Clément Béra ~ Smalltalk, Tips 'n Tricks

Spur's new object format

16 - *Thursday* - *Jan 2014*

Posted by Clement Bera in Spur

≈ **11 Comments**

Hey folks,

As some of you may know, the Cog VM will soon feature a new MemoryManager, Spur. Here's Eliot Miranda blog post about it: http://www.mirandabanda.org/cogblog/2013/09/05/a-spur-gear-for-cog/. This new memory manager includes a new object format. We will describe here first the old object format and then the new object format.

I'd like to thank Jean-Baptiste Arnaud, Igor Stasenko and Stéphane Ducasse that drew some of the figures of this article.

Reminder 1: Conversion table

In 32 bits, 1 word = 32 bits = 4 bytes.
In 64 bits, 1 word = 64 bits = 8 bytes.

the old object format

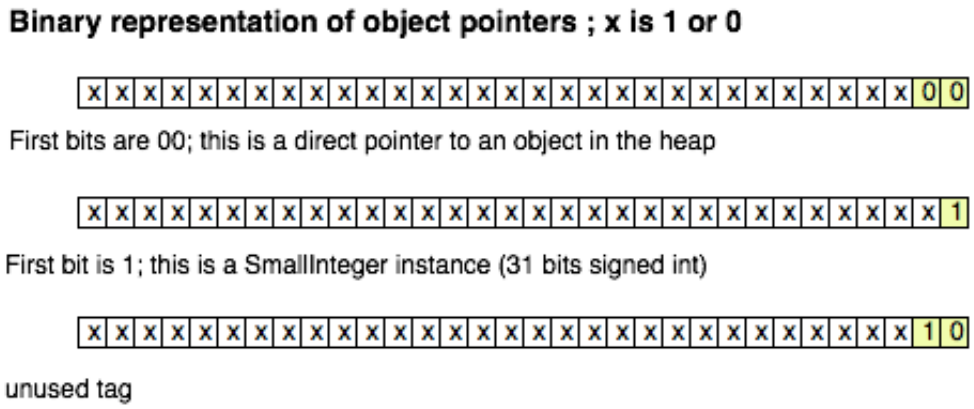
Object pointer

In 32 bits, a pointer is a 32 bits integer. The pointer value is the address of a memory location. In the real world, an address is a street's number, a street and a city. However, all memory blocks live in the same city and in the same street. Therefore, you only need to precise the street's number to find a memory block. This is why a pointer address is only a number. True story.

A pointer can target any byte. In 32 bits, this means any byte between 0 and 2^32. However, in object-oriented virtual machines, memory locations are word-aligned for performance. This means that a pointer to a memory location can only have the address of 1 byte out of 4 and therefore is always a multiple of 4. In a binary representation of a pointer, this means that an object-oriented pointer (oop) always has 00 as lower bits.

The CogVM takes advantage of this and uses the 2 bits to mark specific objects. More precisely, SmallInteger are marked (a common term is SmallInteger are **tagged**). This permits to limit the size of SmallInteger objects in memory. This is why in Pharo, all objects are passed by reference,

except SmallInteger that are passed by value. In addition, it permits in the JIT to map easily integer operation to native code signed operations.



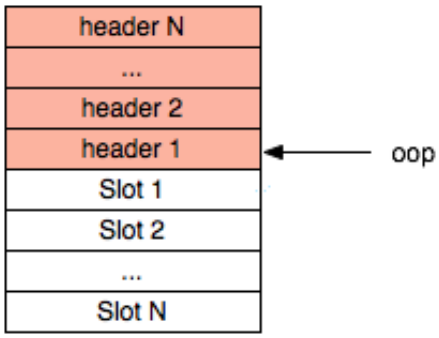
Note:

- An object which is directly encoded in the pointer and has no memory location, as SmallInteger, is called an **immediate object**.
- 64 bits is not supported by the old object format.

