OOPSLA'91 Workshop on Reflection and
Meta-Level Architectures in Object-Oriented Programming

Position Paper

Reflection in Prototype-Based Object-Oriented Programming Languages

J. Malenfant, P. Cointe and C. Dony
Équipe mixte Rank-Xerox France - LITP
Université Pierre et Marie Curie
Tour 45-55, bureau 203
4 Place Jussieu
75252 Paris CEDEX 05, FRANCE
e-mail: {malenfan,pc,chd}@rxf.ibp.fr

Abstract

In this position paper, we investigate structural reflection in prototype-based languages. The language Self is used as experimental subject because of its complete implementation and its available documentation. In Self, two entities deal with structural properties of objects: maps and mirrors. Maps factorize common information about the format of similar objects; they play the structural role of classes yet they are not Self objects. Mirrors, on the other hand, are Self objects but they don't store information on their own: they give a reading access to internal information about objects through a set of virtual machine primitives. We look at the properties of maps and mirrors and actually, the interesting points are the following. First, maps have the advantage over classes to be managed automatically by the system; reifying them into the language could lead to the study of the automatic management of shared structural meta-objects which may be viewed as a high-level storage management. Second, the use of mirrors raises the question of whether systems that only offer causally connected reading and writing accesses to their internal data structures through system primitives can be considered as reflective or not.

Résumé

Dans cet article, nous nous penchons sur la réflexion de structure dans les langages à prototypes. Le langage Self est utilisé comme terain d'expérienmentation en raison de son implantation complète et de la disponibilité de sa documentation. En Self, deux entités contiennent des informations sur les propriétés structurelles des objets: les descripteurs et les miroirs. Les descripteurs factorisent les informations communes sur le format des objets de même structure; ils jouent le rôle structurel des classes sans être des objets Self. Les miroirs, par contre, sont des objets Self mais ils ne contiennent aucune information en propre; ils donnent un accès en lecture à des informations internes sur les objets à l'aide d'un ensemble de procédures système. L'examen des propriétés des descripteurs et des miroirs, a mis en évidence les points intéressants sont les suivants. Premièrement, les descripteurs ont l'avantage par rapport aux classes d'être gérés automatiquement par le système; les réifier dans le langage mènerait à l'étude de la gestion automatique de méta-objets de structure partagés, ce qui pourrait être considéré comme une forme de gestion de mémoire de haut niveau. Deuxièmement, l'utilisation des miroirs soulève la question de savoir si les systèmes qui offrent uniquement des accès en lecture et en écriture à leurs structures de données internes, à l'aide de procédures système, tout en respectant la connexion causale, peuvent être considérés comme réflexifs ou non.

1 Introduction

Today, many object-oriented systems exhibit some form of reflective capabilities. In class-based languages for example, classes and metaclasses implement what has been called *structural reflection*. The corollary *behavioral reflection* is concerned with the execution of objects: message passing as well as method lookup and invocation. 3-KRS [Mae87b, Mae87a] has developed these ideas using the concept of *meta-object*. In the 3-KRS framework, every object has a one-to-one relation to a meta-object which represents the explicit information about its referent (e.g about its behavior and its structure). Meta-objects are themselves objects thus reifying this information into language constructs. Watanabe and Yonezawa [WY88] have then studied the use of meta-objects in the concurrent reflective language ABCL/R while Ferber [Fer89] has suggested an integration of meta-objects into a class-based model, namely ObjVLisp [Coi87].

Despite the intrinsic interest of these attempts to capture the essence of reflection, we actually feel that the extra burden of concurrency as well as the confusion arising between classes, metaclasses and meta-objects, as in Ferber's proposal, makes it a lot harder to reach the fundamental issues of reflection. Hence, we suggest to study reflection in less elaborated OO models. The underlying conjecture asserts that the study of crucial issues like the causal connection between the object level and the meta-level should be easier in such models.

We try to "simplify" the traditional OOP model by restricting ourselves to sequential objects without classes. At first sight, prototype-based models [Lie86, US87] seem to obey to these "simplifying" assumptions thus we have chosen them as the basis of our study. However, the apparent simplicity of prototype-based models is deceptive. Prototype-based systems proposed since a few years, such as Lieberman's prototypes [Lie86], Self [US87], Stein's Hybrid language [Ste87] and Lalonde's examplars [LaL89], tend to have fuzzy semantics. A correlated goal of our research is an attempt to give precise semantics of prototype-based languages using reflection, eventually leading to a *Meta-Object Protocol* [**?**] for prototypes.

