executing those methods, a bytecode compiler processes the source code and translates it to an executable form by performing a sequence of transformations until it generates a `CompiledMethod` instance. A parsing step translates the source code into a tree data structure called an *abstract syntax tree*, a name resolution step attaches semantic to identifiers, a lowering step creates a control flow graph representation, and finally a code generation step produces the `CompiledMethod` object containing the code and meta-data required for execution.

## Intermezzo: Variables in Pharo

To understand how Pharo code works, it is useful to do a quick reminder on how do variables work. In this chapter we will deal with the low-level representation of variables (how read/writes are implemented). For a more complete overview on variables and their usage, please refer yourselves to Pharo by Example[1].

Pharo supports three main kind of variables. Each kind of variable is stored differently in memory and has different lifetime and behavior *i.e.,* they are allocated and deallocated at different moments in time.

**Temporary variables and parameters:** A method has a list of formal parameters and a list of manually declared temporary variables. Temporary variables and parameters are only accessible within the method that defines them and live during the entire method's execution. In other words, they are allocated when a method execution starts and deallocated when a method returns. Each method invocation has its own set of temporary variables and parameters, property allowing recursion and concurrency (understanding why is left as an exercise for the reader).

Temporary variables and parameters have each a unique `0`-based index per method. Moreover, they share the same index namespace, meaning that no two temporaries or parameters can have the same index. For example, assuming a method with `N` parameters and `M` temporary variables, its parameters are indexed from `0` to `N`, and its temporary variables are indexed from `N+1` to `N+M`.

**Instance variables:** Instance variables are variables declared in a class, and allocated on each of its instances. That is, each instance has its own set (or copy) of the instance variables declared in its class, and can directly access only its own instance variables. Instance variables live as long as its containing object. Instance variables are allocated as part of an object slots, occupying a reference slot, and are deallocated as soon as an object is garbage collected.

Instance variables have also a unique `0`-base index per ascending hierarchy, because an instance contains all the instance variables declared in its class and all its superclasses. For example, given the class `Class` declaring `N` instance variables and having a superclass `Super` declaring `M` instance variables, the variables declared in `Super` have indexes from `0` to `M`, the variables declared in `Class` have indexes from `M+1` to `M+N`.

**Literal variables:** Literal variables are variables declared either as *Class Variables*, *Shared Variables* or *Global Variables*. These variables have a larger visibility than the two other kind of variables. Class variables are visible by all classes and instances of a hierarchy. Shared variables work like class variables but can be imported in different hierarchies. Global variables are globally visible. Literal variables live as long as the program, or a developer decides to explicitly undeclare them.

**Listing 3-1** Source code with several literals

```
MyClass >> exampleMethod
    self someComputation > 1
    ifTrue: [ ^ #() ]
    ifFalse: [ self error: 'Unexpected!' ]
```

Literal variables do not have an index: they are represented as an association (a key-value object) and stored in dictionaries. Methods using literal variables store the corresponding associations in their literal frame, as we will see next.

## Literals and the Literal Frame

Pharo code includes all sort of literal values that need to be known and accessed at runtime. For example, the code below shows a method using integers, arrays and strings.

In Pharo, literal values are stored each in different a reference slot in a method. The collection of reference slots in a method is called the *literal frame*. Remember from the object representation chapter, that `CompiledMethod` instances are variable objects that contain a variable reference part and a variable byte indexable part.

Commonly, the literal frame contains references to numbers, characters, strings, arrays and symbols used in a method. In addition, when referencing globals (and thus classes), class variables and shared variables, the literal frame references their corresponding associations. Finally, the literal frame references also runtime meta-data such as flags or message selectors required to perform message-sends.

It is worth noticing that the literal frame poses no actual limitation to what type of object it references. Such capability is exploited in rare cases when a method's behavior cannot be expressed in Pharo syntax. This is for eample the case of foreign function interface methods that are compiled by a separate compiler and stores foreign function meta-data as literals.

## Method Header

All methods contain at least one literal named the *method header*, referencing an immediate integer representing a mask of flags.

- **Encoder:** a bit indicating if the method uses the default bytecode set or not.
- **Primitive:** a bit indicating if the method has a primitive operation or not.
- **Number of parameters:** 4 bits representing the number of parameters of the method.

- **Number of temporaries:** 6 bits representing the number of temporary variables declared in the method.
- **Number of literals:** 15 bits representing the number of literals contained in the method.
- **Frame size:** 1 bit representing if the method will require small or large frame sizes.

The encoder and primitive flags will be covered later in this chapter. The frame size will be explored in the context reification chapter.

### Method Trailer

Following the method literals, a `CompiledMethod` instance contains a byte-indexable variable part, containing bytecode instructions. However, it is of common usage in Pharo to make this byte-indexable part slightly larger to contain trailing meta-data after a method's bytecode. Such meta-data is called the *method trailer*.

The method trailer can be arbitrarily long, encoding binary data such as integers or encoded text. Pharo usually uses the trailer to encode the offset of the method source code in a file. It has, however, also been used to encode a method source code in utf8 encoding, or zipped.

## 3.2 Stack Bytecode and the Sista Bytecode Set Overview

Pharo encodes bytecode instructions using the Sista bytecode set[**?**]. The Sista bytecode set defines a set of stack instructions with instructions that are one, two or three bytes long. Instructions fall into five main categories: pushes, stores, sends, returns, and jumps.

This section gives a general description of each bytecode category. Later we present the different optimizations, the bytecode extensions and the detailed bytecode set.

### A Stack Machine

Pharo bytecode works by manipulating a stack, as opposed to registers. Typically, an operation accesss its operands from the stack, operates on them, and places the result back on the stack. We will call this stack the value stack or operand stack, to differentiate it from the call stack that will be studied in a later chapter.

For example, the following code shows the tree stack instructions required to evaluate the expression `2+7`. First, two push instructions push the values 2 and 7 to the stack. Second, the + instruction pops the two top values in the stack, operates on them producing the number 9, and finally pushes that value to the stack.

**Listing 3-2** Pseudo-bytecode performing 2+7

```
push 2
push 7
+
```

## Push Instructions

Push instructions are a family of instructions that read a value and add it to the top of the value stack. Different push instructions are:

- push the current method receiver (`self`)
- push an instance variable of the receiver
- push a temporary/parameter
- push a literal
- push the value of a literal variable
- push the top of the stack, duplicating the stack top

## Store Instructions

Store instructions are a family of instructions that write the value on the top of the stack to a variable. Different store instructions are:

- store into an instance variables of the receiver
- store into a temporary variable
- store into a literal variable

## Control Flow Instructions – Send and Return

Send instructions are a family of instructions that perform a message send, activating a new method on the call stack. Send instructions are annotated with the selector and number of arguments, and will conceptually work as follow:

- pop receiver and arguments from the value stack
- lookup the method to execute using the receiver and message selector
- execute the looked-up method
- push the result to the top of the stack

Conversely to send instructions, return instructions are a family of instructions that return control to the caller method, providing the return value to be pushed to the caller's value stack.

**Listing 3-3** The SmallInteger addition method is a primitive method

```
SmallInteger >> + addend
  <primitive: 1>
  ^super + addend
```

### Control Flow Instructions – Jumps

Control flow instructions are a family of instructions that change the sequential order in which instructions naturally execute. Different jump instructions are:

- conditional jumps pop the top of the value stack and transfer the control flow to the target instruction if the value is either `true` or `false`
- unconditional jumps transfer the control flow to the target instruction regardless of the values on the value stack

## 3.3 Primitive Methods

Some operations such as integer arithmetics or bitwise manipulation cannot be expressed by means of message sends and methods. Pharo express such operations through primitives: low-level functions implementing essential or optimized operations.

Primitives are exposed to Pharo through primitive *methods*. A primitive method is a bytecode method that has a reference to a primitive function. For example, the method `SmallInteger>>#+` defining the addition of immediate integers is marked to as primitive number 1.