Object header

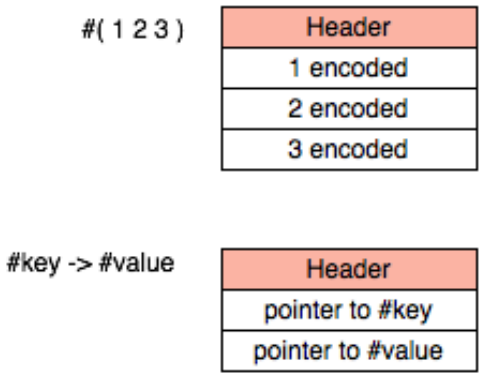
In the memory, an object (except SmallIntegers) is represented by a header of a certain size and a certain number of slots (slots are sometimes also called fields). A pointer to an object always targets the first header word.

The header corresponds to some kind of object metadata for the VM to know more about the object, as for example its size in memory. The fields corresponds to information about the state of the objects (Considering an object is a state and a behavior, the state informations is stored in the fields).



An object can have different kind of fields.

Examples:



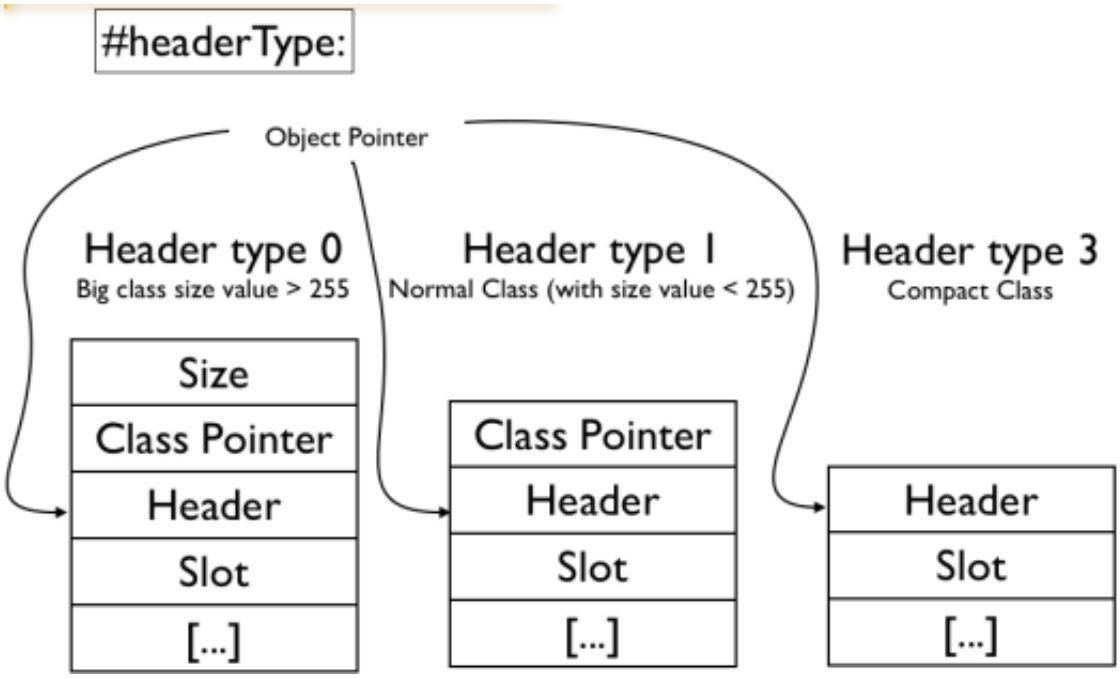
Here on the one hand there's an Array which has a different number of fields depending on how you created it:

Array new: 5 "Array with 5 fields"
Array new: 15 "Array with 15 fields"

These objects are called variable sized objects. They have **indexable fields**, which are accessed through primitives (#at: for example). On the other hand, there are objects with instance variables. This kind of objects has always the same field size, which is the number of instance variables. These objects are called fixed-sized objects. They access their **fixed fields** through instance variables.

A Variable Sized object is created when its class is defined with #variableSubclass: keyword instead of #subclass: (See class definition in Pharo of Association and Array for example).

