Clément Béra ~ Smalltalk, Tips 'n Tricks

The Cog VM lookup

09 - *Friday* - *Aug 2013*

Posted by Clement Bera in Cog

≈ **3 Comments**

Hey folks,

Recently, I spent some time working on lookups for Smalltalk systems. It took me some time to get the Cog VM lookup optimizations, and I'd like to explain my understanding. Eliot Miranda has already explained some of it in his post Build me a JIT as fast as you can. His post is really good but when I first looked at it I didn't understand everything because there are some parts explained by showing slang and x86 assembly which were not familiar to me.

So here it is, the explanation of how the Cog VM lookup works.

Plan

- Lookup definition and Smalltalk details
- Naive lookup
- Global lookup cache
- Inline caches
- The next steps

Lookup definition and Smalltalk details

Lookup definition

When a message (receiver + selector + arguments) is sent, the method corresponding to the message selector is looked up through the inheritance chain.

Smalltalk details

What happens if no method is found during the lookup ?

Then the VM does a second time the lookup, sending this time the message 'doesNotUnderstand: aMessage', aMessage being the message not found. By default, it will trigger the Object>>doesNotUnderstand: implementation that raises an error. This doesNotUnderstand: method is so important in Smalltalk that all the optimization on the lookup were hacked to work with it (for example, you can store in a cache as a lookup result the doesNotUnderstand result).

What happens if the superclass of a class is incorrect during the lookup ?

The superclass of an object has to be nil or a Behavior. If it is nil, the lookup stops. If it is a Behavior, the lookup continues until it finds the appropriate method. It is not possible to put something else than nil or a behavior in the superclass slot of a class.
Example:
MyClass superclass: Object new "raises the error, superclass must be a class-describing object"
MyClass instVarNamed: #superclass put: Object new "Crashes the image"

What happens if the method dictionary of a class is incorrect during the lookup ?

The methodDictionary has to be an instance of MethodDictionary. Random behaviors happen if you put something else:
MyClass methodDict: Object new "Lookup is normal except it skips the current class methodDictionary
MyClass methodDict: 42 "Crashes the image"
MyClass methodDict: nil "This case in very strange. VM side, I see: 'MethodDict pointer is nil (hopefully due a swapped out stub) — raise exception #cannotInterpret:.'. Now I tried in Pharo 3, the image crashed. I tried in Squeak 4.3, I got an error 'oops'. oops ? What does that mean ? I looked carefully and it come from ClassDescription>>recoverFromMDFault (MD being for method dictionary)"

What happens if the compiledMethod found is not a compiledMethod but some random object ?

The VM tries to execute the object as a compiledMethod, triggering the method 'run: oldSelector with: arguments in: aReceiver'. Look at ObjectsAsMethodsExample in Pharo to have more details. However, some optimizations does not support this feature, making it quite slow in some case. For example, method activation using 'run:with:in:' cannot be optimized by Cog's JIT.

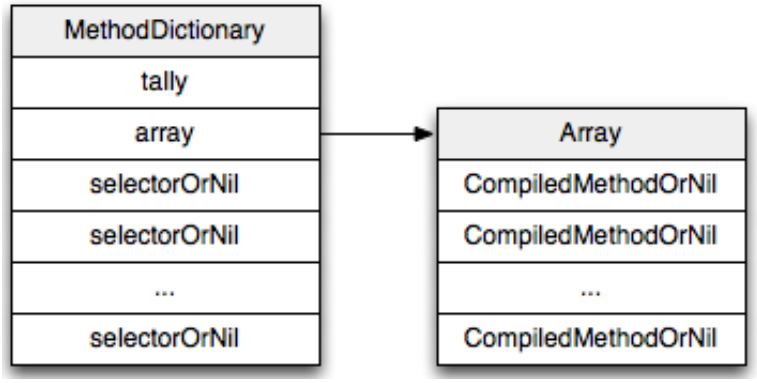
Naive lookup

This first case happens when it has been the first time since a huge while that the user looks up a selector for a specific class. The idea is to really do the lookup, going from class method dictionary to class method dictionary until you find the appropriate compiledMethod. Now of course it is done VM side, so it is not trivial.

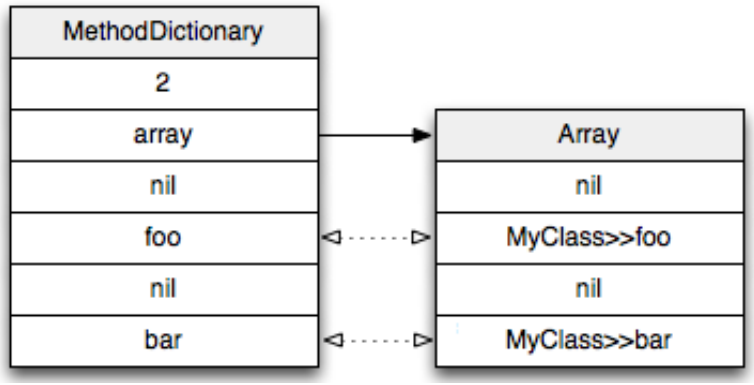
There is some kind of loop to walk up the class hierarchy (I removed non important data):

```
StackInterpreter>>lookupMethodInClass: class
    [currentClass ~= objectMemory nilObject] whileTrue:
        [dictionary := objectMemory fetchPointer: MethodDictionaryIndex
ofObject: currentClass.
        found := self lookupMethodInDictionary: dictionary.
        found ifTrue: [^currentClass].
        currentClass := self superclassOf: currentClass].
```

Then, you need to find the method in the Dictionary (`lookupMethodInDictionary:`). For that, you need to understand the low level implementation of the method dictionary.