Primitives are implemented as stack operations having only access to the value stack. When a primitive function is executed, the value stack contains the method arguments.

A key difference between primitive functions and bytecode instructions is that primitives *can fail*. When a primitive method is executed, it executes first the primitive function. If the primitive function succeeds, the primitive method returns the result of the primitive function. If the primitive funciton fails, the primitive method executes *falls back* to the method's bytecode. For example, in the case above, if the primitive 1 fails, the statement `^ super + addend` will get executed.

**Design Note: fast vs slow paths.** The failure mechanism of primitives is generally used to separate fast paths from slow paths. For example, integer addition has a very compact and fast implementation if we assume that both operands are immediate integers. However, Pharo by design needs to support the arithmetics between different types such as immediate integers, boxed large integers, floats, fractions, scaled decimals. In this scenario, a primitive is used to cover the fast and common case: adding up two immediate integers. The primitive performs a runtime type check: it verifies that both operands are immediate integers. If the check succeeds, the primitive performs its operation and returns without executing the fallback bytecode. This first execution path is the *fast path*. If the check fails, the primitive fails and the method's fallback bytecode implements the slower type conversion for the other type combinations.

## 3.4 Bytecode Encoding and Optimizations

The instructions of a method are encoded as bytes, that need to be decoded to either interpret them, JIT compile them or decompile them. Each instruction is made of an **opcode**, or operation identifier, followed by zero or more arguments. For example, the instruction `push instance variable 42` is encoded with bytes `#[226 42]`, where `226` is the opcode identifying the `push instance variable`, and the second byte (`42`) is the index of the instance variable to read.

### Variable-length Bytecode Encoding

Pharo encodes bytecode instructions using a variable-length encoding: instructions are encoded using one, two or three bytes. The encoding is designed for compactness and ease of interpretation. Commonly used instructions are encoded with less bytes, rarely used instructions use more bytes.

The variable-length bytecode design has two consequences:

1. **Compact representation of bytecode methods.** Using shorter bytecode sequences for common instructions works as a compression mechanism. This allows the virtual machine to fetch less bytes during interpretation, and to use less space to encode methods.
2. **Ambiguity during decoding.** Bytecode in a method needs to be decoded for reasons such as debugging or decompilation. However, decoding cannot start from any arbitrary point in a variable-length encoding. Consider a two-byte bytecode. If we start decoding bytecode instructions from the second byte, the decoder will interpret this byte as the first byte of an instruction. In the best case, the decoder will eventually fail. In the worst case, the decoder succeeds and returns a wrong decoding.

In the following section we will go into how such optimizations take place concretely in the Pharo's bytecode set.

### Optimising for Common Bytecode Instructions

As we said before, the variable-length bytecode encoding allows for shorter bytecode sequences for common instructions. For example, we can take the most common bytecode from the Pharo12 release (build 1521) using the script that follows. The script takes all the compiled code (methods and blocks), decodes all their instructions and groups them by their bytes.

```
((CompiledCode allSubInstances flatCollect: [ :e | e
    symbolicBytecodes ])
  groupedBy: [ :symBytecode | symBytecode bytes ])
    associations
      sorted: [ :a :b | a value size > b value size  ]
```

From that list, the 5 most common bytecode are:

| Instruction | Count |
|---|---|
| Pop | 209537 |
| Push self | 201567 |
| Push first temp/arg | 163965 |
| Send message in 1st literal | 77767 |
| Method return | 77090 |

We see in this list that the first three are largely more numerous than the two last. This tendency continues in the entire list of bytecode following an exponential decay. The first fifty instructions happen tenths of thousands of times, while the vast majority appear less than a thousand.

This observation is enough motivation to optimize such *very common* cases. Indeed, amongst the 255 most common instructions, 183 are already encoded as a 1 byte instruction.

## Encoding of Single-byte instructions

Instructions such as `pop` or `push self` are single instructions that do not need any parameter. The encoding of these instructions is straight forward: they are given a single byte. For example, `pop` is encoded as 216, while `push self` is encoded as 76.

There are however other common instructions that have parameters. This is the case, for example, of the `push instance variable` bytecode that is parameterized with the index of the reference slot in the receiver (the instance variable) to push. To encode this instruction as a single byte, the index is encoded within the instruction. That is, the bytecode `push instance variable at 1` is encoded as 0, the bytecode `push instance variable at 2` is encoded as 1.

Single-byte parametrized bytecode instructions are organized in ranges, often of a size that is a power of 2. For example, 1-byte `push instance variable` instruction is organized in a range of 16 instructions (2^4). 1-byte `push instance variable` instructions are encoded with bytes from 0 to 15, parameterized with indexes from 1 to 16 respectively.

An alternative way of seeing this encoding is to see that an instruction opcode is not the byte on itself but the most significant bits of the byte. If we consider again the range of bytecodes `push instance variable`, the most significant nibble remains always zero regardless of the bytecode, while the lowest part always changes following the index to push.

```
"The most significant nibble is always 0 for this range of bytecode"
0 to: 15 do: [ :e | self assert: ((e >> 4) bitAnd: 16rF) = 0 ].
"The least significant nibble is always the index to push"
0 to: 15 do: [ :e | self assert: (e bitAnd: 16rF) = e ]
```

### Optimising for Common Bytecode Sequences

Besides common instructions, another useful observation is that many instructions are usually combined together. Consider for example the statement `^ self`, which is commonly used to perform an early exit from a method, and inserted at the end of every method that does not have an explicit return. A naïve translation of `^self` could use the following sequence of instructions.

```
push self
return top
```

Using two instructions requires – at least – two bytes, and forces the interpreter to pay twice the cost of instruction fetch/decode. Pharo optimizes such common sequences using a single (often also single-byte) instruction to do the entire operation, often called (static) super instructions in the literature [3].

### Optimising for Common Messages and Literals

Another source of overhead happens on the over-reliance on literals. In Pharo, each method has its own literal frame: literals and constants are not shared between methods, causing a potential redundancy and memory inefficiency.

One way to minimize such overhead is to design special instructions for well-known constants. Constants such as `nil`, `true`, `false` need to be known by the VM for several tasks such as initializing instance variables, or interpret conditional jumps. The VM benefits from this knowledge to provide specialized instructions such as `push true` that do not fetch the `true` object from the method literal frame but from the pool of constants known from the VM.

In the same venue, immediate objects can be crafted by the VM on the fly, avoiding the storage in the literal frame. Instructions such as `push 0`, encoded as 80, represent the usage of constants that appear often, for example, in loops. When executing those instructions, the VM create an immediate object by tagging a well-known value.

Finally, another variation of this optimization happens on common message sends *e.g.,* arithmetic and comparisons selectors. These selectors happen so often, that instead of storing the selector in the method's literal frame, they are stored in a global table of selectors called `special selectors`. The Pharo bytecode set defines `send special selector` instructions.

## 3.5 The Sista Bytecode

### Bytecode Extensions

Some of the bytecodes take extensions. An extension is one or two bytes following the bytecode that further specify the instruction. An extension is not an instruction on its own, it is only a part of an instruction.

CHAPTER 4

# Calling conventions

A calling convention dictates how two procedures communicate. This has two main aspects:

- first, how arguments are passed between caller and callee (by reference, by copy...), how the procedure returns
- second, how limited resources such as registers are maintained.

The principle is that procedures are black boxes. A procedure does not know the shape of its caller, nor the shape of its callee. The caller may be optimized differently, use a different/unconventional set of registers. This means that a procedure must be written to be called from anywhere and to call procedures that can do anything.

## 4.1 Passing arguments

The calling convention dictates how arguments are passed, and where they are stored. This way, the convention decouples procedures from their implementations. For example, the Smalltalk-80 calling convention dictates that upon a message send, the receiver and all arguments are pushed to the stack. Then the method executed, which knows it has N arguments by construction, can access the receiver (`self`) by skipping the N top elements of the stack.