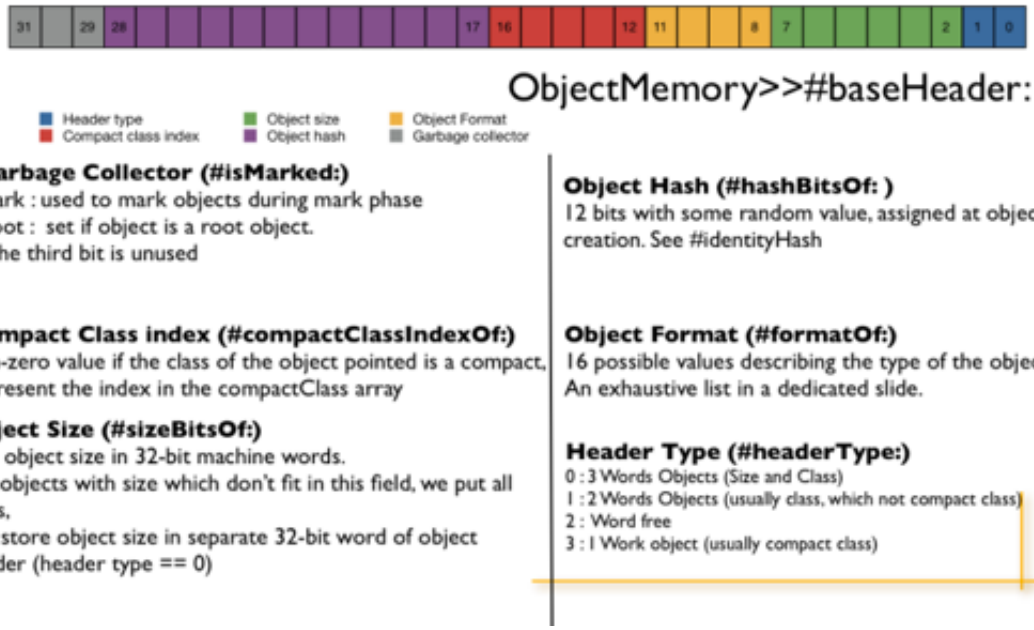
In the current object format, the header can have 3 different sizes, 1, 2 or 3 words.



- Header type 1 is the standard one. It has one word that is a pointer to the object's class, and a header word, that will be explained below
- Header type 0 is a specific header for objects that have more than 255 fields. Usually the

object's field size is encoded in the header word, but for big objects there's not enough rooms, therefore an additional word is allocated
- Header type 3 is a specific header for very common classes. Usually the object knows its class thanks to its class pointer in the second header word. For these objects, a bit pattern permits to determine the object's class instead of the pointer. This permits to save 1 word in memory per object. As only very common objects have this header, this trick allows the VM to save a lot of space

Now let's look into the base header word:



As you can see:

- 3 bits are used for garbage collection
- 12 bits are used for the object's hash
- 5 bits are used for the compact class index
- 4 bits are used for the object's format
- 6 bits are used for the object's size. If this is not enough the header is extended to header type 0
- the last 2 bits are reserved to mark the header type

In the 3 GC (Garbage collector) bits, one is used to mark if the object is a root (the object graph is traversed from the roots to find alive objects). Another one is used to mark the object in the GC marking phase.

The 12 bits for hash are cool, but we need more bits. For big hashed collection (Set, Dictionary), it is very common that several objects have the same hash. This results in lots of conflicts, decreasing the collection's performance.

The 5 bits for compact class index are used only in header type 3. These bits allow the user to specify 16 classes that will have instances 1 word smaller. These classes are specified in the CompactClassArray. For example, Array, Point and Rectangle are compact classes. One can have the list of compact classes by evaluating 'Smalltalk compactClassesArray' in a workspace.

The 4 bits for the object format specify how to access the object's field. Previously in the post, we have already talked about the Variable-sized objects and the fixed-sized objects. There are more kinds of objects. For example, you can have weak objects (object's fields are garbage collected if referenced only by a weak pointer). There's also a specific kind of fields for ByteArray, for WordArray and for CompiledMethod. This specification for CompiledMethod is not specific to Smalltalk, but to Cog (For example in HPS, the VisualWorks VM, CompiledMethods are regular fixed-sized objects).

