Hazelnut: dynamically create a kernel in a reflexive language

Benjamin Van Ryseghem

20th May 2011

Contents

Chapter 1

Hazelnut

Hazelnut is one of the **Seed** project, where **Seed** was originally composed by different projects whose goal is to generate new kernels, all based on the project *Micro-Squeak* (Squeak is another implementation of Smalltalk) For now we essentially distinguish two of them.

- PineKernel: it is a port of Micro Squeak in Pharo
- Hazelnut: it is the building of a kernel in Pharo starting from the Pharo kernel.

This project is composed by three different parts, the kernel collection, the kernel isolation and finally the image creation.

1.1 Kernel Class Identification

Goal: The goal of this part is to create an alternative SystemDictionary (a SystemDictionary is a namespace holding all the classes of the system) starting from the Pharo one and to collect classes which are needed to build the kernel.

Problems:

- Which classes need to be collected ?
- How do we fill up the new SystemDictionary with those classes ?

Solutions:

- A first naive approach is to collect every classes Object depends on in order to have an autonomous system. But due to bad dependencies in the system, the kernel collected this way contains half of the Pharo classes. This is clearly not working. Therefore we have decided to have another approach. The second and final approach is to provide to the builder the

list of classes the user wants in the new kernel plus some classes absolutely needed by the system[1]. The problem is that we had to determine which classes are the absolutely needed ones. In order to answer this question, a tool to analyze classes dependencies has been written and recursively used starting from the Kernel package until we had a quite autonomous kernel composed of around 200 classes[2]. This tool also flags bad dependencies, but this part will be exposed in the next chapter (page 9).

- MicroSqueak's solution to fill up the new SystemDictionary is to recompile needed classes with a prefix and then to collect them. It's quite efficient when you have 20 classes to copy, but here we have the constraints that we do not know by advance what we will copy and then we want to be as fast as possible.

The solution we adopted is to create a new instance of SystemDictionary and to directly copy classes into it without recompiling them. The classes are still pointing to their original namespace as shown by Figure 1.1.

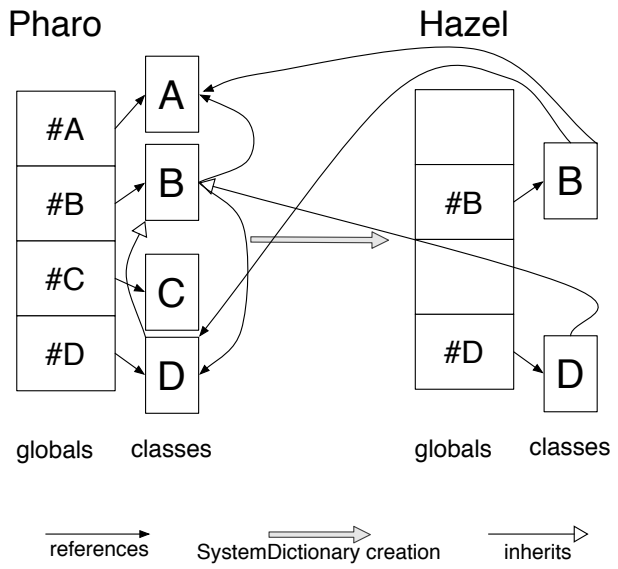

Figure 1.1: Step 1 - Copy the classes B and D into the new SystemDictionary

The second step is to make sure that the class and metaclass hierarchy is maintained in both the environments and that the methodDictionary[3] is also copied. To be sure to reconstruct the hierarchy, the copy method recursively rebuild class, metaclass and superclass hierarchy (see Figure 1.2).

[1] as Object, ProtoObject or MethodContext for example

[2] Pharo contains around 1800 classes

[3] a *methodDictionary* is a dictionary implemented in each class and containing all the methods of the class

