

# On the lightweight and selective introduction of reflective capabilities in applications

Rémi Douence, Mario Südholt

École des Mines de Nantes, Nantes, France, {douence, sudholt}@emn.fr

## Abstract

Computational reflection is gaining interest in practical applications: modern software frequently requires strong adaptability conditions to be met in order to fit a heterogenous and evolving computing environment. Most of these applications only need limited but flexible reflective capabilities rather than sophisticated reflective languages and reflective runtime systems. However, there is no notion of a reflective application which is independent from an underlying reflective system. Our paper aims at filling this gap by means of a constructive definition: we introduce a reification technique that allows the transformation of non-reflective applications into reflective ones.

## 1 Introduction

Computational reflection is gaining interest in practical applications: modern software frequently requires strong adaptability conditions to be met in order to fit a heterogenous and evolving computing environment. Concretely, the JAVA programming environment heavily relies on the use of reflection for the implementation of the JAVABEANS component model and its remote method invocation mechanism JAVARMI. Furthermore, adaptability is a prime requirement of middleware systems and several groups are therefore doing research on reflective middleware [ref99].

Most of these applications are implemented using fully-fledged reflective languages and runtime systems. However, they frequently rely on limited reflective capabilities only: for example, the JAVABEANS model relies on the introspection of the structure of objects and JAVARMI relies on the interception of method calls. Neither of the two requires, for instance, the execution stack to be reified. Moreover, the set of required reflective capabilities may change during the life cycle of applications. Consequently, we advocate the use of reflective applications instead of reflective languages. Unfortunately, there exists no notion of reflective applications which is independent from an underlying reflective system.

In this paper, we introduce a reification technique which allows reflection to be introduced *à posteriori* by automatic transformation of non-reflective applications into reflective ones. The resulting applications exhibit reflective capabilities which are specifically-tailored to the application's needs and are only present at well-chosen places in the code. This technique can be interpreted as a constructive definition of reflective applications.

The paper consists of two main parts. Section 2 presents the transformation-based technique by exemplifying the introduction of reflection in order to expose internal representations and applying it to a simple example. (Note that the example is developed in JAVA; however, our technique can easily be adapted to other languages.) Section 3 discusses our approach from a compilation perspective and briefly presents other reflective capabilities which can be introduced by transformation.

```

class Point {
    private PolarCoord representation;
    Point(float d, float a) {
        this.representation = new Coord(d,a);
    }
    float getAbscisse() { return this.representation.getAbscisse(); }
}
class PolarCoord {
    float distance, angle;
    PolarCoord(float d,float a) {
        this.distance = d; this.angle = a;
    }
    float getAbscisse() { return this.distance * Math.cos(this.angle); }
}

```

Figure 1: Geometric points represented by polar coordinates

## 2 Exposing internal representations by reflection

From a software engineering point of view, reflection can be seen as providing a compromise between freely accessible white box systems and non-adaptable black box systems (for a motivation why neither of the two is sufficient see [kic97]). A typical use of reflection in the context of such “gray box” behavior is the (structural) modification of a representation (i.e. private data) of a black box. As a running example, consider the abstraction of geometric points shown in Figure 1 which internally represents coordinates in the polar coordinate system (`PolarCoord`).

In order to introduce reflection in existing black-box applications, we transform classes which need reflective capabilities. First, our reification technique introduces into classes an extra method `reify()` that exports private representations, as detailed for `Point` in Figure 2. We transform the internal representation from `PolarCoord` to `ReifiedPolarCoord` which is exported and provides a particular reflective capability. A reified representation may provide different reflective capabilities depending on its intended use. For example, in Figure 2, the class `ReifiedPolarCoord` makes explicit field accesses and the method-call mechanism. These reflective capabilities would make it easy to dynamically add a third field in order to represent three-dimensional points or to introduce a `rotate()` method at run time.

The transformations reifying field accesses and method calls are formally defined using the notation  $lhs \implies rhs$  in Figures 3 and 4, respectively. Note that these two transformations are orthogonal: they can be applied in any order in the case that both reflective capabilities are needed. Each transformation consists of two parts: the first part (a) transforms the representation class (e.g. by introducing a list of fields), the second part (b) transforms code that uses the representation (e.g. field assignments are replaced by a call to the method `write()`, which updates a list). Figure 3 details a compilation scheme for field accesses: a representation class `C` containing a sequence of field definitions  $\overrightarrow{fields}$  is transformed into a reified class `ReifiedC`. The reified class features a single field `listOfFields` which contains the same fields as `C`. Method calls are reified in the second transformation (see Figure 4): each method of the original representation class is transformed into a class of its own, each of which defines a method `apply()` enabling methods to be applied to the reified class. These methods are stored in a static field `listOfMethods` in the reified class.

Note that any of the two reification schemes provides a well-defined meta object protocol (henceforth MOP) of its own. The MOP of such a scheme consists of the interface enabling the manipulation

```

class Point {
  private ReifiedPolarCoord representation;
  Point(float d, float a) {
    this.representation = new ReifiedPolarCoord(d, a);
  }
  ReifiedPolarCoord reify() {
    return this.representation;
  }
  float getAbscisse() {
    return ReifiedPolarCoord.listOfMethods.lookup("getAbscisse")
      .apply(this.representation);
  }
}
class ReifiedPolarCoord {
  List listOfFields = List.nil.cons("distance").cons("angle");
  static List listOfMethods =
    List.nil.cons("getAbscisse", new getAbscisse_Method());
  PolarCoord(float distance, float angle) {
    this.listOfFields.write("distance", distance);
    this.listOfFields.write("angle", angle);
  }
}
class getAbscisse_Method {
  float apply(ReifiedPolarCoord that) {
    return that.listOfFields.read("distance")
      * Math.cos(that.listOfFields.read("angle"));
  }
}

