The next 700 reflective object-oriented languages

Rémi Douence, Mario Südholt

École des mines de Nantes

Dept. Informatique

F - 44307 Nantes Cedex 3

http://www.emn.fr/{douence, sudholt}

Abstract

Since Smith seminal work, there have been numerous reflective language definition and implementation proposals. These proposals, initially restricted to functional languages, have been quickly extended to object-oriented languages. Unfortunately, reflective object-oriented language definitions remained mostly ad hoc.

In this paper, we present a generic reification technique which enables the selective reification of arbitrary parts of object-oriented language interpreters. Our program transformation can be applied to different interpreter definitions. Each resulting reflective implementation provides a different meta-object protocol based on the original interpreter definition. This technique paves the way to a systematic study of reflective object-oriented language implementations.

Technical report no.: 99-1-INFO

Contents

1 Motivation and related work **3**

2 Smithsonian Reflection **4**

3 System architecture **4**
3.1 Overview 4
3.2 A simple non-reflective interpreter 5

4 Generic reification (DIY) **6**
4.1 Overview of the generic reification scheme 6
4.2 Example: making the class `Instance` reifiable 7

5 Reflective Programming **12**
5.1 Examples of reflective programming 12
5.2 What's going on backstage? 14

6 Conclusion and future work **14**

1 Motivation and related work

Smith has argued in [smi84] that reflection could become as common as recursion in programming languages. However, the lack of a clear semantics for reflective object-oriented programming features hinder its wide-spread use. Recently, component-based systems, where adaptability is a prime requirement, revived interest in reflection. For example, the Java programming environment provides a reflective API in order to support Beans, and reflective middleware [ref99] promises to add flexibility to coarse-grained component-based applications. As a special case, ORBs which offer dynamic invocation with the help of interface repositories emulate some reflective introspection facilities.

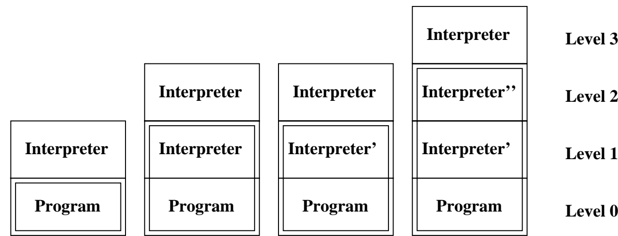
Smith's seminal work on reflective 3-Lisp [smi84] remains a key reference for reflective programming language design. He introduced the notion of reflective towers and proposed a reflective semantics for the language (but no effective implementation). This work was soon extended by different reflective Lisp implementations: 3-Lisp [riv84], Brown [fri84][wan86], Blond [dan88], Stepper [baw88], and [jef92].

Many reflective object-oriented languages have been proposed (e.g. ObjVLisp [coi87], CLOS [kic91], Smalltalk [bri89] [riv96], Classtalk [bri89], Neoclasstalk [riv96b]). All of them implement an ad hoc meta-object protocol: only a small fixed set of language features can be accessed using reflection. Furthermore, they lack a clear semantic model and introduce incompatible definitions of such notions as "meta-object" and "meta-class." The most "Smithsonian" approach for reflective object-oriented languages is 3-KRS described by Patty Maes in her Ph.D. thesis [mae87]. 3-KRS is implemented on top of Lisp and introduces objects providing access to the underlying Lisp data structures and functions. Unfortunately, few documentation (Chapter 6 in [mae87] and [mae87b]) and the lack of an available implementation prevents us from comparing 3-KRS and the approach presented in this paper.

In this paper, we present a generic reification mechanism for object-oriented interpreters based on program transformation techniques. This mechanism can be applied to different base interpreter definitions in order to automatically get different reflective interpreters. Each resulting reflective implementation provides a different meta-object protocol directly derived from the original interpreter definition. We believe this technique paves the way to a systematic study of reflective object-oriented language implementations.