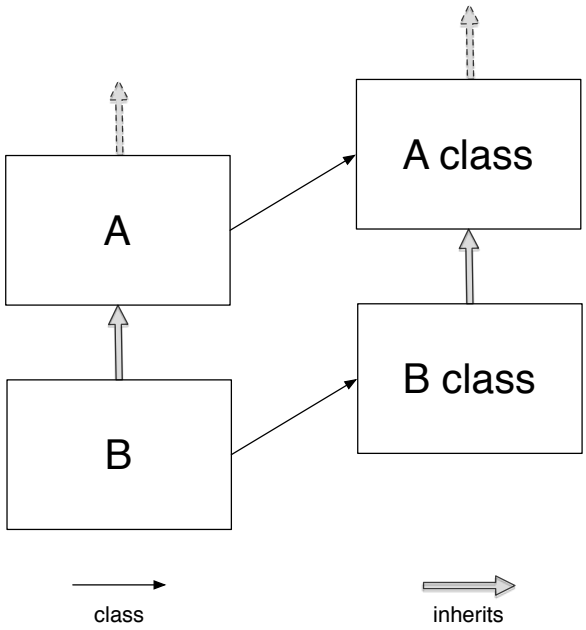

Figure 1.2: Class and MetaClass Hierarchy

Here is the pseudo code in Smalltalk that add a class in the Hazel SystemDictionary and check the hierarchy:

```
HazelKernelBuilder>>#addAClassInDictionary: class
"Add a copy of the class in the Hazel SystemDictionary then answer the copy"

    | hazel copy className |
     className := class name asSymbol.

    "Check if the class is already in the dictionary"
    (self list includesKey: class name)
          ifFalse: [^ nil].

     hazel := Smalltalk at: #HazelSmalltalk.
    (hazel globals includesKey: className)
          ifTrue: [^hazel at: className].

    "If not, add a copy in the dictionary"
     copy := self copyClass: class.
     self registerClass: copy.

    "then check the superclass"
```

```
copy superclass ifNotNilDo: [:superclass || superCopy |
    "add the superclass"
    superCopy := self addAClassInDictionary: superclass.
    "change the superclass"
    copy superclass: superCopy.
    "then change the metaclass's superclass"
    copy class superclass: (superCopy class)].

"Check all literals of all methods"
self checkMethods: copy.

"Check all class var"
self checkClassVar: copy.

^ copy
```

The last instructions will be commented in the next section.

The only wrong inheritance which remains is that ProtoObject in the Hazel world inherits from nil which is still in the Pharo world. But this will be fixed when we will change nil (see paragraph 1.3 page 10).

In a nutshell: We are now able to copy wanted classes and needed classes into a new SystemDictionary, with a good hierarchy, but Hazel classes keep references to Pharo ones (see Figure 1.1).

1.2 Kernel Isolation

Now that the kernel is created, we need to isolate it by removing dependencies to the Pharo world. There is two different types of dependencies which need to be fixed.

1.2.1 References to unwanted classes

Goal: Here we want to remove dependencies from Hazel classes to Pharo classes we haven't copied.

Problems:

- How to detect unwanted references ?
- How to remove those dependencies ?
- How to be sure it will not crash the system ?

Solutions:

- There are two places where unwanted references can be found:
  - In a method: in a method literal there are references to invoke classes (see page **??**). Due to that, we can found references to unwanted classes;
  - In a class variable[4]: we can have an instance of an unwanted class or just an unwanted class itself.

  The solution adopted is to check its methodDictionary and class variables during class copy. If unwanted references are found, the following solution is applied.
- This is the key point of this cleaning step.

  For class variables, the solution was to set them to nil (in the minimal kernel creation process, only one class variable had to be set this way (HaloFont from StandardFonts).

  For methods we have considered several solutions. Our first thought was to remove the method and to recursively remove the sender of the method. But due to class structure, we removed that way almost all methods of the kernel. Then we have though to create a *NullPattern* object implementing all the methods removed. The problem was to find which kind of answer is expected from each method, and how to dynamically replace the sender in the code source. Finally we chose to remove the method and to keep the senders.

[4] a class variable is a variable shared by all the instances of a class

- We haven't found a solution to this question. As long as your system can be changed, you can't certify that your kernel is totally functional. We are working on a better isolation of the Pharo kernel to reduce as much as possible the number of references (see Section **??** page **??**).

In a nutshell: We have removed the references to unwanted class (see Figure 1.3), but because of that the integrity of the kernel may be corrupted.

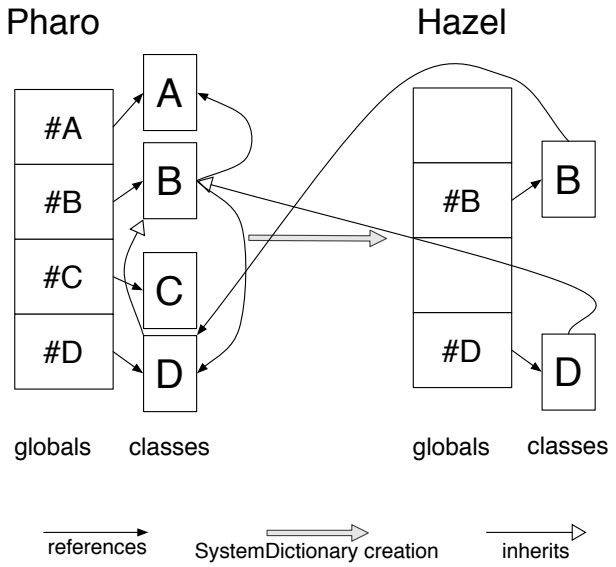

Figure 1.3: Step 2 - Remove references to unwanted classes

1.2.2 Reroute dependencies from original classes to Hazel classes

Goal: Here we want to reroute dependencies from Pharo classes to Hazel classes to have only intern references in the Hazel kernel. Those references are stored into methods literals.

Problems:

- How to change those references ?