```

Figure 2: Point represented by reifiable Polar coordinates

a) Change of representation

```

class C {  $\overline{\text{field}}$ ;  $\overline{\text{method}}$ ; } \implies
class ReifiedC {
  List listOfFields = List.nil. $\overline{\text{cons}}(\text{"field"}, \overline{\text{field}})$ ;
   $\overline{\text{method}}$ ;
}

```

b) Change of classes that use the representation (where `lexp instanceof C`)

```

lexp.field = rexp \implies lexp.listOfFields.write("field", rexp)
lexp.field \implies lexp.listOfFields.read("field")

```

Figure 3: Transformation of field accesses

a) Change of representation

```

class C {  $\overrightarrow{\text{field}}$ ;  $\overrightarrow{\text{method}(\overrightarrow{\text{arg}})\{\text{body}\}}$ ; }  $\implies$ 
class ReifiedC {
     $\overrightarrow{\text{field}}$ ;
    static List listOfMethods = List.nil.cons("method", new Method());
}

 $\overrightarrow{\text{class Method}}$  {
    Object apply(ReifiedC that,  $\overrightarrow{\text{arg}}$ ) {
        body where this  $\implies$  that
    }
}

```

b) Change of classes that use the representation (where `lexp instanceof C`)

```

lexp.m( $\overrightarrow{\text{exp}}$ )  $\implies$  ReifiedC.listOfMethods.lookup("m").apply(lexp,  $\overrightarrow{\text{exp}}$ )

```

Figure 4: Transformation of method calls

of the reified representation introduced by the transformation. For example, in order to add, remove or rename a field of a representation of a point `p`, the user could call methods to manipulate the list `p.reify().listOfFields`. In order to modify method call behaviors, he could subclass `List` and redefine `lookup()`.

Obviously, the `Point` example is quite simple. A more interesting example would be distributed applications which use a class `Proxy` implementing remote communication. Thanks to reflection, we could add at runtime a field to a proxy for profiling the network bandwidth. Furthermore, we could dynamically modify communication methods of the proxy in order to compress data to be transmitted according to the profiled bandwidth. In order to achieve this, we would make reifiable the class `Proxy` using our technique.

### 3 Reification transformations as compilation schemes

The program transformations which reify field accesses and method calls can be seen as compilation schemes: they decompose high-level mechanisms (such as field accesses) into lower-level computations (such as list accesses). If the transformations are applied to JAVA programs, these programs are transformed into a subset of JAVA. For example, once fields are compiled, transformed classes have only one field `listOfFields`. A main feature of this technique is its light weight: transformations can be applied locally and therefore reflective capabilities can be introduced in applications exactly where they are needed.

Other JAVA features can be compiled by transformation the same way. Each transformation exposes a different reifiable feature and ensures different constraints on the subset of JAVA constituting the target language. For example, a transformation exposing parameters which are passed to methods could ensure that methods have only one argument environment (which is a list of arguments). Table 1 lists several examples of JAVA features to be reified, the corresponding reified representation/structural restriction and potential applications of these features.

Note that a single feature may have several compilation schemes. For example, the transformation in Figure 4 compiles method calls using `lookup()`, a finer-grained scheme could split `lookup()` in

Reifiable feature	Reified representation/ Structural restriction	Application
Field access	1 field <code>listOfFields</code>	extending the class structure
Method call	1 field <code>listOfMethods</code>	introduce before-after methods
Parameter passing	1 parameter environment	vary parameter arities
Access to local variables	1 local variable frame	profile memory
Object creation	1 class <code>Instance</code>	profile object creation
Statement sequencing	no <code>return</code> but calls of continuations	define new control structures
Exceptions	make stack explicit (based on previous transf.)	define retry mechanisms

Table 1: Reifiable features

two parts: `send()` on the message-sending side and `receive()` on the receiver side (see [mca95] for a motivation).

Our technique is inspired from previous work on METAJ [met99], a reflective JAVA interpreter based on Smith’s tower-based interpreting method. METAJ also features selective reification of reflective capabilities but introduces an interpretation layer everywhere in a program. The work presented here can be seen as an optimization of the transformations used in METAJ: interpretation is replaced by compilation.

## 4 Conclusion

In this paper, we proposed a transformation-based technique to introduce reflection into applications. This technique has three main properties. First, it is selective because it features different (orthogonal) reflective capabilities expressed as transformations, but only required reflective mechanisms are introduced in applications. Each transformation introducing a reflective capability provides a different MOP which is specially-tailored towards a target applications. Moreover, since the transformations are orthogonal to one another, the reflective capabilities of an application can be easily extended by transforming an already transformed application. Furthermore, arbitrary reflective capabilities may be added by defining new transformations. Second, the technique is lightweight because reflective capabilities are introduced only where they are needed. Indeed, different subsets of the classes of an application can be selected for transformation by the programmer according to the required reflective capabilities. Third, the technique features reflective capabilities with a clear semantics because each transformation is formally defined. For lack of space, we could not detail the different program transformations. Further work is required to complete this collection of transformations and provide a large set of reflective capabilities. Furthermore, we intend to investigate properties, such as security properties, of reflective applications. We anticipate that the application of our technique at the source code level will facilitate this investigation.

## References

- [ref99] P. Cointe (ed.). Proceedings of Reflection’99, LNCS 1616, Springer Verlag, 1999.
- [kic97] G. Kiczales et al. Open Implementation Design Guidelines. ICSE, 1997.
- [mca95] J. McAffer. Meta-level Programming with CODA. ECOOP, LNCS 952, Springer Verlag, 1995.
- [met99] METAJ home page. <http://www.emn.fr/sudholt/research/metaj>