Let's get deep into the subject. Let's introduce Spur's object format.

Spur's object format

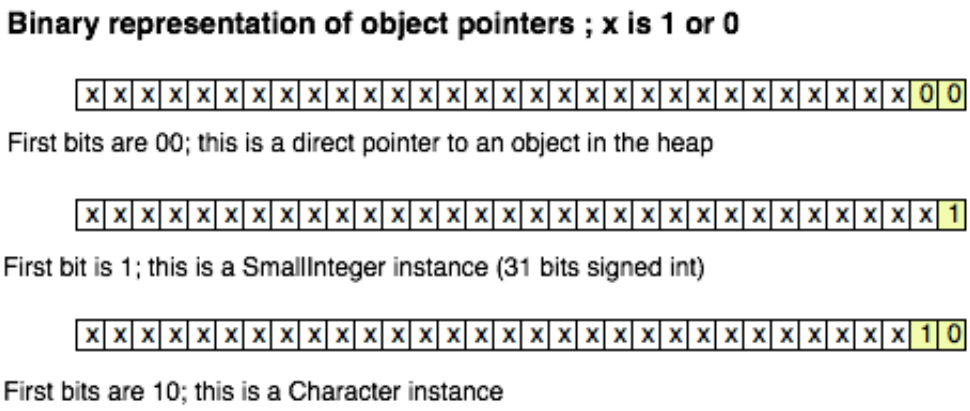
Object pointer

One thing to note is that Spur is an hybrid memory manager that supports both 32 bits and 64 bits. Most things Eliot implemented in Cog are hybrid anyway: the stack frame representation, the interpreter + JIT structure, … I would not be surprised if one day he tells me he had bought an hybrid car :-).

In both 32 and 64 bits, every object is 64 bits aligned. However, the object pointer representation is different.

In 32 bits

In 32 bits, the main improvement is the addition of immediate Characters. Let's see the new Object pointer representation:



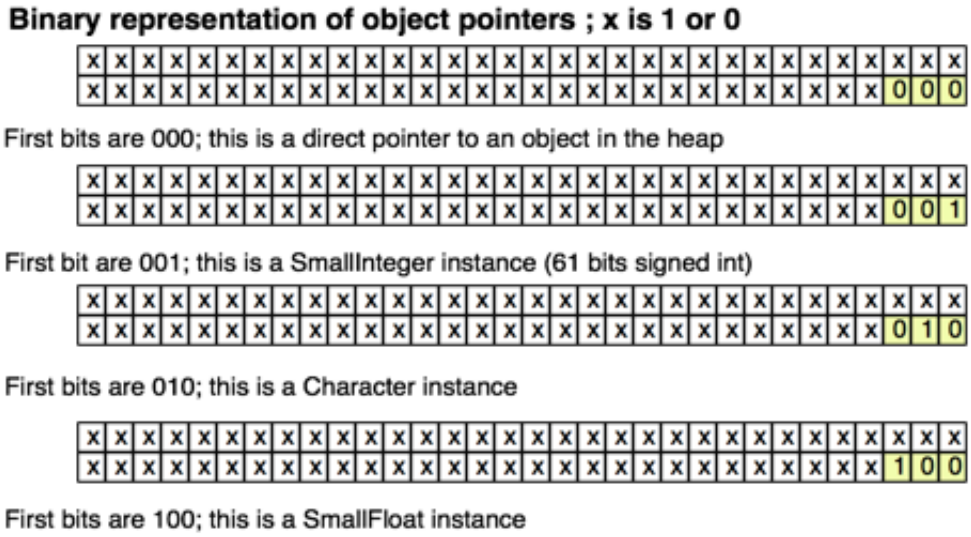
Basically immediate characters will mostly speed up String accessing, especially for WideString, since no instantiation needs to be done on at:put: and no dereference need be done on at:. It will save a little memory too (even though in most case it will save memory for only 256 objects which is not a lot).