Solutions:

- This part is quite simple because the field was well prepared. Only methods refering copied classes are not fixed yet. So for those classes, the methodDictionary had just to be parsed in order to fix literals. And for fixing literals, we just change the Pharo associations to their corresponding Hazel one (and we are sure it exists).

In a nutshell: We now have a kernel isolated with only internal references (see Figure 1.4).

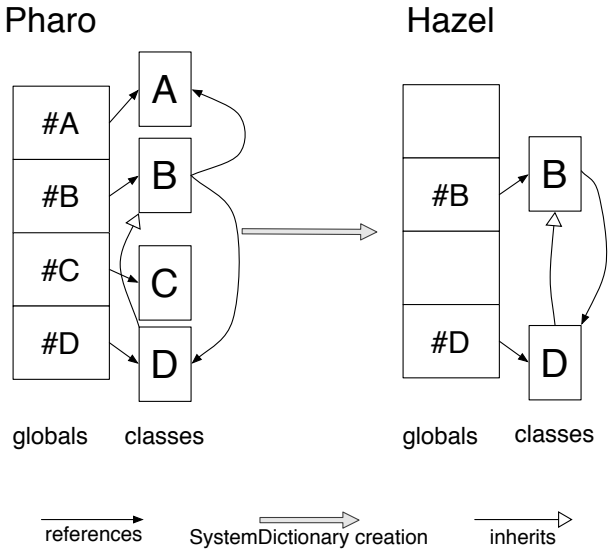

Figure 1.4: Step 3 - Reroute the remaining dependences

1.3 Image Creation

Goal: The goal of this part is starting from an isolated kernel succeed to build a new image using this kernel. An image is a snapshot of living objects binary saved in a file. They basically contains classes and some living instances.

Problems:

- Which technique should we use to create the image ?
- Is it necessary to have a specific Virtual Machine to build image ?
- Is it necessary to have a specific Virtual Machine to read the image ?
- How to successfully replace the Special Objects Array ?

Solutions:

- Two solutions have been tested to create the image:
  - The Micro Squeak solution which consists in the collection then the serialization of all the needed objects into a new image. This technique works for Micro Squeak thanks to the limited amount of classes and of objects. Moreover two passes are done on objects with different algorithms and due to that difference, we had some missing objects. In a first time, we tried to fix or rewrite methods, but we finally decided to consider another solution, dynamically switch the Special Objects Array (see Figure 1.5);
  - The second solution is to take a lively image and to dynamically switch the Special Objects Array in order to make the unneeded object garbage collected (see *Garbage Collector* page **??**). The difficulty of this method is that we are drastically modifying the image during its own execution. Some objects can easily be changed (as nil or Character) when some others freeze the Virtual Machine (as String or Semaphore). We think it's due to the fact that during the execution of the switching method, we are modifying the method context, and the Virtual Machine points to unaccessible pointers and we got an error[5] (see Figure 1.6).
- In order to avoid the previous problem, we have started to implement new primitives in C for the Virtual Machine, which force users to use a specific Virtual Machine for image creation.
- The new primitives are only used to switch objects (basically change pointers to those objects), they should not be needed to run the image. That way, users of the new image should not have to use a specific Virtual Machine.

[5] segmentation fault

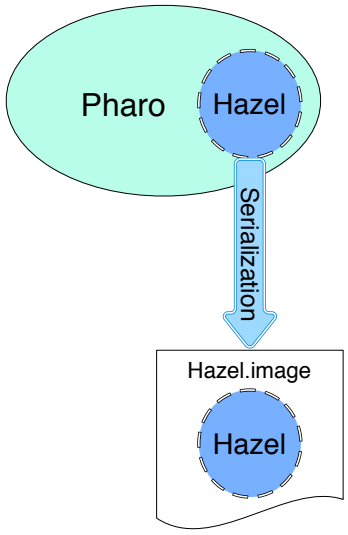

Figure 1.5: Micro Squeak - Serialization of needed objects

- To replace the Special Objects Array, the first approach was to take the current Special Objects Array, which is a Dictionary, and to replace its values. The problem is that some values called every time by the image are buffered in the Virtual Machine (those values are nil,true and false) and updated at the opening of the image. So we have decided to use the primitives become: and becomeForward: which basically switch references between the receiver and the argument.

In a nutshell: Due to some objects that can't easily be switched, now we are not able to generate a new image starting from our Hazel kernel.

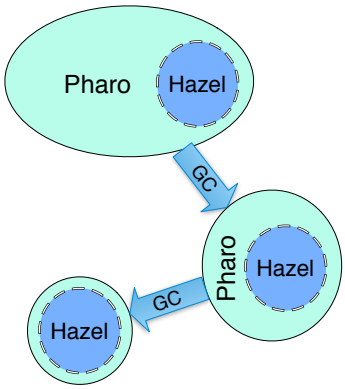

Figure 1.6: Hazel - Garbage Collection of unneeded objects