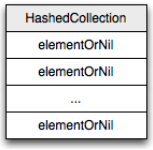


As you can see, the first field of the methodDictionary corresponds to the number of methods it has, the second field contains an array of methods, and the rest of the object contains the selector *ordered by hash*. There is a fixed mapping between the selector index in the methodDictionary and the index of its corresponding compiled method in the array. Below is an example.

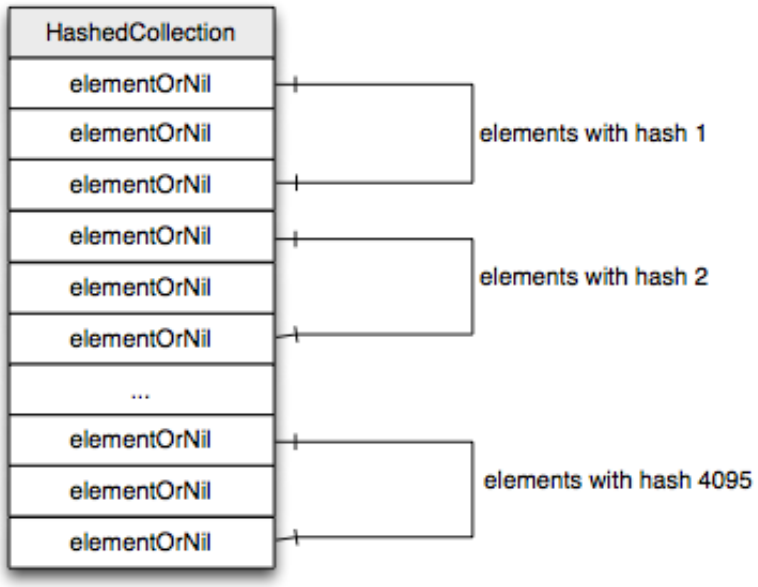


Now you also need to understand the low level representation of a hashed collection. The idea of hashed collection is to order the elements of the collection in the memory to access a specific element of the large collection faster than walking other the whole collection.

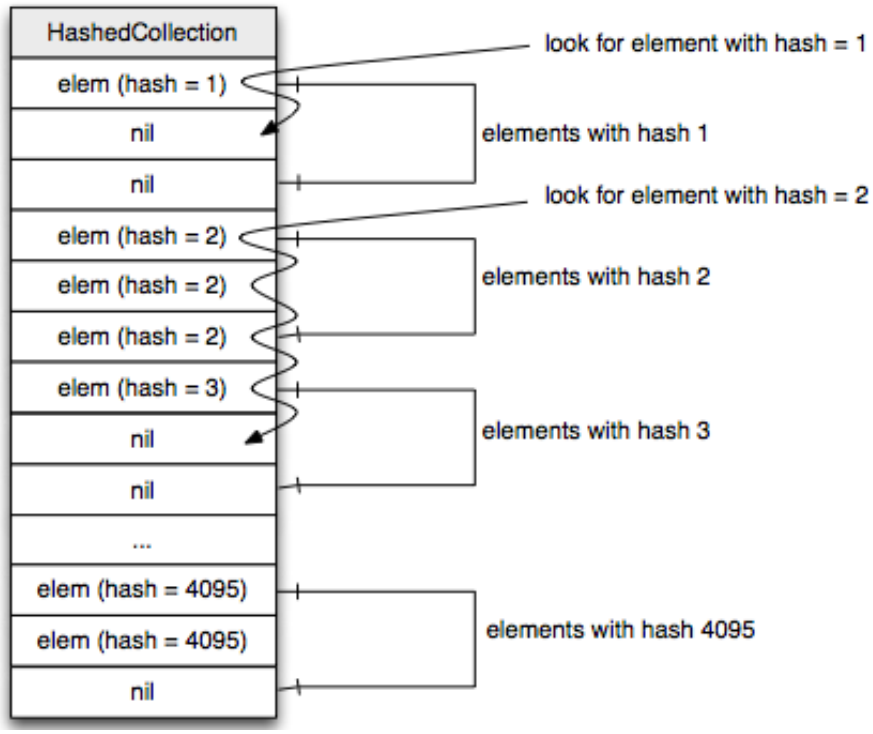
A hashed collection looks like that:



Now the idea is to order it by hash. However, Pharo having a 12bits hash, it is very common that objects has the same hash. Objects with the same hash are near each other in a so-called bucket.



To find an object, you then need to start iterating at the first object of your hash's number bucket until you find the object or a nil, meaning that the object is absent.