In 64 bits

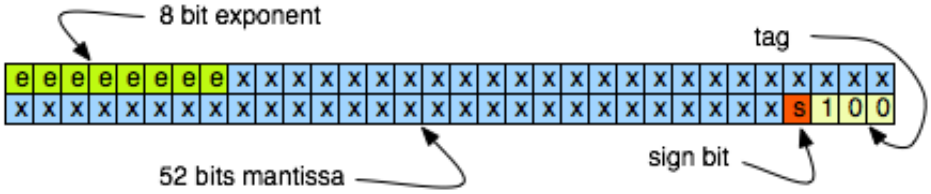
Due to 8 bytes alignment, in 64 bits one more bit is available to tag objects. This allows us to have additional immediate objects. In addition to the existing ones, Eliot added immediate floating pointers. It will mostly reduce the float's size in memory and speed up floating pointers

arithmetics (no instantiation for the result of each binary operation).

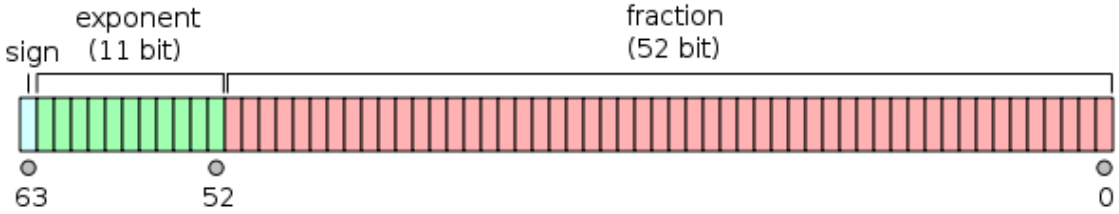
Here's a sum-up:



In details, the immediate float structure is:



One interesting thing to note is that SSE2 native instructions work with IEEE double precision floating pointer format:

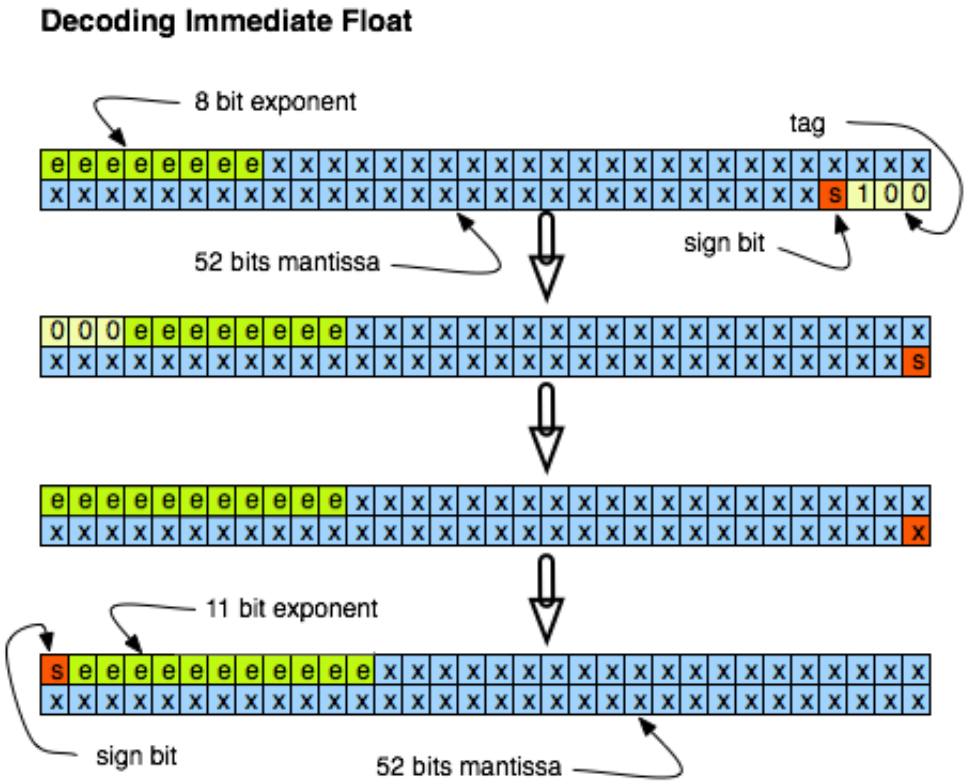


On the contrary to the IEEE representation, the Cog VM needs 3 tag bits. These bits are taken from the exponent, creating a floating pointer format as precise as the double precision IEEE floating pointers but with a smaller range. The range is around 1/8th of the double range (the middle 1/8th).