Actually, Self is certainly the most documented and the most thoroughly implemented prototype-based language. As reflection asks for the reification of entities realizing the implementation, it seems reasonable to begin our study by looking at Self and at its implementation. Reflection in Self is also appealing since it is in the tradition of Lisp and Smalltalk, two languages in which reflection has been studied extensively [**?**]. The rest of this paper is organized as follows. The second section introduces Self from a programmer point of view. The third section succintly presents the Self implementation. The fourth section discusses in a structural reflection point of view two entities that form the cornerstone of the Self implementation and kernel facilities: maps and mirrors. We then conclude on some issues raised by the study of maps and mirrors.

2 Self: a programmer point of view

In this section, we examine the Self model of computation as well as the programming methodology proposed by the Self group in several papers [CUCH91, UCCH91, HCCU90].

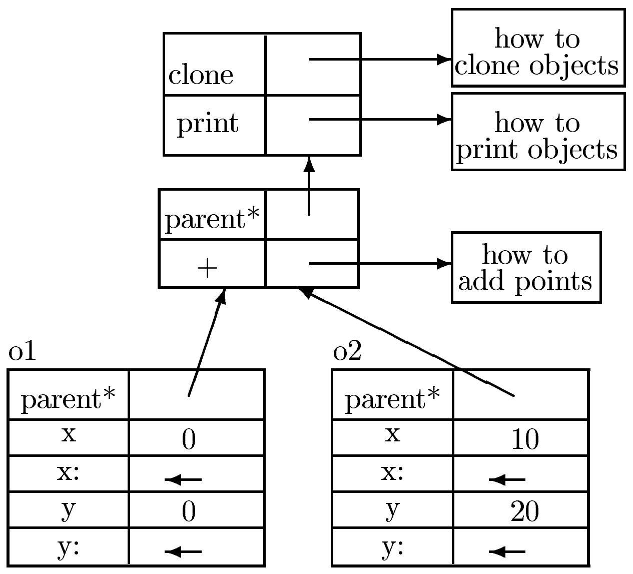

Figure 1: Cartesian points in Self [US87].

2.1 Basics

The Self model of computation is based on standalone objects using *delegation* [Lie86] to share properties. A Self object is a collection of slots, keyed by slot names. Self's design omit classes but also "instance" variables which are implemented as "methods" returning a constant value, accessible through message passing [US87]. Each object can serve as prototype for the cloning operation which is the basic operation to create new objects. Arbitrary Self objects can inherit from one another through parent slots. Message passing is the fundamental computation mechanism in Self. Self also owes to Smalltalk [GR83] much of its basic syntax as well as its treatment of control structures using blocks and message passing.

As an example of Self programming, the Figure 1 shows two cartesian point objects which inherit the `'+'` method from a common parent which, in turn, inherits the `print` and `clone` methods from its own parent. Basically, sending a message to a Self object involves first a lookup to find a slot corresponding to the message selector and, second, the application of the slot contents in the scope of the receiving object. The lookup first searches a matching slot in the receiving object and if it fails to find one, it explores the parent hierarchy [CUCH91]. For example, assume that we send a `print` message to the object `o1` in Figure 1. The lookup first searches a `print` slot in `o1` and fails to find one; it then searches in the parent of `o1`, fails again and finally finds one in the parent of the parent. The contents of the `print` slot is a method object which is applied in the context of `o1` to retrieve the appropriate slots and slot values to be printed out. More programming examples can be found in [US87, CUCH91, UCCH91].

2.2 Traits: a programming methodology

Prototype-based languages emphasize flexibility against rigid organizations. Lieberman [Lie86] pointed out that prototypes allow creating individuals prior abstractions while class-based systems force the definition of the abstractions (classes) before individual instances. However, the lack in organizational constraints can lead to unreadable programs which, in turn, can drastically reduces reusability (even though prototype model are designed to make sharing easier than in the class-instance model [Lie86]). To fill the organizational gap, Self proposes a programming discipline based on *traits objects* [UCCH91]:

> *... data types may be defined in a classless language by dividing the definition of the type into two objects: the prototypical instance of the type and the shared traits object. The prototype defines the instance-specific aspects of the type, such as the representation of the type, while the traits object defines common aspects of all instances of the type. No special language features need to be added to support traits objects and thus user-defined data types—a traits object is a regular object shared by all instances of the type using normal object inheritance. Since traits objects are regular objects, they may contain assignable data slots which are then shared by all instances of the data type, providing the equivalent of class variables [UCCH91].*