## 4.2 Returning

Low-level architectures store the current program counter in a special CPU register. The program counter register is unique, and can only hold a single instruction pointer, which for efficiency reasons is made the program counter of the *Low-level architectures store the current program counter in a special CPU register. The program counter register is unique, and can only hold a single instruction pointer, which for efficiency reasons is made the program counter of the* currently*Low-level architectures store the current program counter in a special CPU register. The program counter register is unique, and can only hold a single instruction pointer, which for efficiency reasons is made the program counter of the _currently* executing procedure. This means that the program counters of all the procedures active on the call stack must be stored somewhere, and restored when control returns to those procedures.

A calling convention dictates how the current program counter is stored when a procedure is called, how the control is passed to the called procedure, and how the program counter is restored when the procedure returns. There are two main families of solutions for this aspect in low-level ISAs (Instruction set architectures).

- In CISC (complex instruction set architectures) machines, the *In CISC (complex instruction set architectures) machines, the* call procedure*In CISC (complex instruction set architectures) machines, the _call procedure* instruction will push the current program counter to the stack and transfer the control to the procedure. The return instruction will do the inverse. In pseudocode:

```
call procedure
=>
push IP
IP := procedure

return
=>
IP := pop
```

- In RISC (reduced instruction set architectures) machines, the *In RISC (reduced instruction set architectures) machines, the* call procedure*In RISC (reduced instruction set architectures) machines, the _call procedure* instruction will copy the current program counter to the link register and transfer the control to the procedure. The return instruction will do the inverse. This register must be saved by the callee explicitly if needed.

```
call procedure
=>
LR := IP
IP := procedure

return
=>
IP := LR
```

## 4.3 Shared state

When procedure A calls procedure B, A does not know what potential effects B will produce. In general, the problem lies in the usage of registers and the call stack. If procedure A was using registers R0 and R1, it cannot know if procedure B will read from those registers or write on them. Thus, procedure A should make sure that its state before the call is preserved after the call returns.

### Keeping registers

Calling conventions dictate how such preserving must be done. In general, registers are split into two sets: *Calling conventions dictate how such preserving must be done. In general, registers are split into two sets:* caller saved*Calling conventions dictate how such preserving must be done. In general, registers are split into two sets: _caller saved* registers, and *Calling conventions dictate how such preserving must be done. In general, registers are split into two sets: _caller saved_ registers, and* callee saved*Calling conventions dictate how such preserving must be done. In general, registers are split into two sets: _caller saved_ registers, and _callee saved* registers.

- A *A* caller-saved*A _caller-saved* register is a register that the caller must preserve before the call and restore after the call.
- A *A* callee saved*A _callee saved* register is a register that the callee must preserve when it's called and restore before it returns.

Preserving and restoring a register is usually done by saving the values in predictable memory positions, usually on a stack.

### Keeping the call stack

The same problem arises with the call stack. A procedure calling another procedure must not only assume that the registers it was using were not modified, but also it must assume that the stack was preserved. This is particularly important when calling high-order functions, closures, or polymorphic procedures. Otherwise, if each procedure leaves the registers and the stack in different states, the caller will not be able to continue correctly.

CHAPTER 5

# The Spur Memory Manager Overview

The Pharo virtual machine implements an object memory manager named Spur. An object memory manager is a memory manager whose allocation units are objects. In contrast to the operating system memory manager that manipulates raw memory, the Spur memory manager manipulates only objects. For example, the lowest-level allocation operation is to allocate an object specifying the desired number of slots, format and class index. We saw in the previous chapters the object format, and what these three arguments mean.

```
memoryManager
  allocateSlots: numberOfSlots
  format: instanceSpecification
  classIndex: classIndex
```

The virtual machine tracks the life-cycle of all objects it allocates. The memory manager implements an automatic garbage collection mechanism. It detects when an object has no more incoming references, and deallocates it. The garbage collector of Spur is precise and generational. It is precise because it distinguishes non-ambiguously object pointers from random memory adresses. It is generational because it categorizes objects *by age*, treating them differently depending on their age.

In this chapter we do an overview of the Spur memory manager, and the concepts behind it. We will study how the memory is organized, how object generations impacts this organization, and how objects grow old in this generational setup.

## 5.1 Spur features

The Spur memory model supports the following features:

- Support both 32 and 64 bits. - Performance improvement. Several decisions led to a much faster system (new GC, large hash, immediate characters). - Variable sized and segmented memory. The memory allocated in the operating system by the virtual machine can grow and shrink according to the image size. Pharo images as large as several Gb are possible. - Incremental and efficient garbage collector. As we describe in the following chapters, the GC is now. - Fast become: the model introduces forwarders are special objects that avoid to walk the complete heap to swap references. - Ephemerons: the model introduces advanced weak structures called *Ephemeron*. An Ephemeron is an object which refers strongly to its contents as long as the Ephemeron's key is not garbage collected, and weakly from then on. - Pinned objects. Pinned objects will not be moved by the garbage collector. This is an important point for Foreign Function Interface - as you can read in the corresponding book.

## 5.2 Memory Structure Overview

The Spur memory manager layouts its memory in two main sections: the new space and the old space. The new space contains objects considered young i.e., objects that have been recently created. The old space contains objets that did survive in the new space for some time, and were promoted as adults in the old space.

At startup, the memory manager requests the operating system a chunk of raw memory to store the new space and the old space. The memory manager uses the first part of this memory as the new space, and the rest as old space. Figure 5-1 depicts how the two spaces are laid-out in memory, considering that lower addresses are at the left, and higher addresses are at the right.

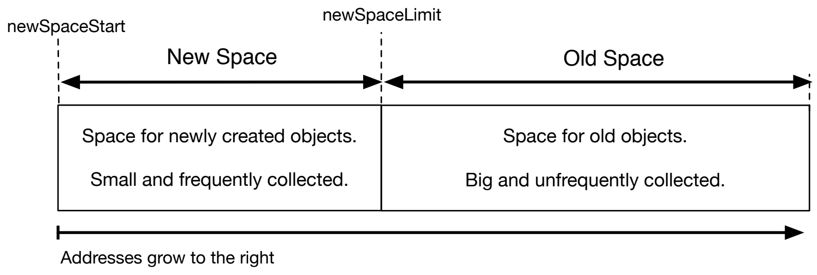
newSpaceStart
newSpaceLimit
New Space
Old Space
Space for newly created objects.
Small and frequently collected.
Space for old objects.
Big and unfrequently collected.
Addresses grow to the right

**Figure 5-1** Memory Map: a new space followed by an old space.

Addresses in the new space are lower than those in the old space. This way, the VM easily determines if an object is old or young by comparing its ad-

**Listing 5-2** A young object is an object located below the newSpaceLimit

```
memoryManager newSpaceStart.
memoryManager newSpaceLimit.

SpurMemoryManager >> isYoung: oop
  <api>
  "Answer if oop is young."
  ^(self isNonImmediate: oop)
    and: [self oop: oop isLessThan: newSpaceLimit]
```

dress against the limit of the new space. The memory manager stores the limits of the new space as `newSpaceStart` and `newSpaceLimit`. It defines that an object is young if its address is less than the new space limit.

## 5.3 Memory Growing and Segments

The new space remains fixed once initialized i.e., it does not grow after its allocation. On the contrary, the old space is organized in one or more memory segments, and it can grow dynamically by adding new segments to it. The new space and first segment of the old space are allocated in single contiguous chunk of memory as we have seen above. Newly added segments do not require to be contiguous, but they need to be at higher addresses than the first segment.

When a new segment is added, a bridge is added to the end of its previous segment. A bridge is a fake object that fills the gap between the two segments. Bridge objects have the format of a byte array simulating a size equals to the gap between the two segments. They give the Spur memory manager and its garbage collector the illusion of an old space made of a single contiguous chunk of memory. Bridge objects are not visible from the program and do not move during garbage collection.