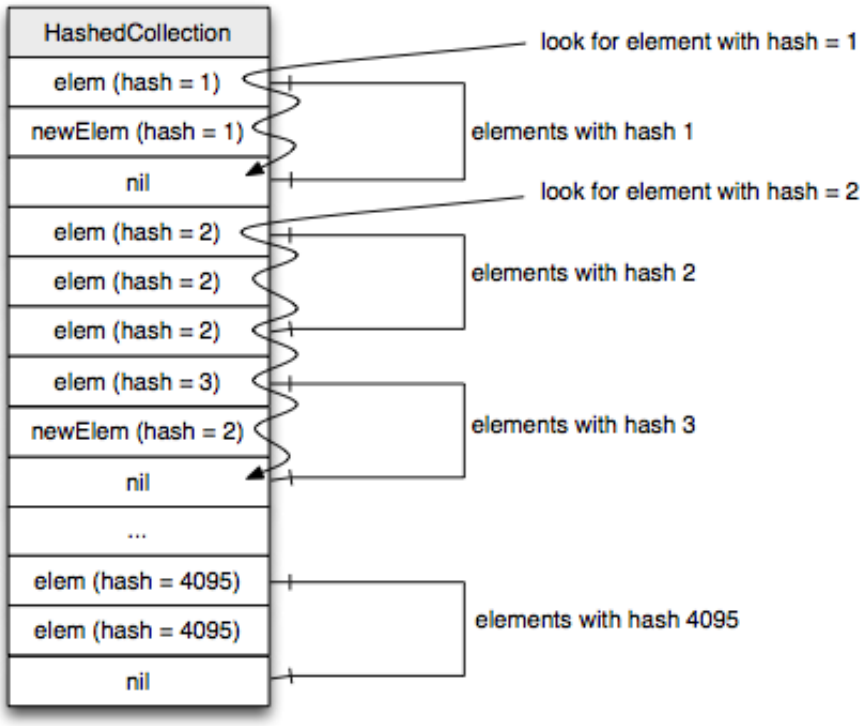


To start the iteration, you need to know the first index of the hash's number bucket of the object you are looking for. To find this index you need 2 informations:

- the size in memory of the hashed collection
- the hash of the object you're looking for

Then basically if the size of the collection is lower than the maximum hash's number, you use a subset of the object's hash bits to have an index from 1 to the hashed collection's size (and not from 1 to hash's max number, which would raise an OutOfBound error).

An efficient hashCollection would be full at 80%, letting quite some nil to stop the iteration and not taking too much memory space. However, rehashing all the collection each time you add an element is too expensive, so it is done only when the collection is full (then you increase its size and rehash it). So some elements of the collection are not placed in the correct bucket.



So the general idea is to find the selector in the methodDictionary, the selectorOrNil fields acting like a hashed collection, and then returns the compiled method which is in the method dictionary's array at the same index as the selector in the method dictionary.

Let's look at the code. I took the lookup from interpreter VM to show that the VM checks if the method is a primitive at lookup time and not at activation time. Again I removed stuff that I thought was not important to show:

- the test for objects found in methodDictionary that are not a compiledMethod, explained in Section *Lookup definition and Smalltalk details* (go look into VMMaker if you are interested in this case).
- the wrap around, which handles the specific case where the last bucket is full (then extra objects with last hash are put at the beginning of the collection) and also handles the case where there is no nil not to have an infinite loop (the hashed collection is full).
- the case where selectors are smallInteger for compact images

```
lookupMethodInDictionary: dictionary
| index mask nextSelector methodArray |
mask :=(self fetchWordLengthOf: dictionary) - SelectorStart - 1.
index := (mask bitAnd: (self hashBitsOf: messageSelector)) + SelectorStart.
```

```
[true] whileTrue: [
nextSelector := self fetchPointer: index ofObject: dictionary.
nextSelector = nilObj ifTrue: [^ false].
nextSelector = messageSelector ifTrue: [
 methodArray := self fetchPointer: MethodArrayIndex ofObject: dictionary.
 newMethod := self fetchPointer: index-SelectorStart ofObject: methodArray.
 primitiveIndex := self primitiveIndexOf: newMethod.
 ^ true].
index := index + 1].
```