When I saw that my first thought was: why the hell is the sign bit the lower bit and not the higher bit ? Actually putting the sign bit there has 2 advantages. Firstly, it allows one to faster encode/decode Cog's floating pointer format to the IEEE double one, because offsetting the exponent can't overflow into the sign bit. Secondly, this permits to knows if the float is positive or negative just with an unsigned compare (<= 0x0f).

Of course, with SSE2 instructions and these tricks, I'm talking about speeding up floating pointers arithmetics in native code, the C implementation may not be very fast (especially there's no rotate in C), but we don't care. And if you care, please read the literature about JIT compilers and hot spots to understand why you should not care.

One last thing is the decode/encode features to be able to use SSE2 native instructions. Decoding a float happens by shifting away the tag, by adding the exponent offset and lastly by rotating the sign bit. Encoding is the exact opposite.



Note:

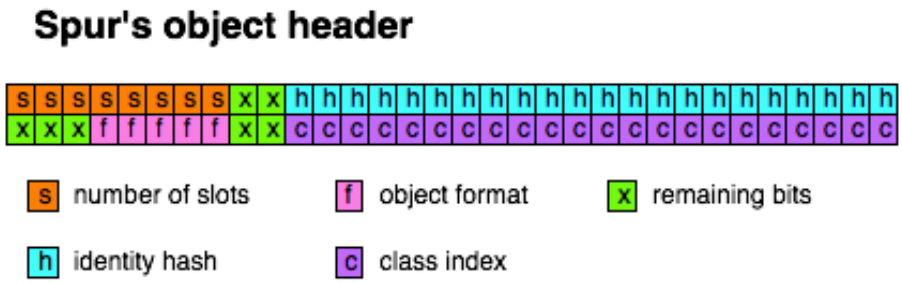
If one needs more immediate objects, the bit pattern 110 is still not used. However, there was a discussion on switching the bit pattern between Character and Float in 64 bits, in order to reserve both 110 and 010 for Floats, giving it an additional exponent bit over its current implementation.

Therefore, to add extra immediate objects, one could just limit SmallInteger to 61 bits instead of 63 to free 3 new tags for immediate objects. It may make sense to keep 31 bits for SmallInteger in 32 bits, but in 64 bits this argument is irrelevant. However I don't really see why one would need this many different immediate objects.

Object header

It used to be that there were 3 different header types. Now there's only one that can be extended.

The new header is 64 bits length, which means it is 2 words in 32 bits and 1 word in 64 bits. Here's its (colorful) structure:



- The 8 red/orange bits are for object's number of slots/fields. If the object has more than 254 fields, then an additional 64 bits word is allocated as a header extension with the correct size. In this case, the 8 bits have the value 255 to let the VM know that there is a header extension (in the previous model, there was a header type field, which does not exist any more).
- The 22 light blue bits are for the identityHash. This means the identityHash of objects is now 10 bits bigger, avoiding most hash interferences. This will considerably speed up large hash collections
- The 5 pink bits are for the new object formats. These bits have their own section below
- The 22 purple bits are for the class index. In the old model, most objects had a pointer toward their class. However, in 64 bits, it does not make sense to waste 64 bits just for the class information. Therefore, there's somewhere deep in the VM a class table. This class index is the index of the class in the table.
- The 7 green remaining bits are allocated for different reasons:
  - 1 bit is reserved for immutability.
  - 1 bit is reserved to mark the object as pinned. Pinning objects is a new feature, also introduced with Spur Memory manager. Basically, a pinned object is an object that cannot move in memory. Usually, objects are moved around by the GC. But not the pinned object. A pinned object will have this bit set.
  - 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.