## 5.4 Memory Initialization

When the virtual machine starts, it requires memory from the operating system to store both the new space and the first segment of the old space. The size of the new space is computed from a parameter stored in the image file header. The image file, storing all objects in previous sessions, is loaded into the first segment of the old space. The size of this first segment is computed as the addition of the image size and a free space headroom to fit objects coming from the new space.

## 5.5 Spur Generational Garbage Collection

Spur implements a generational automatic garbage collector based on an heuristic named the generational hypothesis. The generational hypothesis states that most objects die young, specially true in highly-interactive applications, so young objects are stored separately than old objects. This is why the Pharo VM uses two different garbage collector algorithms: one for new objects implementing a generation scavenger, and one for old objects implementing a mark and compact.

The memory manager allocates by default objects in the new space. When the new space has little space left, it is garbage collected using a copy collection algorithm named generation scavenger, that we will explore in detail in a following chapter. The new space is much smaller than the old space, so garbage collecting it is fast, producing unnoticeable application pauses. If the generational hypothesis holds, unused young objects are reclaimed shortly after their instantiation and never moved to the old space.

Objects that are not reclaimed during a garbage collection are called *survivors*. As objects survive several new space garbage collections they grow old. Eventually, objects old-enough are *tenured*: they are moved to the old space. The old space is several times bigger than the new space, thus garbage collecting it is expensive and creates long application pauses. Most objects are collected during new space collections, so collect the old space is not often required.

When the old space has little space left, a mark and compact collection algorithm reclaims unused objects. This algorithm first marks all used objects, and then scans the entire old space freeing unmarked objects and compacting the memory.

## 5.6 The Stack

The stack (both Pharo and C) resides in the lower address of the memory. This is the stack used by the C code and also the stack pages are allocated in this stack. All the execution of a process stores the information in the stack. The stack is the real representation of the contexts in the image. The frames are in a sequence in the stack. Each frame knows the calling frame with a pointer. Objects referenced into stack frames are retained i.e., never garbage collected.

## 5.7 Conclusion

We sketch a first overview of the memory architecture of Pharo.

## 5.8 The New Space

The new space is the region where young objects are allocated. It is the place where the generational garbage collector is taking place. it is commonly named a *scavenger*.

### Minimal objects: Liliputians

```
\textbf{Note:}
 In Spur, the minimum size of an object is the size of its header and one slot.
This means that an object without slots will contain one extra hidden slot.
In 64 bits, the smallest object is 16 bytes long: 8 bytes of header + 8 bytes of o
In 32 bits, the smallest object is also 16 bytes long: 8 bytes of header + 8 bytes
```

This minimum size for any object is important for some features of the VM (GC, become). For example, when the VM needs to move an object in memory, it ensures that there is always enough space to put a **forwarder object** at the previous object position. A **forwarder object** is a 2 words value that encodes the new position of the moved object.

## 5.9 New Space Memory Layout

The new space is divided into three areas (eden, future and past) as shown by Figure 5-3. The Eden (5/7 of the new space) and two other areas (1/7 of the new space each) that alternatively play the role of *past* and *future* space. The VM always allocates new objects into the Eden space if there is enough space.

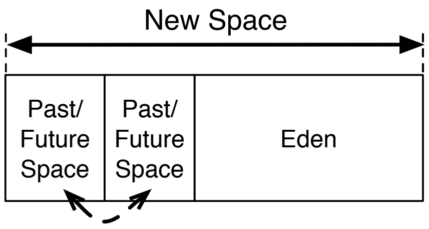

**Figure 5-3** The NewSpace structure composed of the eden, and two past and future regions.

## 5.10 The Scavenger

The scavenger is a copy-collector responsible to manage the new space memory region. The scavenger is periodically triggered on some events such as:

the eden is full i.e., left space reached a predefined threshold. There are multiple scavenger policies:

- TenureByAge: it is the default policy. It consists in copying surviving objects either in future space or old space depending on a threshold (cf `SpurGenerationScavenger>>shouldBeTenured:`). Surviving objects in past space with addresses below this threshold are tenured i.e., copied to the old space instead of future space. Initially, the threshold value is `0` meaning that the scavenger will not tenure any surviving objects, they are copied to future space. At the end of the scavenge, the scavenger updates the treshold if the future space is filled at more than 90%. The next scavenge will then tenure objects.
- TenureByClass: this policy consists in tenuring objects instance of a specific class.
- TenureToShrinkRT: this policy consists in tenuring objects to shrink the remember table i.e., minimizing objects in the old space that reference objects in the new space.
- DontTenure: this policy consists in not tenuring any objects i.e., threshold is fixed at `0`.
- MarkOnTenure: the full mark and sweep GC of the old space calls the scavenger with this policy. The threshold will be `0`.

The VM allocates new objects in the eden space. When a newly allocated object address in the eden space reaches the scavenge threshold, the scavenger is triggered. To free some space in the eden, the scavenger does not iterate over all objects it contains. It computes surviving objects i.e., referenced by **root objects**. There are three kind of roots:

- objects referenced in the stack i.e., used as receivers or parameters;
- objects stored in the remembered set. This set contains all objects allocated in the old space that contains at least one reference to an object in the new space;
- special objects known by the VM such as: nil, true, false, class table, etc.

Add a picture with pointers from the oldspace to the newspace (and that are in the remembered set) The scavenger starts by copying roots references allocated into Eden or past spaces into the future space. Then, it traverses these copied objects and copies their referenced objects that reside into Eden or past space into the future space. At the same time, traversed references are fixed to correctly reference the copied objects. Similarly, root objects references are also updated. Finally, the future space contains all reachable objects from roots that were present into the Eden and past spaces. Moreover, all their references have been updated to correctly point to objects into the future space. If the future space is filled during the scavenge, some objects are tenured i.e., copied into the old space.

There are multiple strategies regarding the tenuring of objects:

- By age, using the addresses in the past (older objects have smaller addresses).
- Tenure to shrink the remembered table, it is used when the remembered set is too big.
- Instances of a given class, it is used by the `someInstance` primitive before its execution making all instances available into the old space.

Once the scavenge is finished, the future and past spaces are switched; it just means that the future space is now considered as the past space and vice-versa. Add a figure showing the switch

## 5.11 Example of a Scavenger pass

```
Roots references: A, C

B -> D
C -> A
A -> B
A -> C
E
```

- Step 1: copy roots references

```
future space: A, C
```

- Step 2: We go over first surviving object (A)

```
future space: A, C, B
(C was already copied, so we just update the reference)
```

- Step 3: We go over second surviving object (C)

```
future space: A, C, B
(C points to A, but A was already copied, so we just update the
    reference)
```

- Step 4: We go over next surviving object (B)

```
future space: A, C, B, D
```

- Step 5: We go over next surviving object (D)

```
Nothing to do, and nothing new in future space
```

- Step 6: exchange past and future spaces

## 5.12 to be continued

## 5.13 The Old Space

At startup, the VM allocates a memory map as depicted on Figure 5-1. Initially the OldSpace has only one segment but it can then vary dynamically by allocating and freeing segments. Figure 5-4 shows an example of an old space with two segments.

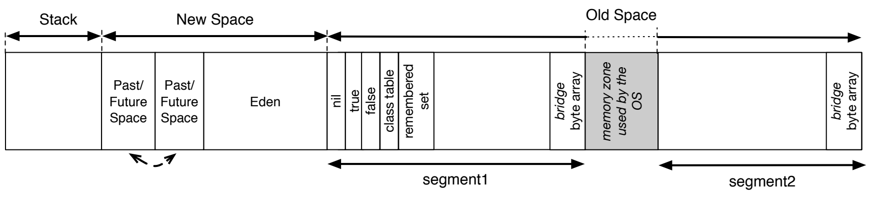

**Figure 5-4** The Old Space with two segments.

The first segment of the old space stores at least these objects:

- nil
- true
- false
- classTable i.e. an array of all the classes in the image; in its header, an object does not directly store a reference to its class but the index of its class in this table.
- remember set
- freeTree
- freeLists
- bridge byte array

## 5.14 The Free List

When a new object should be allocated, the VM checks first if there is an object that was already allocated and garbage collection of the given size that can be reused. Else it allocates a new object by ”cutting” and splitting a large free object. The management of such used *cells* is managed by the *Free List* and its companion the *Free Table*. The Free List manages chunks of memory below numFreeLists (i.e., 63 in 64 bits architecture). The Free Table manages larger chunks of memory.