Global lookup cache

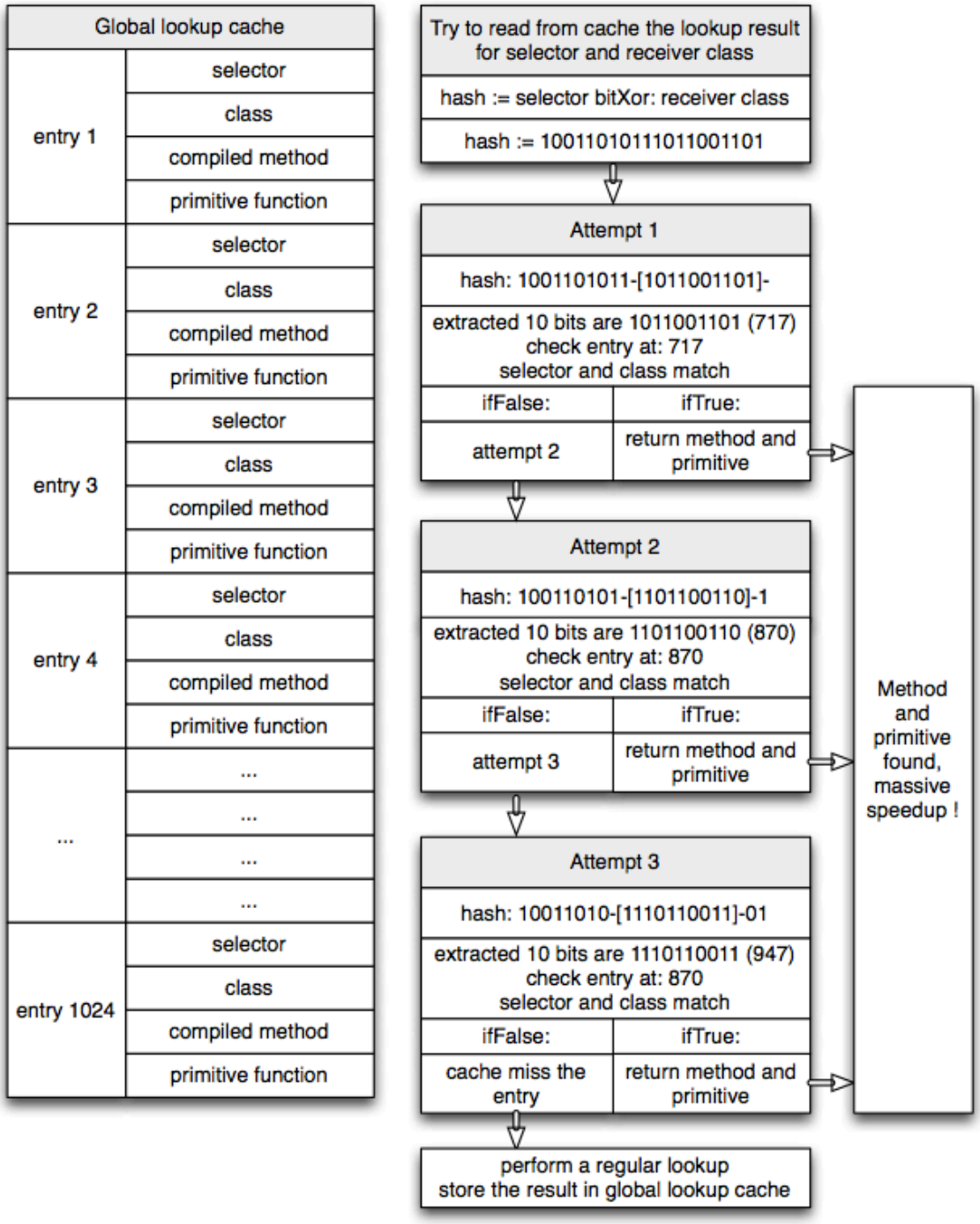
Now that we understood the regular lookup, the first optimization is to have a global cache. As we saw, the lookup currently fetches the compiled method and the primitive index of the method out of the receiver's class and the selector.

An entry in the cache will then take 4 slots:

- 1. selector
- 2. receiver class
- 3. compiled method
- 4. primitive function

The cache has currently in Cog 1024 entries. Why 1024 entries ? Firstly, because this size permits to avoid 97% of lookup, secondly, because it had to be a multiple of 2 for hash mapping.

To read an entry into the global lookup cache, you try 3 times. Basically you create a hash from the selector and the receiver's class. Then, you extract 10 bits from it at a fixed position (10 bits corresponding to a hash of 1024 entries). You check the entry corresponding to this bits (does the selector and receiver's class match). If it matches, you return the corresponding compiled method and primitive function. It it doesn't match,then you retry twice more by extracting another 10 bit portion of the selector and receiver's class hash. After 3 times, if still failing, you fall back on the regular lookup routine and save the new entry in the cache (see figure below).



To add an entry into the global lookup cache, with the same principle of reading it, you try 3 times with the 3 different 10 bits extracted from the hash until you find an empty entry to fill with your result. If no empty entry is found, you then delete the three uncorrect entries and you write your result at the first attempt location.

Note: The problem is that you need to manage the cache in case of a moving garbage collection. In Cog, in this case, the global lookup cache is flushed. You can see the comment: “WARNING: Since the hash computation is based on the object addresses of the class and selector, we must rehash or flush when compacting storage. We’ve chosen to flush, since that also saves the trouble of updating the addresses of the objects in the cache.” in StackInterpreter>>lookupInMethodCacheSel:class:

Inline caches

Then the next optimization is to speed up the lookup at each send site. This is very well explained in Cog's blog and in Urs Hölzle Phd

A send site is when the code send a message, for example, `anObject foo`. Statistically, when you write this code, in 90% of cases at runtime `anObject` may have always the same class. These send sites are called monomorphic. In 9% of cases, `anObject` may have a few different classes possible. These send sites are called polymorphic. And lastly, in 1% of cases, anObject may have a lot of different possible class (megamorphic send site).