Hence, traits objects are shared repository for common behaviors and state. Let's illustrate this traits-based design methodology using the cartesian point example. The traits-based methodology developes this example in three steps:

1. The first step is to create the cartesian point traits with the shared behavior for cartesian points (here a slot named `'+'` which associates the selector `'+'` with the method code to add cartesian points). The cartesian point traits can inherit less specific shared traits, like here a general traits which holds the slots `print` and `clone` which associates the selectors `print` and `clone` to the method code that respectively prints and clone any object.
2. The second step creates the first prototype of cartesian point which will examplify their representation, here two assignable slots named `x` and `y`. The first prototype, called prototypical instance in the Self terminology, initializes the "parent-of" link to the cartesian point traits in a slot called `parent*`[1].
3. The last step is the normal use of cartesian points. New points can be created by cloning the prototypical instance of cartesian point and by initializing the `x` and `y` slots to appropriate values (by sending `x:` and `y:` messages). For example, the point `o2` in Figure 2 is created by sending the message cascade `(clone x:10) y:20` to the prototypical instance of points `o1`.

This design methodology establishes a clear distinction between the data type design and its use, with respect to the way the objects are created. In general, when a new data type is created, the traits and the prototypical instance of the type must be created *ex nihilo*, not by cloning existing objects. An alternative mean to create these "first-of-a-kind" objects must be provided. In Self, two approaches are proposed. The first is code elaboration where a textual representation of the objects (in a file) is read and where the parser create the corresponding objects. The second approach is to build the objects slot-by-slot using the primitives `_AddSlots:` and `_DefineSlots:` [HCCU90]. After the cartesian point data type has been created (traits object and prototypical instance), the cloning operation is sufficient to create new points.

We note here that, in contrast with class-based languages, all "instances" of a type are not created in the same way: the first is constructed *ex nihilo* using more complex operations while the following ones are created by the simpler cloning procedure. On the

[1] Slot names with trailing asterisks denote parent slots used by the lookup procedure; the number of asterisks determine the slot priority (the less asterisks is the higher priority) when there is more than one parent.

other hand, a complete discussion on whether this programming discipline goes against the advocated advantages and *raison d'être* of prototypes is well beyond the scope of this position paper.

3 Self: an implementor point of view

In this section, we look at two main Self entities: maps and mirrors. The first is used in the implementation of the virtual machine while the second is used in the language kernel.

3.1 Maps

One of the major problem when implementing prototype-based systems is how to optimize the storage management when a large number of similar objects are created. For instance, in a naive implementation of Self, each object must hold the names of all its slots as well as their associated values. Hence, such a naive implementation would waste space by repeating thousands times the same information [CUL89]. To solve this problem, *maps* were introduced:

> ... *as an implementation technique to efficiently represent members of a* clone family[2]*. In our Self object storage system, objects are represented by the values of their assignable slots, if any, and a pointer to the object's map; the map is shared by all members of the same clone family. ... From the implementation point of view, maps look much like classes, and achieve the same sorts of space savings for shared data. But maps are totally transparent at the Self language level ... [CUL89].*

Maps factorize information that is constant for all objects in its clone family. This information is the slot names as well as the value of constant slots (a constant slot is a slot which has no corresponding assignment slot). The Figure 2 shows the cartesian point example of Figure 1 but represented using maps. Maps are created according to the following rules. A map is created whenever a new object is created without cloning an existing object (i.e. through code elaboration) or when an existing object is modified by one of the primitives `_AddSlots:`, `_DefineSlots:`, `_RemoveSlot:`, etc[3]. This map contains the slot names of the corresponding object. The map also relates slot names to two kinds of information: for constant slots, the map stores directly the value associated to the slot while for assignable slots, it stores an offset which indicates where to find the value of the slot in objects of its clone family. For example, in Figure 2, the map for cartesian points contains the slot names `parent*`, `x`, `x:`, `y` and `y:`. For `x` and `y`, it stores their respective offsets in the object while for `parent*`, because it is a constant slot, it directly stores the pointer to the parent of a point which is the cartersian point traits[4].

It is important to notice that *map-of* links and maps are invisible to the Self programmer: maps are created and managed automatically by Self. Therefore, the Self world is divided in two parts: an external part which is visible from the Self language and an hidden part which contains maps. Moreover, the visible part of the world works as if objects contain slot names and values even if their real implementation relies on maps.

[2]A clone family is: "a prototype and the objects cloned from it, identical in every way except for the values of their assignable slots" [CUL89].