## 5.15 Free cells in memory

The free list is a structure that keeps information about the memory that can be used to allocate objects. It refers to free cells (which are special object tagged as Free class). The minimum size of a free cell is two words: one for the object header and the next. Such minimal cells are used to allocate an object with one single slots (because one slot is for the header and the second one is for the slot).

The free cells stays in the memory where they are allocated as shown in Figure 5-5. In Figure 5-5, there are two objects of size 5 word is free. Such objects are structured in a linked-list of objects of the same size. A free cell is an element of a linked-list. The first object then points to the next free object of the same size. The second object is the last one of this size so its next slot is empty. In addition when the size allows it, the free object has a previous pointer in addition to the next one (implementing a double linked list).

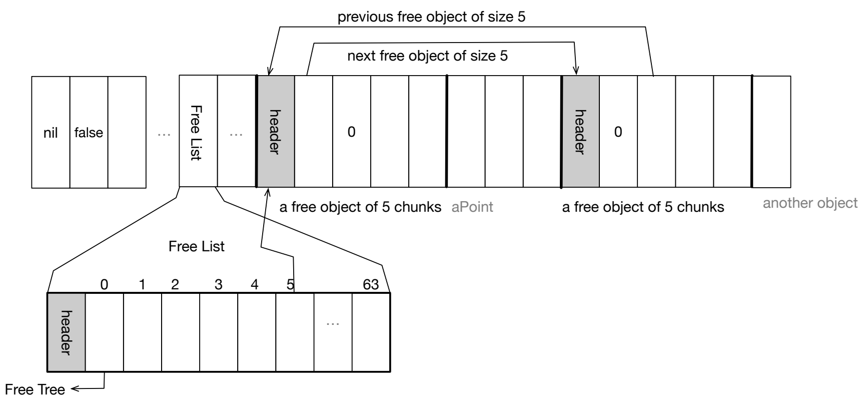
previous free object of size 5
next free object of size 5
nil
false
...
Free List
...
header
0
header
0
a free object of 5 chunks
aPoint
a free object of 5 chunks
another object
Free List
0
1
2
3
4
5
63
header
...
Free Tree

**Figure 5-5** Free cells of size 5 in the memory.

## 5.16 Free list

The free list is a structure that keeps tracks of free cell objects based on their size. Figure 5-5 shows that the free list linked-lists are built using the free objects.

On 64 bits, the free list has 63 slots to keep a free cell linked list per number of slots of the object. The first element of the free list is a pointer to the free tree (a tree structure that manages chunk of memory larger than 63 words).

The free list first element is a pointer to the tree table, the next elements are pointers to linked-list of free objects, one linked-list per object size.

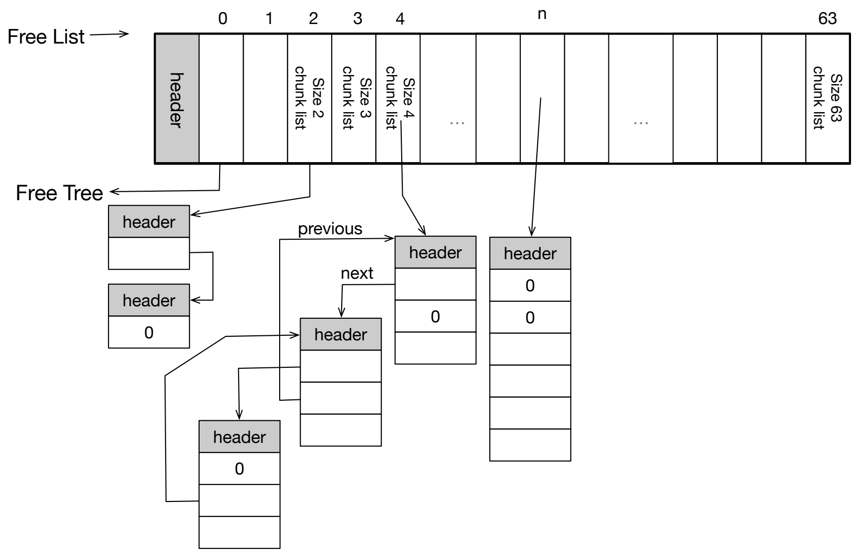

**Figure 5-6** Free list view is a table of linked-lists of free objects.

## 5.17 Free Table

# CHAPTER 6

# Weak Objects

References to Objects are actually devided in two categories: strong and weak.

## 6.1 Strong and Weak references

Strong references are visited by the garbage collector, and they are used to calculate the reachability of an object. An object that is not reachable through a path of strong references from any of the given roots will be collected.

Weak references are not visited by the garbage collector, and they are not analyzed by the garbage collector. An object referenced only by weak references will be garbage collected.

In PharoVM, a reference is a pointer. This pointer does not encode the strong/weak information. The reference is considered to be strong or weak depending on which object is holding it. An Object has a specific format which defines if its references are all strong or all weak (see **??** for a special case). We cannot mix strong and weak references in an object. Therefore normal objects have all strong references. We call weak objects the objects that have all weak references.

## 6.2 Strong and Weak Objects

The format of an object encodes the instance specification (instSpec). The format of an object is stored in its header **??**. The instSpec of a weak object is 4. This instance specification describes a layout that contains both fixed part (instances variables) and variable part. Every reference contained by this object will be weak. An example of this are the instances of the WeakArray class.

While an object referenced by a weak object is reachable, the reference in the weak object will be valid, and will point to the object. A weak object is collected as any other. When the garbage collector collects an object referenced by a weak object, the reference in the weak object will be set to nil.

If during garbage collection a weak reference is set to nil, a semaphore is signaled to allow the image to handle it. This semaphore is used in the image side to implement a finalization process. The semaphore is registered in the SpecialObjectArray **??** in the 42th position.

## 6.3 Weak reference collection during scavenging

The handling of weak references is done during the execution of the garbage collector. In this subsection we will focus on the garbage collection of the New space: the scavenge. The scavenger copies the weak objects but does not scan the references it contains. When we scavenge an object, only the strong references are visited. A weak object only has weak references, so the references are not visited. Once the weak object is copied, a reference to it is kept in a data structure that we will call the weak list. The number of strong references in an Object is calculated from its format (SpurMemoryManager » #numStrongSlotsOf:format:ephemeronInactiveIf:).

After all the Eden and the Past spaces have been scanned, weak list is iterated. For each of the objects in the weak list, the scavenger attempts to update each of its references. If the reference is pointing to a object in the Eden or Past space that has been forwarded, the scavenge follows the forwarder and update the reference to the new address in the Future space or in the Old space, because the refered object could have been tenured. If the reference points to an object that has not been copied to the Future space nor been tenured (ergo it's not a forwarder), the reference in the weak object is set to nil. If any reference was set to nil for a given weak object, it will be counted as pending finalization (instanceVariable : StackInterpreter » #pendingFinalizationSignals). This variable is checked in the StackInterpreter » #checkForEventsMayContextSwitch: and if there is pending finalization, it signals the TheFinalizationSemaphore. Then the variable is cleared.

TODO: #diagram SpurGenerationScavenge » #processWeakLinks.

During the copy of a weak object, this object may have been tenured. If an object in the weak list has been tenured, it is also checked to see if it should be in the remembered set. If a weak object's referenced object have been tenured, this weak object may be removed from the remembered set as well.