The inline cache optimizations consists in optimizing these cases. Basically, after several iterations, all the different classes possible for the receiver of a send site will have been executed. Each time a lookup is executed, the VM saves at the send site the method found.

```
anObject foo
```

Default send site code (for first executions):

```
method := interpreter lookupSelector: selector inClass: anObject class.
jitCompiler rewriteMonomorphicInlineCache: method.
method activate
```

Case of a monomorphic inline cache, the receiver has always the class `Class1`. The send site will have been rewritten to:

```
anObject class = Class1
    ifTrue: [ method := methodPreviouslyFoundForClass1AndFoo ]
    ifFalse: [ jitCompiler extendSiteToPolymorphicInlineCache ]
method activate
```

Case of a polymorphic inline cache, the receiver has always the class `Class1`, `Class2`, `Class3`. The send site will have been rewritten to:

```
anObject class caseOf:
    Class1 -> [ method := methodPreviouslyFoundForClass1AndFoo ]
    Class2 -> [ method := methodPreviouslyFoundForClass2AndFoo ]
    Class3 -> [ method := methodPreviouslyFoundForClass3AndFoo ]
otherwise: [
    numberOfCaseInTheSwitch < 7
        ifTrue: [ jitCompiler addLookupResult: anObject class ]
        ifFalse: [ jitCompiler extendSiteToMegamorphicInlineCache ]
method activate
```

Currently a PIC in Cog can have up to 6 entries, in Visual Work they can have up to 8 (reportedly), in Self VM they can have up to 10.

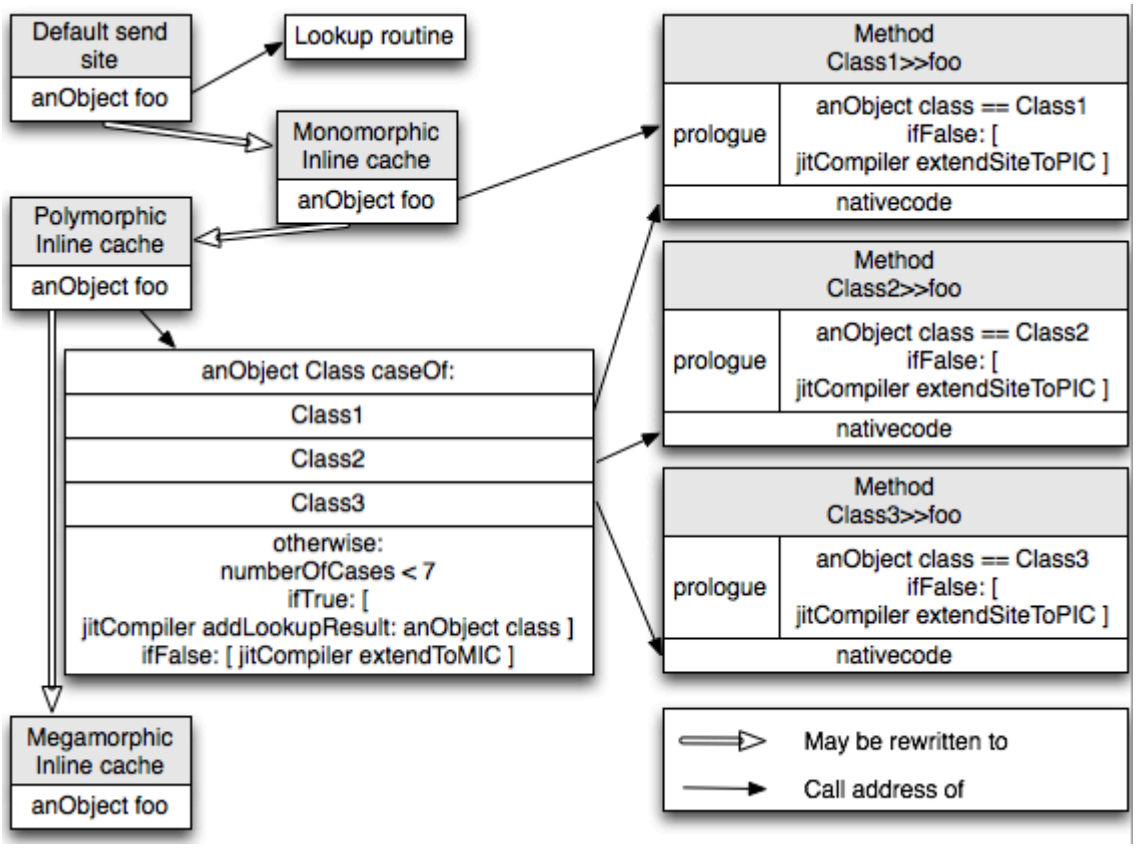
Case of a megamorphic inline cache, the receiver has always the class `Class1`, `Class2`, …, `ClassN`. The send site will have been rewritten to:

```
method:= sendSiteCache at: anObject class ifAbsent: [ jitCompiler
addLookupResultForCase: anObjectClass ].
method activate
```

Of course you cannot use a Dictionary at low level, so you have instead a hashmap similar to the one of the global lookup cache for the `sendSiteCache` but this map does not store the selector, only receiver class, method found and primitive function because the selector is fixed at a send site (here it is always foo).

Now we need to think how to implement that on a low level (Assembly code level). There is a size problem, by default you have at your send site 2 instructions that you can rewrite (`method := interpreterlookupSelector: selector inClass: anObject class. jitCompiler rewriteMonomorphicInlineCache: method.`). So basically you can rewrite 2 instructions in the send site, not a full switch case.