[3]Actually, `_AddSlots:` and `_DefineSlots:` do both: they modify the structure of the receiving object and create new objects by parsing their argument.

[4]This view of maps is an external view which shows the sharing properties of maps. Maps also contains other information about the structure of objects in their clone family such as their physical length and the

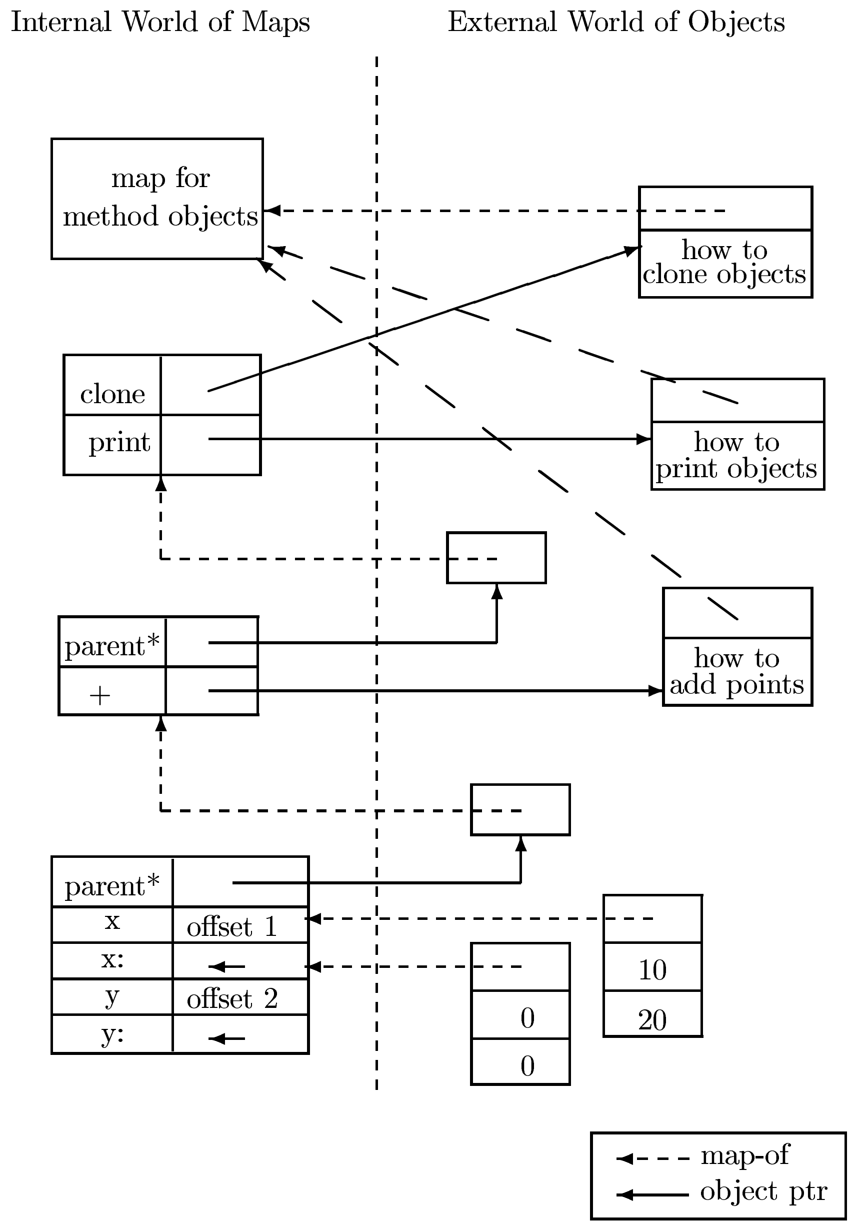

Figure 2: Self objects represented using maps [CUL89].

3.2 Mirrors

Another important entity in Self which provides an interface between the language and the implementation levels are the *mirrors*:

> *A mirror on an object* `x` *is obtained by sending the message* `reflect:x`*, which is defined in default behaviors to invoke the* `_Mirror` *primitive on* `x`*. Mirrors make their reflectees look like collections of slots, keyed by slot names, with values that are mirrors on the slot contents. A mirror permits queries to be made about the object it reflects, such as the number of slots, the name of each slot, whether a slot is a parent, the visibility of a slot, etc. Operations on mirrors also include retrieving a mirror on the contents of a slot and returning mirrors on all references to an object. Asking a mirror for the name of the reflected object gives the string returned by the name inferencer.*
>
> *There are eleven kinds of mirrors, one for each kind of object known to the virtual machine: the assignment object, blocks, byte vectors, integers, floats, strings, object vectors, methods, mirrors, and plain objects.*
>
> *Iterating through a mirror returns objects representing each slot of the reflected object. The prototypes for these objects are:* slots plain*, representing an ordinary slot;* slots parent*, representing a parent slot;* slots argument*, representing an argument slot; and* slots method*, representing a slot with code. "Fake" slots are objects representing code (for a method), reflectee (for a mirror), and vector elements (for an object vector of byte vector) [HCCU90, p. II–15].*

The Figure 3 shows the relationships between a cartesian point object `o1`, its mirror `mo1` and the contents of the slot `x` of `o1`. A mirror `mo1` on a point `o1` is created by sending `o1` the message `_Mirror` (a Self primitive). The mirror inherits from traits collection the behavior of collections. Thus, the protocol to access individual slots of the reflectee (`o1`) is by sending the mirror messages such as `at:i`, where `i` is the index of the slot to be observe in the reflectee ($0 \leq i \leq n-1$). The answer of the message `at:1` to `mo1` is a slot descriptor object `sd1` which defines the properties of the corresponding slot in `o1`. Messages such as `visibility`, `isAssignable`, `isParent`, etc. can be sent to this slot descriptor object `sd1`. When `sd1` receives a message `value`, it returns a mirror on the contents of the corresponding slot of the reflectee. For example, in the Figure 3, the message `value` to `sd1` returns a mirror on the contents of the slot named `x` of `o1`. Thus, the object `mv1` which is the result of the message `value` to `sd1` appears to be the same as the result of a message `_Mirror` sent to the value `v1` of the slot `x`. This `_Mirror` message is implicit in the Figure 3 since nothing says how `mv1` is actually created in Self[5].

Examples of mirrors utilization

Mirrors are thoroughly used in Self source code to access internal information about objects. For example, the read-eval-print loop in Self is implemented by the slots named `doIt` and `printIt` of the `lobby`[6]. First, the expression is read, parsed and stored in the slot `doIt`; then the message `printIt` is called resulting in the execution of the following code:

number of their slots [CUL89].

[5]The message `value` to `sd1` actually sends the message `contentsAt:1` to the mirror `mo1`, which is defined in traits mirror as calling the primitive `_MirrorContentsAt:1` on `mo1`.

[6]The `lobby` is an elected object in Self which gives a comprehensive context for the evaluation of Self expressions. It is used as the top-level context for the read-eval-print loop.

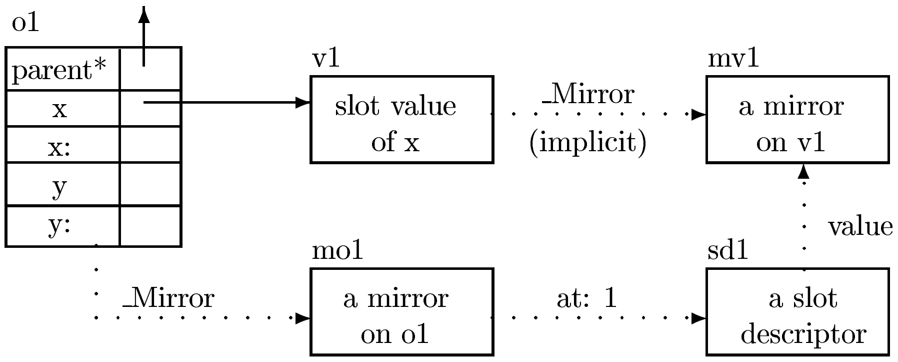

Figure 3: Relationships between mirrors and their reflectees [HCCU90].

```
printIt = ( (reflect: doIt) name printLine. self )
```

This method operates as follows. The `lobby` first sends itself the message `doIt`[7]. The result of `doIt`, which is the result of the evaluation of the top-level expression, is used as argument of the message `reflect:` returning a mirror on this result. The mirror is sent the message `name`, which results in a string representing the name of the object which would be inferenced by the name inferencer[8]. This string is printed out by the message `printLine` and `self` is returned as the result of the `printIt` method.