Objects in the Old space that reference objects in the new space are kept in the remembered set. Objects in the remembered set are roots of the scavenging process. Therefore the remembered objects in the remembered set need to update their references. For weak objects in the remembered set, the references are updated or set to nil.

If the old object in the remembered set does not have any more the references to objects in the New space, it is removed from the remembered set.

## 6.4 Weak list structure

When copying an object, the scavenger leaves a forwarder to the new location. When copying a weak object, the scavenger leaves a corpse. The weak list is a linked list of the corpses. Each corpse contains the address to the new location of the weak object and a reference to the next corpse in the weak list.

A corpse is a normal forwarder. The memory format guarantees that every single object has at least one slot. This slot is used to hold the forwarded reference. It also guarantees that there is a 64 bits header. As there is no waranty that the object than more than one slot, the address of the next corpse in the weak list has to be encoded in the object's header.

All the addresses of the corpses, including the one in the weakList instance variable (in the SpurGenerationScavenger), are offsets from the start of the New space. This offset is expressed in allocationUnits size (8 bytes). The offset starts in 1 to detect if the list has ended, a zero value for the offset is not a valid corpse marking the end of the link list.

This offset is encoded in 27 bits of the object header (22 bits of the hash and 5 bits of the format). The remaining part of the header's informations are used by the scavenger. For example, the class index is used to know which objects are forwarders. With this trick, we can address objects inside the new space of a maximum size of one gigabyte. The current calculation of the VM to find the next corpse from the current one is the following:

```
allocationUnit = 8.
offset = (formatBits + (hashbit << 5)) * allocationUnit.
nextCorpseAddress = offset - 1 + newSpaceStart.
```

The corpses are added to the list in the beginning of the list. The head of the list is the last added corpse. This allows the scavenger not to traverse the whole list to add a new corpse.

The weak list is built in the scavenging and it is discarded after it ends.

CHAPTER 7

# Ephemerons

Let's start with an example: An open File object uses a operating system handler. When opening a File object, a file is opened in the operating system. This is a limited and should be given back to the system as soon as it is not used anymore. If the File object is collected without being closed, the system won't be able to close the file. Therefore we need an additional mecanism to detect when a file object is collected. One such mechanism is called Ephemerons.

CHAPTER 8

# JIT Vocabulary

A Jitted method is a function/routine that was translated to native code at runtime from a bytecode method (I know you know this, just putting this to contrast with what is below). An intrinsic is a function/routine defined by the compiler that is not a method.

Intrinsics are defined by compilers for things that cannot generally be expressed in the host language (nor Smalltalk nor C), because for example you want to do some strange register usage.

For example, trampolines are intrinsics of our JIT compiler, they are not methods. Other intrinsics in our VM are routines to expose the Stack Pointer, or the machine code implementation of pushThisContext, which is a separate function. When the JIT compiler starts, it will define all intrinsics these as little assembly functions. Intrinsics are never garbage collected.

CHAPTER 9

# Looking at Stack Structure

## 9.1 Stack/Context Terminology

Since Pharo reifies the notation of execution stack we have two separate concepts and different relationships.

- A *stack frame* is a frame in the native execution stack.
- A *context* is the reified causally connected frame available in the image as an object.

A frame can be in the following states:

- Single: They don't have matching reified context in the image.
- Married: It has a matching context in the image.

A context can be in the following states:

- Single: there is not matching context in the image for a given frame.
- Married: There is a matching context in the image and a corresponding frame in the stack.
- Widowed: There is a context in the image, but the frame in the stack has returned.

TODO: EXPLAIN THE DIVORCE MECHANISM

- -

## 9.2 Cog Stack Structure

Frameless method activation looks like

```
receiver
args
sp->  ret pc.
```

Format of a stack frame in a frameful activation. Word-sized indices relative to the frame pointer.

Stacks grow down:

```
receiver for method activations/closure for block activations
arg0
...
argN
caller's saved ip/this stackPage (for a base frame)
fp->  saved fp
method
context (initialized to nil)
frame flags (interpreter only)
saved method ip (initialized to 0; interpreter only)
receiver
first temp
...
sp->  Nth temp
```

In an interpreter frame, frame flags hold

- the backward jump count (see ifBackwardsCheckForEvents)
- the number of arguments (since argument temporaries are above the frame)
- the flag for a block activation
- and the flag indicating if the context field is valid (whether the frame is married).

saved method ip holds the saved method ip when the callee frame is a machine code frame. This is because the saved method ip is actually the ceReturnToInterpreterTrampoline address.

In a machine code frame

- the flag indicating if the context is valid is the least significant bit of the method pointer
- the flag for a block activation is the next most significant bit of the method pointer

Interpreter frames are distinguished from method frames by the method field which will be a pointer into the heap for an interpreter frame and a pointer into the method zone for a machine code frame.

The first frame in a stack page is the baseFrame and is marked as such by a saved fp being its stackPage, in which case the first word on the stack is the caller context (possibly hybrid) beneath the base frame.

# CHAPTER 10

# Adding Static Methods

In this chapter we will show how you can develop a prototype version of static methods in Pharo. A static method is a method with no lookup. It means that the call site defines the exact class where the method is defined. The VM has just to grab the method from such a class.

For this introduction, we will

- define a new bytecode
- extend the bytecode builder
- extend the bytecode interpreter to handle this bytecode

## 10.1 Bytecode table

Since we will add a bytecode we start to have a look at the bytecode table. You will find it in the method `StackInterpreter class >> #initializeBytecodeTableForSistsV1`

This table links a bytecode and the method that defines its behavior.

What you can see is that the bytecodes 246 and 247 are free.

```
...
    (243    extStoreReceiverVariableBytecode)
    (244    extStoreLiteralVariableBytecode)
    (245    longStoreTemporaryVariableBytecode)

    (246 247  unknownBytecode)

    "3 byte bytecodes"
    (248    callPrimitiveBytecode)
    (249    extPushFullClosureBytecode)
...
```

We will use the bytecode 246. Once we will have extended the interpreter we will come back and modify such a table.

## 10.2 About method execution

Let us study a bit the normal message send bytecodes. For a default late bound message

- the receiver and args are on the stack
- the method selector is stored in the method literal frame

```

    (128 143  sendLiteralSelector0ArgsBytecode)
    (144 159  sendLiteralSelector1ArgBytecode)
    (160 175  sendLiteralSelector2ArgsBytecode)
```

The table tells us that send bytecodes range from 128 to 175. Such bytecodes are compact in the sense that they encode their number of arguments. In addition, they encode the place in the literal frame where the selector of the method to be looked up is placed.

For example, 128 means that the selector is located in the first place of the literal frame.

```
128 bitAnd: 16rF
> 0
```

## 10.3 Study 128

The interpreter method associated to bytecode 128 is `sendLiteralSelector0ArgsBytecode`

```
StackInterpreter >> sendLiteralSelector0ArgsBytecode
  "Can use any of the first 16 literals for the selector."

  | rcvr |
  messageSelector := self literal: (currentBytecode bitAnd: 16rF).
  argumentCount := 0.
  rcvr := self stackValue: 0.
  lkupClassTag := objectMemory fetchClassTagOf: rcvr.
  self assert: lkupClassTag ~= objectMemory nilObject.
  self commonSendOrdinary
```

What we see is that

- The message selector is extracted from literal frame using the the bytecode encoding.

- Then it sets the number of argument, here to zero
- It then looks the class up.
- And finally executes the method `commonSendOrdinary`

```
StackInterpreter >> commonSendOrdinary
  "Send a message, starting lookup with the receiver's class."
  "Assume: messageSelector and argumentCount have been set, and that
  the receiver and arguments have been pushed onto the stack,"
  "Note: This method is inlined into the interpreter dispatch loop."

  <sharedCodeInCase: #extSendBytecode>
  self sendBreakpoint: messageSelector receiver: (self stackValue:
    argumentCount).
  self doRecordSendTrace.
  self findNewMethodOrdinary.
  self executeNewMethod: false.
  self fetchNextBytecode
```

Once the method is found, it is executed by `executeNewMethod: false`. The argument means that the method should not be compiled by the JIT compiler. Then the interpreter fetches the next bytecode to be executed.