The idea is that monomorphic inline cache will not call directly the method, but its method prologue, which has the receiver's class checks. Polymorphic and megamorphic inline caches are compiled in a different place in the memory and the send site call this new place in memory to execute them. Obviously, creating a polymorphic inline cache always requires to allocate enough space for 6 cases even if it is useless, and something similar is needed for megamorphic send site.



You can check the code for Cog in VMMaker, or check Cog's blog for more infos. You can find interesting method in:
CogIA32Compiler (protocol inline caching)
Cogit (protocol in-line caching)
Note: In Cog, for convenience
monomorphic inline cache are named inline cache
polymorphic inline cache are named closedPIC
megamorphic inline cache are named openPIC

Note: it may look like that polymorphic inline caches are not very useful. In fact, they improve the speed of the system by around 10% only, compared to a system with only monomorphic inline cache and megamorphic inline cache. But the idea is that the polymorphic inline cache enables other optimizations, such as the one described in the next section.

The next steps

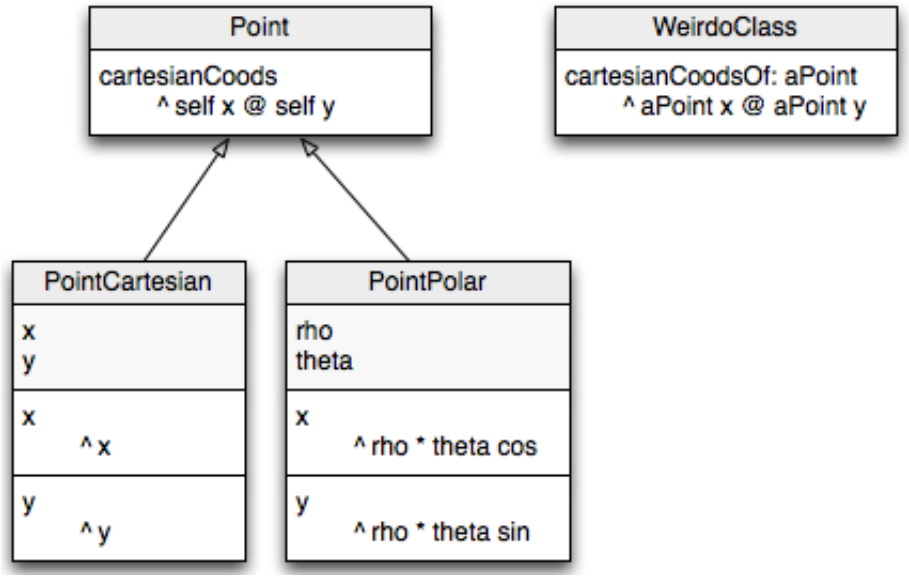
Cog is an efficient virtual machine, but it could be even better. The main developer of Cog VM, Eliot Miranda, is definitely able to make it better. So it is not a problem of skills and knowledge. The problem is to find people willing to pay you to improve the VM …

I will describe in this part some optimizations that could be done in Cog in the future.

All these optimizations are related to adaptive recompilation, which means that the JIT compiler recompile at runtime the method into something more efficient several times, depending on how often the method is called. To detect how often the method is called, you need to add some counters in the inline caches (in the polymorphic inline cache or in the method prologue), that may explodes, triggering a recompilation of the method into a code more performant.

Some optimizations take a lot of time to be compiled, so the JIT will do them if the method is called at least 1000 times. Other optimizations are cheap, letting the JIT doing them when the method is called around 10 times. These methods, called very often, are called hot spots. On a regular system, 10% of the methods are called very often (being hot spots), 90% are rarely called. Rarely called methods run unoptimized, but there are with very few overhead in the whole system speed because they are rarely called.