Object format

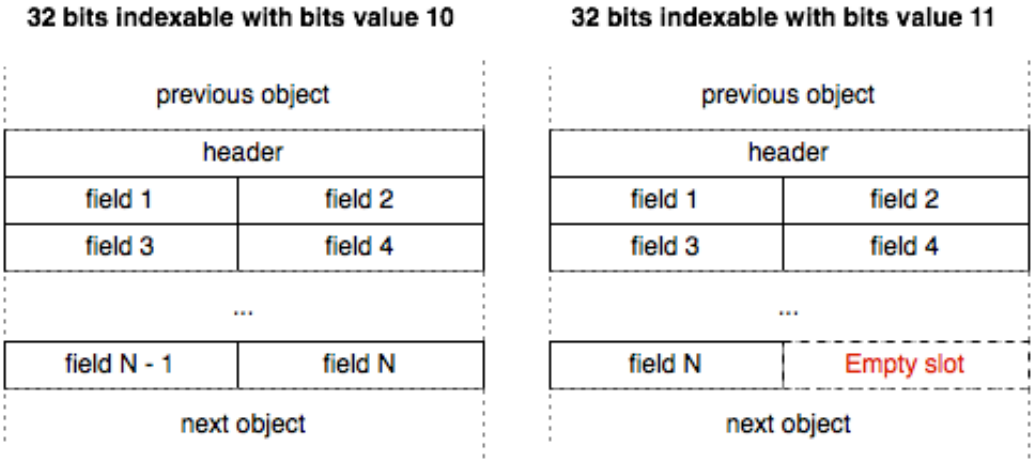
The new object format follows this table:

bits value	object format	example(s)
0	0 sized object	nil, true, false
1	fixed-sized object with inst vars	Point
2	variable sized objects with no inst vars	Array
3	variable sized objects with inst vars	MethodContext
4	weak variable sized objects	WeakArray
5	weak fixed sized objects with inst vars	Ephemeron
6, 7, 8	unused	
9	64 bits indexables	
10 - 11	32 bits indexables	
12 - 15	16 bits indexables	
16 - 23	8 bits indexables	ByteArray
24 - 31	compiled methods	CompiledMethod

The first fields are similar to the existing ones, except 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.

One new thing is the 16 and 64 bits indexable arrays. Right now Pharo has ByteArray and WordArray, which are specific arrays that can only store respectively 8 bits and 32 bits unsigned integers. Now we have 2 more for 16 and 64 bits unsigned integers. This is important for example for graphics where each pixel is stored in an array (See ShortRunArray, RunArray implementations and uses).

Lastly, you may have noticed that some object formats have several possible bits value. For instance, 32 bits indexable can have 10 or 11 as a bits value. This is because the bits value tips the VM on where is the last slot of the object in its last word (see figure below).



A few class index details

As we explained, an object has now in its header its class index instead of its class. This leads to unobvious issues.

issue 1: When I create a new object, when sending #basicNew, the class used to tell object "Oh, here's my address, take it just in case you need me". And the object knew the class address. Now, the problem is that the class does not know its index, so it cannot tell its new instance about it. One solution could be to walk over the class table and look for its index. But this happens at *each* instantiation, so the class cannot waste time walking over the whole table. Here's the trick: a class identityHash is now its index. This fits perfectly well because both the identityHash and class index slot are 22 bits length. In addition, this provides the additional benefit that every class has a different identityHash. The probability that 2 classes has the same identityHash is 0, whereas for most objects, even if this probability is low, the probability does exist.

issue 2: When I send the message #class to an object, this message send is slower because you need to fetch the class from the class table instead of having a direct reference. This is true. However, 2 tricks avoid most of the slow down.

- **Trick 1:** most common classes are put in the first page of the class table. Ah yeah. Because we are in the low level world. And there, you cannot just say "Hey cpu, I want a collection that grows and shrinks according to how many values there are in there". The only thing you can ask the cpu is "I want x bytes of memory". Therefore, the class table is a linked list of pages, with a class list on each page. So most common classes are put on the first page to avoid walking over all the pages to fetch an object's class.
- **Trick 2:** a massive part of Cog speed boost comes from the inline caches. An inline cache needs to check an object class to know if it can reuse the previous lookup result. Comparing class' pointers is not easy, because classes are moved in memory at each garbage collection. The class index now allows the inline cache to check only if the receiver class index is the same than in the previous lookup to be able to reuse the previous lookup result. This is a quite important optimization, and it speeds up all message sends. Therefore the slow down on #class is insignificant compared to this speed up.