## 10.4 A first version of sendStaticLiteralMethod

The bytecode 246 is a two byte bytecode. Let us start to define a new method `sendStaticLiteralMethodBytecode` that defines the behavior of the static send. Since we want to avoid performing a method lookup we decide that the compiled method the send will execute should be stored in the method literal frame.

```
StackInterpreter >> sendStaticLiteralMethodBytecode
  "two bytecodes
    opecode
    literal offset "
  | methodLiteralOffset |
  methodLiteralOffset:= self fetchByte.
  newMethod := self literal: methodLiteralOffset.
  self executeNewMethod: true. "could be compiled"
  self fetchNextBytecode
```

This is a first version because the interpreter may use the values of other global variable such as the argument count. We will refine this definition later.

Now we declare that the bytecode 246 is defined by `sendStaticLiteral-Method`

```
...
    (243    extStoreReceiverVariableBytecode)
    (244    extStoreLiteralVariableBytecode)
    (245    longStoreTemporaryVariableBytecode)

    (246    sendStaticLiteralMethodBytecode)
    (247    unknownBytecode)

    "3 byte bytecodes"
    (248    callPrimitiveBytecode)
    (249    extPushFullClosureBytecode)
...
```

## 10.5 Compiling the VM

Let us check that our additions do not break the VM build - so far we nearly do anything that could but this way you can practice. Note that we only compile the VM interpreter without the JIT compiler.

- First save your code, the build will take the current branch has input.
- Go to the iceberg folder in the pharo-local folder and execute the following. Here we asked to grab the binaries of external projects to make the compilation faster.

```
cmake -S iceberg/pharo-vm -B build -DFLAVOUR=StackVM
    -DPHARO_DEPENDENCIES_PREFER_DOWNLOAD_BINARIES=TRUE
```

Then we compile the Vm and the result will be in the build folder.

```
cmake --build build --target=install
```

We can now launch the resulting VM to execute your image as follows:

```

./build/build/dist/Pharo.app/Contents/MacOS/Pharo ../YourImage.image
    --interactive
```

Note that you will have to rebuild the VM in the following. Before recompiling to not forget to save your code and remember that the build is taking the current branch as input.

## 10.6 Getting a compiled method

In this hands on, we focus on the virtual machine logic therefore we do not want to modify the syntax of Pharo. Still we need a way to get compiled methods with the new bytecode.

The Pharo compiler supports a bytecode builder, using the pragma `opal-BytecodeMethod` we can create the body of a method has the compiler would do.

For example the following method `exampleIRBuilder` just returns 2.

```
MyXP >> exampleIRBuilder

  <opalBytecodeMethod>
    ^ IRBuilder buildIR: [:builder |
      builder
        pushLiteral: 2;
        returnTop ]
```

Now we can just execute the method.

```
MyXP new exampleIRBuilder
> 2
```

Here is the definition of `factorial`, we call it `lateBoundFactorial` since we will define alternate versions using static message sends later.

```
Integer >> lateBoundFactorial

  <opalBytecodeMethod>

  ^ IRBuilder buildIR: [ :builder |
      builder
        pushReceiver;
        pushLiteral: 1;
        send: #'<=';
        jumpAheadTo: #gogogo if: false;

        "Base case"
        pushLiteral: 1;
        returnTop;

        "Recursive case"
        jumpAheadTarget: #gogogo;
        pushReceiver;
        pushReceiver;
        pushLiteral: 1;
        send: #-;
        send: #lateBoundFactorial;
        send: #*;
        returnTop ]
```

Notice that here this is the default method passing message semantics.

To support static calls, we will define a new IR instruction in addition to the bytecode to be able to define static sends.

## 10.7 Fixing some Pharo logic

Before continuing we should fix the method `refersToLiteral:` because it can loop if if literal frame contains a compiled method and this is what we want to do for our solution.

```
CompiledCode >> refersToLiteral: aLiteral [
  "Answer true if any literal in this method is literal,
  even if embedded in array structure."

  1 to: self numLiterals - self literalsToSkip do: [ :index |
    "exclude selector or additional method state (penultimate slot)
    and methodClass or outerCode (last slot)"
    (self literalAt: index) == self ifFalse: [
      ((self literalAt: index) refersToLiteral: aLiteral) ifTrue: [
        ^ true ] ] ].

  ^ false
]
```

## 10.8 Extending the IRBuilder

We extend the builder with the new message `sendStatic:`.

```
IRBuilder >> sendStatic: aMethod
  ^ self add: (IRSendStatic sendStatic: aMethod )
```

We define a new instruction subclass of `IRInstruction`. This instruction will refer to the invoked method.

```
IRInstruction << #IRSendStatic
  slots: { #calledMethod };
  tag: 'IR-Nodes';
  package: 'OpalCompiler-Core'
```

```
IRSendStatic >> calledMethod
  ^ calledMethod
```

```
IRSendStatic >> calledMethod: aCompiledMethod
  calledMethod := aCompiledMethod
```

We also define a class method

```
IRSendStatic class >>
```

We define the corresponding methods to support the interaction with the Visitors who are responsible for compilation.

```
IRSendStatic >> accept: visitor

  visitor visitStaticSend: self
```

```
Visitor >> visitStaticSend: anIRStaticSend

  self subclassResponsibility
```

```
IRPrinter >> visitStaticSend: send

  stream nextPutAll: 'staticSend: '.
  send calledMethod selector printOn: stream.
```

## 10.9 Study the Translator

Before we extend the code translator to generate the adequate bytecode, let us get inspiration from the method `visitSend:`.

We see that `visitSend:` is basically `gen send: send selector`.

```
IRTranslator >> visitSend: send

  send superOf
    ifNil: [ gen send: send selector ]
    ifNotNil: [ :behavior |  gen send: send selector toSuperOf:
    behavior ]
```

The `send:` method of the `IRBytecodeGenerator` uses the selector to send adequate information to the bytecode encoder.

```
IRBytecodeGenerator >> send: selector
  | nArgs |
  nArgs := selector numArgs.
  stack pop: nArgs.
  ...
  encoder genSend: (self literalIndexOf: selector) numArgs: nArgs
```

The method `send:` is basically a send to `genSend:numArgs:`.

```
EncoderForSistaV1 >> genSend: selectorLiteralIndex numArgs: nArgs


  ...
  (selectorLiteralIndex < 16 and: [nArgs < 3]) ifTrue:
    ["128-143 1000 iiii     Send Literal Selector #iiii With 0
    Argument
      144-159 1001 iiii     Send Literal Selector #iiii With 1
    Arguments
      160-175 1010 iiii     Send Literal Selector #iiii With 2
    Arguments"
     stream nextPut: 128 + (nArgs * 16) + selectorLiteralIndex.
     ^ self].
  ...
```

## 10.10 Translator extension

We make sure that the IRFix visitor does not raise an error by defining the method `visitStaticSend:` doing nothing.

```
IRFix >> visitStaticSend: anIRStaticSend
```

We extend the translator by adding the following `visitStaticSend:`

```
IRTranslator >> visitStaticSend: anIRStaticSend

  gen sendStatic: anIRStaticSend calledMethod
```

We define the method `sendStatic:`as follows:

```
IRBytecodeGenerator >> sendStatic: aMethod

  | nArgs |
  nArgs := aMethod numArgs.
  stack pop: nArgs.
  encoder genSendStatic: (self literalIndexOf: aMethod)
```

We finally emit the new bytecode: it basically emits the bytecode 246 followed by the literal frame offset in which the compiled method is stored. A better version should do a bit of validation.

```
EncoderForSistaV1 >> genSendStatic: methodLiteralOffset

  stream
    nextPut: 246;
    nextPut: methodLiteralOffset
```

## 10.11 Testing

Now we define a simple method using a static send. This method adds one to the receiver.

```
Integer >> staticPlus

  <opalBytecodeMethod>

  ^ IRBuilder buildIR: [ :builder |
      builder
        pushReceiver;
        pushLiteral: 1;
        sendStatic: (SmallInteger >> #'+');
        returnTop ]
```

```
1 staticPlus
> 2
```

## 10.12 The case of recursion

Since we are compiling a recursive method (factorial), we need a way so that the literal frame of this method refers to the compiled method itself.