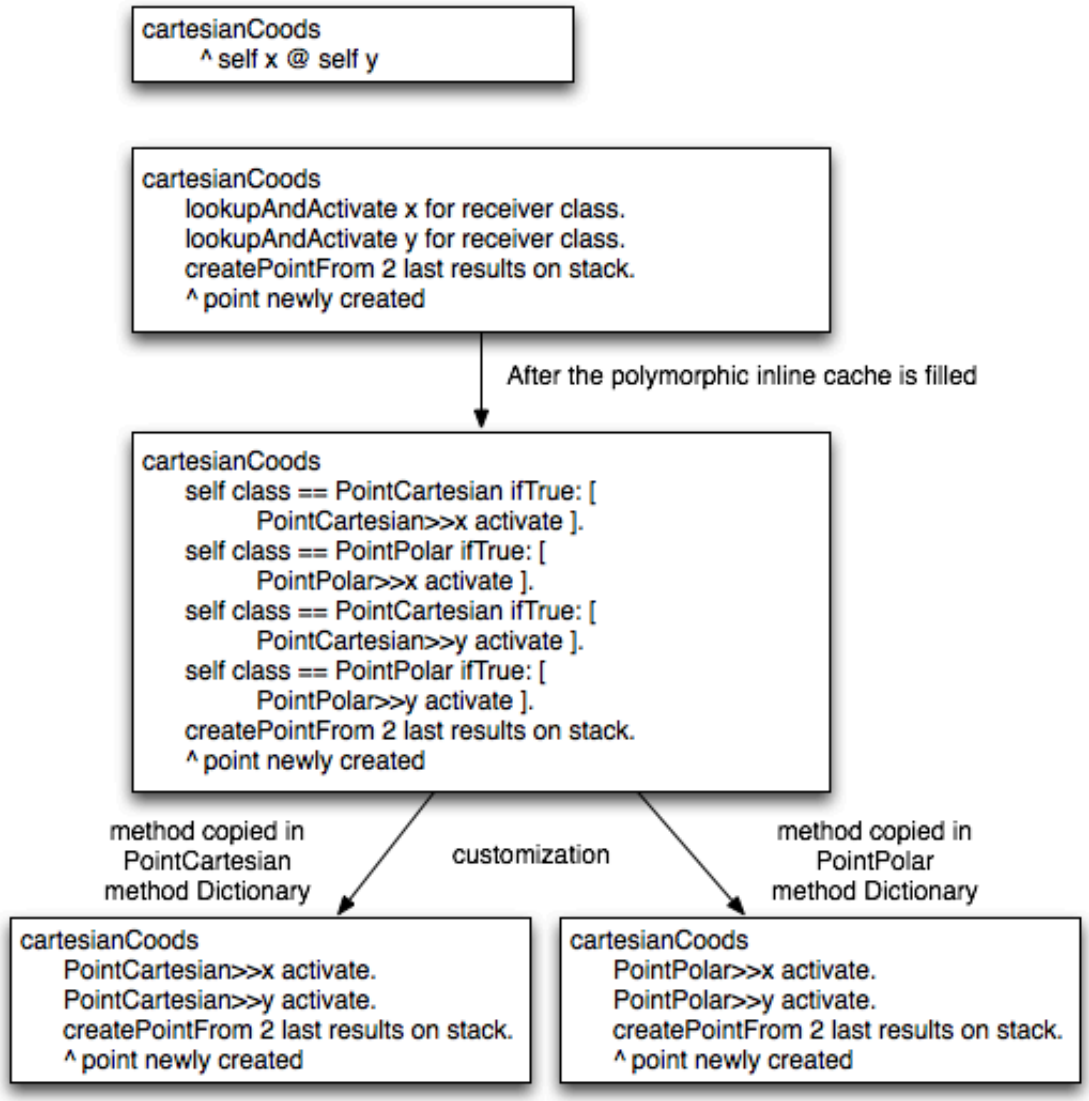
For the optimization examples, I will always refers to these classes:



Method customization

This optimization targets polymorphic inline caches on the receiver. The idea is to copy the method in its subclasses method dictionaries, to transform the polymorphic inline caches into inline caches. An inline cache is faster and is more likely to benefit from the aggressive inlining (see below) optimization.

Here is an example with the different steps of the customization of the method `Point>>cartesianCoods`. I removed the case where the virtual machine handles the uncommon case where the object is neither a PointCartesian / a PointPolar to simplify the example.



Aggressive inlining

Lastly the idea here it to avoid method activations by copying the code of the called method in the calling method. Obviously it permits to avoid the method activation and all its related costs (but this is usually already removed by the cpu), but most importantly it permits to improve all the other optimizations (such as value numbering, constant propagation, dead code elimination …), by increasing the number of cases where they can apply.

The main drawback of aggressive inlining is that it decreases a lot the debugging capabilities of the Smalltalk system. However, the Self team had found a solution for this problem, with on the fly deoptimization, letting the programmer full debugging power and massive speed improvement.

Having a JIT that does aggressive inlining and nice debugging capabilities (I mean being able to create new methods, edit methods or execute code during debugging, I do not mean crappy debugging like it is available in java systems where you cannot create a method without recompiling and restarting the current execution flow) is possible (done in Self and Strongtalk) but very hard.

Currently virtual machines in production with aggressive inlining strategies are for example Java hotspot, Javascript V8 or C# CLR. Python's Pypy has a different strategy, using metatracing to create trace of inlined methods, which is seemingly faster to implement but slower at runtime.

Message splitting

There is 2 ideas here:

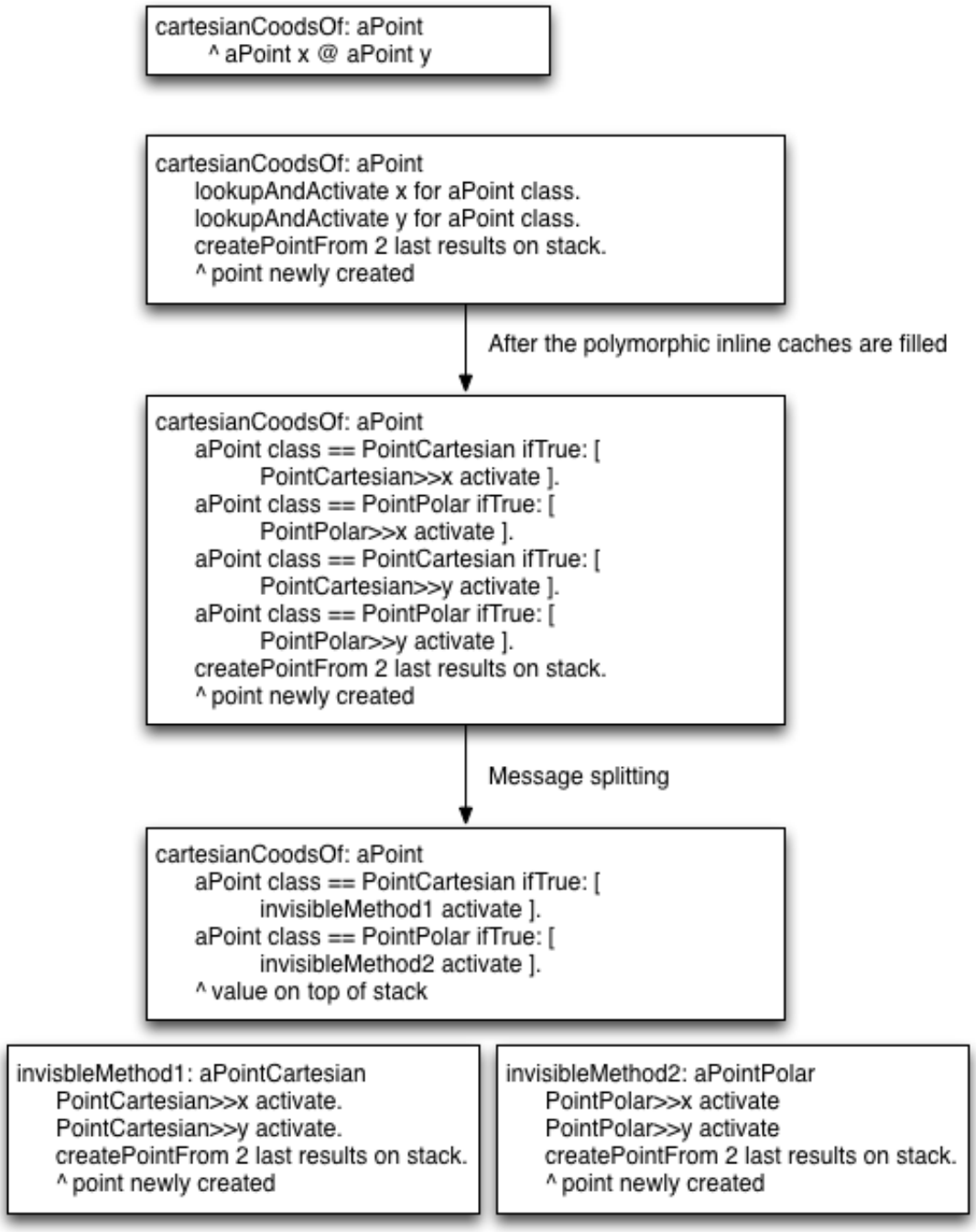
- Merging conditions
- Inlining even if the method is very long

Merging conditions is a basic idea, it applies if several conditions are executed on the same side-effect free expression. For example, if in your method you have twice a condition on `myObject isBehavior`, you can merge the different branches to have only one condition. But wait, in object oriented programming we never write conditions ! This optimization is not good ! In fact it is good, because polymorphic inline cache are switch cases, so they benefit from this optimization.

The second idea is to improve the inlining capabilities of the system. To reach maximum speed, method's native code should not be too long (for example, in Self they limited the size of the native code of a method to 2000 instructions). This is because the instruction cache of the cpu is not as good for bigger method. However, you want to inline as many methods as possible to have a fast system, without taking care of this limitation. Therefore, when your method reaches its maximum size, you can look for multiple polymorphic inline caches on the same receiver, and merge them. However, instead of just merging the code in the method, you're going to write the code for each case in a separate method that you will call (these new methods will only be referenced by inline caches, and will not be in the method dictionary). This will permit each newly created method to be optimized in a better way (having now inline cache instead of PICs on each selector), and will permit the main method to be smaller, allowing even more inlining.

Here is an example with the different steps of the message splitting of the method `WeirdoClass>>cartesianCoodsOf:`. Again, I removed the case where the virtual machine handles the uncommon case where the object is neither a PointCartesian / a PointPolar to simplify the example. Note that:

- in the invisibleMethods, you have only monomorphic inline caches (instead of the PICs we had previously).
- the method `WeirdoClass>>cartesianCoodsOf:` is now smaller, so it could allow more inlining (this applies if the method had already been inlined in other methods …)



To sum up, message splitting permits to:

- merge conditions on the same expression, removing jump overhead (even if this is not relevant in most case because the cpu will anticipates these branches for you)
- reduce the size of the method, allowing more methods to be inlined
- allow more optimizations on PICs, dispatching their calls into new methods where PICs will become monomorphic inline cache (monomorphic inline caches are better optimized than PICs)

Hope you guys enjoyed this post🙂

thoughts on “The Cog VM lookup”

1. Pingback: Cog | L'Endormitoire
2. Pingback: 5000 views ! Thank you ! | Clément Béra
3. Pingback: Squeak / Pharo VM documentation links | Clément Béra

3