A second example of the use of mirrors is the method `inspect:`, used to inspect the contents of an object:

```
inspect: obj = ( inspectReflectee: ( reflect: obj). obj )
```

Again, the receiver of the message `inspect:` first get a mirror on the argument `obj` by sending itself the message `reflect:obj` and then invoke the method `inspectReflectee:` on this mirror:

```
inspectReflectee: aMirror = (
   '( |' print.
   aMirror doFirst: [| :slot | ' ' print. slot print. '.' print. ]
           MiddleLast: [| :slot | '\n    ' print. slot print. '.' print. ]
           IfEmpty: nil.
   aMirror isReflecteeMethod
           ifTrue: [ ' |' printLine.
                 (aMirror code source asParagraph asCodePad: 2) print.
                 ')' printLine. ]
           False: [ aMirror hiddenDo: [| :fake |
                        '\n      ' print. fake print. '.' print. ].
                  ' | )' printLine. ].
   aMirror )
```

`inspectReflectee:` merely prints the reflectee's slots and, if the reflectee is a method, then it also prints its source code. Finally, if the object contains "fake slots", they are

[7]the implicit receiver in the Self terminology [HCCU90].

[8] *"The name inferencer is invoked by sending the message* `name` *to a mirror on an object. If the object contains a slot called* `thisObjectPrints`*, indicating that the object knows how to print itself intelligently, the name inferencer sends* `printString` *to the object and returns the result as the name. If there is no such slot, the name inferencer consults the path cache to infer a name (see section II-1.3). If the object is not in the path cache (is not well known) the generic name* `<an object>` *is returned" [HCCU90].*

printed last.

4 Structural reflection

Structural reflection aims at giving a self-representation of structural aspects of objects which can be manipulated in a causally connected fashion within the language. In Self, two entities deal with the representation of objects: maps and mirrors. We discuss both in turn in a structural reflection point of view.

4.1 Maps

As we have seen in the previous section, maps describe the format of all objects in their clone family in a way which resembles what classes do for their instances. Maps hold the name of the slots, pointers to the value of constant slots and offsets where to find the value of assignable slots in the clones. Accordingly, objects in their clone family are like object vectors where the first entry is a pointer to the map followed by the value of their assignable slots. Maps play the structural role of classes. The main difference is that maps are created and managed automatically by the system. But, in the structural point of view, the map-of link from an object to its map is of the same nature as the class-of link in class-based languages.

Even though maps were introduced in Self for the very reason of space savings, an interesting question is whether they can serve as the basis for structural reflection in prototype-based languages. The two main interests in this are, first, to see if automatically generated structural meta-objects is a feasible approach to structural reflection and, second, to reify maps in order to modify the strategy used to generate them. Maps have the advantage over classes to be induced by the system; they don't force the programmer to create an abstraction prior individual objects. Hence, they preserve the prototype way of thinking while achieving the space savings of classes when many similar objects are created. The actual Self strategy for creating maps is efficient in time but it fails to fully optimize space in many cases. For "one-of-a-kind" objects, creating a map is a loss of space. Also, the Self system will recognize two objects as sharing the same map only if the are in the same clone family i.e. if one has been created by cloning the other or if they have both been created by cloning objects which are in the same clone family. More appropriate strategies may be implemented such as creating a map for a clone family only when it contains more than one individual. Also, a process similar to garbage collection could be invoked from time to time to reorganize the memory and to match structurally equal clone families into one clone family sharing a unique map. Implementing such strategies could be done by modifying the Self virtual machine but a more interesting approach would be to reify maps in order to implement new strategies in the language itself.

Actually, maps are not Self objects, they only exist at the implementation level. Also, objects in the base level hide their map-of link to the user (one cannot send a message to an object to get its map-of pointer). A first approach to make maps visible at the base level would be:

1. To make maps true Self objects.
2. To represent the map-of link as a parent-of link (the only link which has a predefined semantics in Self and which is used by the lookup algorithm).

Just doing this does not work. For example, sending a message `x` to a cartesian point in Figure 2 would then retrieve an offset from the map, not the value of `x`. To get the value of x, the offset must be re-interpreted as a structural meta-information showing where to find the desired value in the context of the receiver. A simple alternative is to replace the offsets stored in the map by a method which can retrieve the appropriate value in the base-level object. This means that instead of storing a declarative information about the structure of the object[9], it would store a procedural information which would freeze the operational meta-level semantics. In other terms, the offset would become an explicit accessor function melding the meta-level interpretation and and the base level offset information.