For this as a temporarily solution we will introduce a placeholder that later we will patch. Here is a definition of factorial where the recursive call is static.

```
Integer >> staticBoundRecursiveFactorial

  <opalBytecodeMethod>
  1halt.
  ^ IRBuilder buildIR: [ :builder |
      builder
        pushReceiver;
        pushLiteral: 1;
        send: #'<=';
        jumpAheadTo: #gogogo if: false;

        "Base case"
        pushLiteral: 1;
        returnTop;

        "Recursive case"
        jumpAheadTarget: #gogogo;
        pushReceiver;
        pushReceiver;
        pushLiteral: 1;
        send: #-;
        sendStatic: (StaticRecursiveMethodPlaceHolder new selector:
    #staticBoundRecursiveFactorial);
        send: #*;
        returnTop ]
```

We have to define the class `StaticRecursiveMethodPlaceHolder`.

```
Object << #StaticRecursiveMethodPlaceHolder
  slots: {#selector};
  ...
```

```
StaticRecursiveMethodPlaceHolder >> numArgs

  ^ selector numArgs
```

```
StaticRecursiveMethodPlaceHolder >> selector: aString
  selector := aString
```

Now we patch the `generate:` method to substitute the placeholder by the compiled method.

```
IRMethod >> generate: trailer

  | irTranslator |
   irTranslator := IRTranslator context: compilationContext trailer:
    trailer.
  irTranslator
    visitNode: self;
    pragmas: pragmas.
  compiledMethod := irTranslator compiledMethod.
  compiledMethod literals doWithIndex: [ :e :index |
    (e isKindOf: StaticRecursiveMethodPlaceHolder)
      ifTrue: [ compiledMethod literalAt: index put: compiledMethod
    ] ].
  self sourceNode
    ifNotNil: [
      compiledMethod classBinding: self sourceNode methodClass
    binding.
      compiledMethod selector: self sourceNode selector ]
    ifNil: [
      compiledMethod classBinding: UndefinedObject binding.
      compiledMethod selector: #UndefinedMethod ].
  ^ compiledMethod
```

## 10.13 Better sendStaticLiteralMethodBytecode

The first definition of `sendStaticLiteralMethodBytecode` was brittle. Indeed the interpreter has some invariants and use about global variables that we did not reset correctly.

This is the case for `argumentCount`. It is used by primitives to check how to access the stack and know how many elements to pop, and generally to check that the stack gets balanced after execution.

The second case is `primitiveIndex` and `primitiveFunctionPointer`.`primitieIndex` should be loaded for each interpreted method. The function `executeNewMethod:` assumes that this index is set during lookup. Thus, if we don't set it, the value will be the one of the last method/primitive called leading to strange bugs.

Here is the new version of the static send bycode logic.

```
StackInterpreter >> sendStaticLiteralMethodBytecode

  "2 Byte Bytecode
    1st Byte: opcode
    2nd Byte: literal offset of the method"

  | methodLiteralOffset primitiveIndex |
  methodLiteralOffset := self fetchByte.
  newMethod := self literal: methodLiteralOffset.
  "argumentCount is used by primitives to
     - check how to access the stack and
     - know how many elements to pop,
  and generally to check that the stack gets balanced after
    execution"
  argumentCount := self argumentCountOf: newMethod.

  "primitiveFunctionPointer needs to be loaded for each method
    interpreted.
  executeNewMethod: assumes that this is set during lookup
  Thus, if we don't set it, the value will be the one of the last
    method/primitive called"
  primitiveIndex := self primitiveIndexOf: newMethod.
  primitiveFunctionPointer := self functionPointerFor:
    primitiveIndex inClass: nil.

  self executeNewMethod: true.
  self fetchNextBytecode
```

https://github.com/evref-inria/pharo-vm/pull/2/files

## 10.14 Bench

Here are the three different versions of factorial that we can now benchmark.

```
Integer >> lateBoundRecursiveFactorial

  <opalBytecodeMethod>
  ^ IRBuilder buildIR: [ :builder |
      builder
        pushReceiver;
        pushLiteral: 1;
        send: #'<=';
        jumpAheadTo: #gogogo if: false;

        "Base case"
        pushLiteral: 1;
        returnTop;

        "Recursive case"
        jumpAheadTarget: #gogogo;
        pushReceiver;
        pushReceiver;
        pushLiteral: 1;
        send: #-;
        send: #lateBoundRecursiveFactorial;
        send: #*;
        returnTop ]
```

```
Integer >> staticBoundRecursiveFactorial

  <opalBytecodeMethod>
  1halt.
  ^ IRBuilder buildIR: [ :builder |
      builder
        pushReceiver;
        pushLiteral: 1;
        send: #'<=';
        jumpAheadTo: #gogogo if: false;

        "Base case"
        pushLiteral: 1;
        returnTop;

        "Recursive case"
        jumpAheadTarget: #gogogo;
        pushReceiver;
        pushReceiver;
        pushLiteral: 1;
        send: #-;
        sendStatic: (StaticRecursiveMethodPlaceHolder new selector:
    #staticBoundRecursiveFactorial);
        send: #*;
        returnTop ]
```

```
Integer >> staticBoundRecursiveFactorialHardcore

  <opalBytecodeMethod>
  ^ IRBuilder buildIR: [ :builder |
      builder
        pushReceiver;
        pushLiteral: 1;
        sendStatic: (SmallInteger >> #'<=');
        jumpAheadTo: #gogogo if: false;

        "Base case"
        pushLiteral: 1;
        returnTop;

        "Recursive case"
        jumpAheadTarget: #gogogo;
        pushReceiver;
        pushReceiver;
        pushLiteral: 1;
        sendStatic: (SmallInteger >> #'-');
        sendStatic: (StaticRecursiveMethodPlaceHolder new selector:
    #staticBoundRecursiveFactorial);
        sendStatic: (SmallInteger >> #'*');
        returnTop ]
```

Need more discussions

```
[17 lateBoundRecursiveFactorial.] bench. "'2774597.961 per second'"
[17 staticBoundRecursiveFactorial.] bench.  "'3693598.280 per
    second'"
[17 staticBoundRecursiveFactorialHardcore.] bench. "'2170939.636 per
    second'"
```

Note that the staticBoundRecursiveFactorialHardcore is slower because the primitives *, -, + are extremely optimized by the VM and do not result in message sends.

## 10.15 Limits and conclusion

There is clearly some more effort to obtain a full working solution. For example, managing the code changes and recompilation of the methods.

This tutorial misses

- syntax support
- JIT support
- invalidation if the called method changes
- indirect recursion support

# Bibliography

[1] S. Ducasse, D. Zagidulin, N. Hess, D. C. O. written by A. Black, S. Ducasse, O. Nierstrasz, D. P. with D. Cassou, and M. Denker. *Pharo by Example 5*. Square Bracket Associates, 2017.

[2] E. Miranda, C. Béra, E. G. Boix, and D. Ingalls. Two decades of Smalltalk VM development: live VM development through simulation tools. In *Proceedings of International Workshop on Virtual Machines and Intermediate Languages (VMIL'18)*, pages 57–66. ACM, 2018.

[3] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 291–300, New York, NY, USA, 1998. Association for Computing Machinery.