Note:

As some of you may have notice, I omitted to talk about the compiled methods memory representation, especially its specific header. This is because a new byte code set will be deployed in the Cog VM in the near future. The compiled method memory representation will change at this point. Therefore I will talk about it in my future post about the new byte code set (no need to talk now about an almost outdated feature).

thoughts on "Spur's new object format"

1. *said:* Philippe Back

   January 19, 2014 at 10:44 am

   Very nice explanations. I am eager to see the speed boost!

REPLY

- *said:***clementbera**

  January 19, 2014 at 11:31 am

  With Spur, Cog is 1.7x times faster (-40% time spent to run benchs). I think the speed boost is mainly due to the new GC and the class index.

  REPLY

2. *said:*Jecel Assumpcao Jr

   January 19, 2014 at 5:17 pm

   Great article! One small detail is that the second figure shows OOPs pointing to the first slot while the fourth figure shows them pointing to the header word (which is the case).

   Does the 64 bit version of Spur really use 4 bit tags for OOPs? That would imply that objects are aligned on 16 byte blocks rather than 8 byte blocks.

   REPLY

3. *said:***clementbera**

   January 19, 2014 at 5:54 pm

   Thanks !

   I fixed the figure you mention.

   You are right I published the post too fast objects are 8 byte aligned not 16 bytes aligned. I fixed the figures and the texts. I don't know why I've always thought than in 64 bits you had 4 bits to tag oops (I've always worked in 32 bits VMs). Thank you very much for noticing me this was an important mistake.

   REPLY

4. Pingback: 5000 views ! Thank you ! | Clément Béra
5. Pingback: Squeak / Pharo VM documentation links | Clément Béra
6. Pingback: VM Learning: Memory Management | Clément Béra
7. *said:*Ben Coman

   December 2, 2017 at 10:39 pm

Thanks for this article. I have referred to it multiple times. A query…
How are the bits of forwarders represented? Are they stored as part of the object-pointer, or as part of the object-header? Maybe you could extend the article with pictures of this.

REPLY

8. *said:*Ben Coman

December 2, 2017 at 10:43 pm

For the "7 green remaining bits" it would be useful to see on diagram which bit goes in which location.

REPLY

9. *said:*Ben Coman

December 2, 2017 at 11:49 pm

Which parts of the object header are (im)mutable, and by VM or Image? For example, pinned/mutable/GC bits obviously mutate. ON the other hand, when adding an instance variable to a class, IIUC a new object is created and data copied to it then #become:'d. So are the slots bits in practice never modified after an object is created? Does changing the class of an objects similarly copy & #become? (This is in consideration of a concurrent GC thread)

REPLY

- *said:***Clement Bera**

  December 4, 2017 at 7:27 am

  – primitive changeClassTo: only works with objects of same size and same format.
  – primitive become: has an option, copyHash (see the different methods in the image) which tells if become should mutate or not the identityHash.
  – primitive setIdentityHash:isBehavior: can set the hash of an object, and updates the class table if the second parameter is set to true.
  Aside from those 3 primitives, I would say…
  * Slots field: never mutated after creation (through it can be during creation, for instance the VM may create a large array and shorten it once created).
  * Hash field: initialized with 0, set up lazily by the VM at first use. So yes it can be mutated after object creation by the VM.
  * Class index: never mutated after creation.
  * Object format: never mutated after creation.
  * isRead-only/isPinned fields: mutated from Smalltalk, I don't think the VM mutates them
  * isGray/Marked/Remembered fields: mutated from the VM, cannot be mutated from Smalltalk (Though some sista inline primitives allow to set some of those fields with the possibleRoot instruction, some standard primitives can allow that too indirectly)

REPLY

1