We draw two preliminary conclusion here. First, it is not so easy to unify the class-of relationship (represented here by the map-of link) and the inheritance relationship (represented by parent-of links) despite claims made in many papers [US87, for example]. There is a meta-interpretation of the information which differs in a structural reflection point of view and which collapses here. The second preliminary conclusion follows immediately: if you want to represent and manipulate some kind of structural meta-object in a language which only supports an inheritance relationship, you will need to invent a new relationship (close to an instance-of relationship), and the semantics of the language must be changed to force the lookup and apply procedures to interpret correctly this new relationship. A reason why maps are invisible in Self is probably the fact that it would have infringed many of the language's basic assumptions, such as the presence of only one explicit sharing link between objects. Naturally, in an hypothetical behaviorally reflective Self, these changes in the operational semantics could be implemented — and eventually modified — in the language itself.

4.2 Mirrors

The mirror is the second entity which explicitely deals with the representation of objects. In contrast with maps, mirrors are Self objects directly accessible from the language. This property is important to achieve true structural reflection. But mirrors are quite elusive entities. At first sight, they seem to be entities that hold much information about there "reflectee", as we would expect from a self-representation of objects. This is not the case in the actual Self implementation. In fact, mirrors contain no visible information on their own. A careful examination of the Self source code defining mirrors shows that the only visible information in a typical mirror are a parent slot which points[10] to the traits mirror object (which defines the common bahavior for mirrors), and a slot `thisObjectPrints` set to true and used by the Self name inferencer. Even the pointer to their reflectee is hidden in the implementation of the mirror, as it is shown in the following cloning experiment on a mirror:

```
> _AddSlots:(| mirrorPoint = point _Mirror |)
> mirrorPoint _Print
mirror <reflectee = <2>> <13>: ( | ^ parent* = <14>.
                                   _ thisObjectPrints = true. | )
> mirrorPoint clone _Print
mirror <reflectee = <2>> <13>: ( | ^ parent* = <14>.
                                   _ thisObjectPrints = true. | )
> mirrorPoint _Clone _Print
```

[9]here an explicit offset and an implicit assumption that base-level objects are represented as object vectors.

[10]This is actually a little oversimplified since different kinds of mirror may have specific behaviors, but yet they will inherit the basic behavior of mirrors accross a more specific traits object [HCCU90].

```
mirror <reflectee = -33555526> <16>: ( | ^ parent* = <14>.
                                         _ thisObjectPrints = true. | )
```

In this simple session, we create a mirror on the prototypical instance of cartesian points and add it to the lobby in a slot called `mirrorPoint`. We print `mirrorPoint` and see that the reflectee's identifier is `<2>` (an object reference in Self is printed as an integer surrounded by angle brackets, [HCCU90]). We then send `mirrorPoint` the message `clone` and print the resulting clone. The clone is exactly the same object as `mirrorPoint`, simply because the clone method is redefined in traits mirror to return the receiver. Next we send mirrorPoint the primitive `_Clone` which cannot be redefined, and print the result. We the see that the reflectee identifier is no longer valid. The reflectee slot is what Self calls a "fake slot" (see on page 7 the quotation on mirrors).

Since a mirror contains no information about its reflectee (except a "fake slot" which points back to it), what does the bulk of the job is a set of language primitives to which mirrors are able to respond: `_MirrorNameAt:`, `_MirrorContentsAt:`, `_MirrorIsParentAt:`, ... [HCCU90]. Hence, mirrors are faking objects which give a reading access to information which are, for some of them, in the map of the object, like slot names and slot privacy attributes, and for the others, in the object itself, the slots values for mutable slots. Following the same pattern, slot descriptor objects (see Figure 3) we retrieve from a mirror by sending message `at:i` are also faking objects in the sense that they contain no information on their own. The protocol defined to access information on slots, with the messages `visibility`, `isAssignable`, etc. (see Section 3.2), also call primitives of the virtual machine. Consequently, mirrors are not really reflective entities but a mean to access – and not modify – system information through system primitives.

So, why Self has mirrors? A reasonable answer actually is linked to the above remarks on maps. It appears that maps are difficult to reify as Self objects. Nevertheless, an access to the information they store is often needed, for example when inspecting an object. Mirrors are closing the gap between the external Self language world of objects and the internal Self implementation world of maps by giving a reading access to this information. But, are mirrors really implementing structural reflection? Given the requirements to achieve structural reflection, i.e. a causally connected representation of program entities in the language itself, the answer is no. Mirrors are merely interfaces to system information but neither writing accesses nor the corollary causal connection are supported.

An open question is the following: if Self had provided a causally connected writing access to the implementation data structures representing objects, would we consider that as structural reflection? Or, stated in more general terms, are causally connected reading and writing accesses to system information through system primitives sufficient to implement structural reflection? Is a representation of system data structures in the language a requisite that cannot be bypassed?

4.3 What about traits?

Traits objects are shared repository for common behaviors (methods) and state thus they play the behavioral role of classes. But traits objects play no role in structural reflection; even though the methods they provide may assume some structural properties about the objects on which they can be applied, traits objects put absolutely no constraints on the structure of their inheriting objects. The lack of space refrains us from drawing a complete analysis of the role of traits in Self, but they still deserve a more comprehensive treatment.

5 Conclusion

An implicit conjecture in the litterature on reflective languages and systems is that classes and metaclasses elegantly solve the structural reflection problem. But, prototype-based systems throw away classes thus reintroducing them, such as in [?], is not a satisfactory answer. Prototypes also give the opportunity to reassess the question whether some new entities may do a better job at structural reflection than classes and metaclasses. In Self, maps have the interesting property to be created inductively and automatically from individuals but the actual map management strategy is rather limited. Reifying maps would have two objectives: to study automatically generated shared structural meta-objects and to experiment with alternative management strategy for them. Unfortunately, the reification of maps in Self is not so simple; it requires a modification of the virtual machine to take into account map-of links at the base language level. We could also modify the Self virtual machine to introduce behavioral reflection: this approach would ease the experimentation with alternative management strategies.

At this time, more research is still needed to firmly conclude about the relative advantages and disadvantages of the inductive approach of maps compared to the deductive approach of classes from both application design and structural reflection standpoints.

The second entity we studied are the mirrors. In contrast with maps, mirrors are Self objects but we have shown that they contain no information on their own. They are merely reading interfaces between the external world of Self objects an the internal world of the implementation. An open question is whether systems that offer causally connected reading and writing accesses to their internal data structures through system primitives can be considered as reflective systems or not.

Bibliographie

[Coi87] P. Cointe. Metaclasses are First Class: the ObjVLisp Model. *Proceedings of OOPSLA'87, ACM Sigplan Notices*, 22(12):156–167, December 1987.

[CUCH91] C. Chambers, D. Ungar, B.-W. Chang, and U. Hölzle. Parents are Shared Parts of Objects: Inheritance and Encapsulation in Self. *Lisp and Symbolic Computation*, (4):207–222, 1991.

[CUL89] C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of Self, a Dynamically-typed Object-Oriented Language Based on Prototypes. *Proceedings of OOPSLA'89, ACM Sigplan Notices*, 24(10):49–70, October 1989.

[Fer89] J. Ferber. Computational Reflection in Class Based Object-Oriented Languages. *Proceedings of OOPSLA'89, ACM Sigplan Notices*, 24(10):317–326, October 1989.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80 – The Language and its Implementation.* Addison-Wesley, 1983.

[HCCU90] U. Hölzle, B.-W. Chang, C. Chambers, and D. Ungar. The Self Manual, version 1.0. distributed with the Self software release, from Stanford University, July 1990.

[Kic90] G. Kiczales. Making Reflection Safe for Real-World Users. In WRMA90 [?].

[LaL89] W.R. LaLonde. Designing Families of Data Types Using Examplars. *ACM Trans. on Prog. Languages and Systems*, 11(2):212–248, April 1989.

[Lie86] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. *Proceedings of OOPSLA'86, ACM Sigplan Notices*, 21(11):214–223, November 1986.

[Mae87a] P. Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987.

[Mae87b] P. Maes. Concepts and Experiments in Computational Reflection. *Proceedings of OOPSLA'87, ACM Sigplan Notices*, 22(12):147–155, December 1987.

[Nak90] S. Nakajima. Metalevel Issues in a Prototype-based Object-Oriented Programming Language. In WRMA90 [**?**].

[Ste87] L.A. Stein. Delegation is Inheritance. *Proceedings of OOPSLA'87, ACM Sigplan Notices*, 22(12):138–146, December 1987.

[UCCH91] D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle. Organizing Programs without Classes. *Lisp and Symbolic Computation*, (4):223–242, 1991.

[US87] D. Ungar and R. Smith. Self: The Power of Simplicity. *Proc. of OOPSLA'87, ACM Sigplan Notices*, 22(12):227–242, December 1987.

[WRM90] *Informal Proceedings of the First Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA/ECOOP'90*, October 1990.

[WY88] T. Watanabe and A. Yonezawa. Reflection in an Object-Oriented Concurrent Language. *Proceedings of OOPSLA'88, ACM Sigplan Notices*, 23(11):306–315, November 1988.