The paper is structured as follows: in Section 2, we briefly introduce Smith's reflective towers. Section 3.2 is an overview of the system architecture underlying our approach and presents a simple (non-reflective) interpreter for a subset of Java. Our generic reification technique is detailed in Section 4, where it is applied to this interpreter in order to make it reflective. The resulting reflective interpreter is called METAJ (the implementation and examples are available on request from the authors). Section 5 is devoted to reflective programming: it details our reification technique at work by presenting several applications of METAJ. Finally, Section 6 discusses some issues concerning our future work.

Figure 1: Smithsonian reflective towers



2 Smithsonian Reflection

Smith's seminal work on reflective 3-Lisp [smi84] defines reflection with the notion of reflective towers. In Figure 1, the left hand side tower shows a user-written (i.e. level 0) *Program* in a double-square box and its *Interpreter,* which defines its operational semantics. A reflective computation is a computation about the computation, e.g. a computation modifying the interpreter. At run time, such a computation creates an extra interpretation layer by means of a reification operator "*reify*," so that the level 1 *Interpreter* becomes now part of the program: in the illustration, it is included in the double square box. We get a second tower with three levels. The *Program* can now modify the standard semantics of the language defined by the level 1 *Interpreter* to get *Interpreter*' as shown in the third tower. Finally, when a non-standard semantics *Interpreter"* of *Interpreter*' is required, a further extra interpretation level can be introduced as illustrated by the fourth tower. A classic example of reflective programming deals with introducing debugging traces. In Figure 1, the *Interpreter'* could generate traces as *Program* is evaluated. The extra interpretation layer *Interpreter"* of the fourth tower would be required to trace *Interpreter'*.

3 System architecture

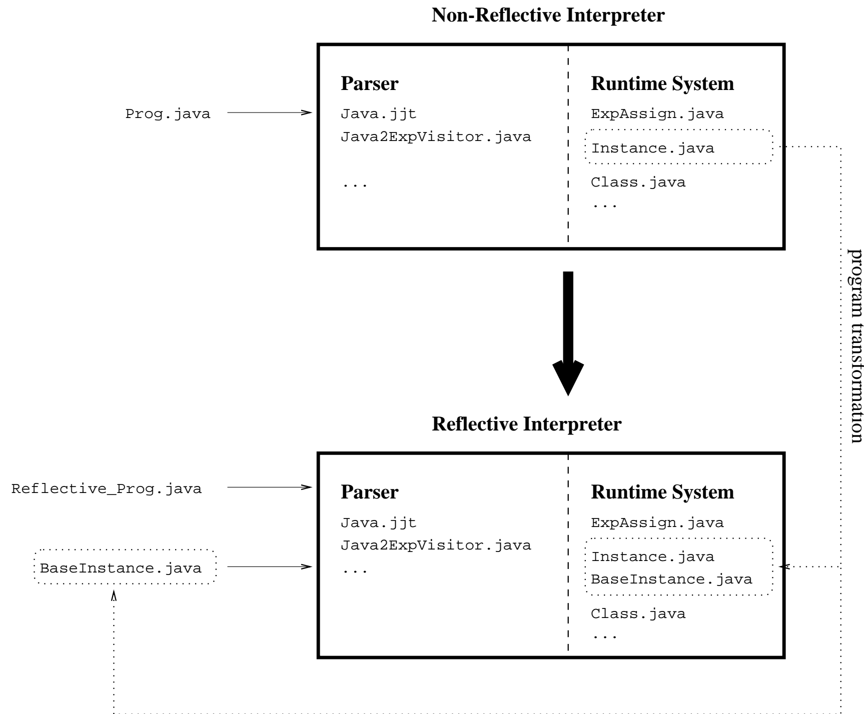
This section presents the system architecture underlying our approach to transform an object-oriented non-reflective interpreter into a reflective one. Furthermore, it briefly introduces our non-reflective Java interpreter implementation.

3.1 Overview

As shown in Figure 2, a non-reflective Java interpreter takes a non-reflective program `Prog.java` as input. This program is parsed into a syntax tree with nodes for the object-oriented and imperative features of Java. The runtime system of the interpreter then evaluates this tree.

In order to make this interpreter reflective, a subset of the interpreter classes is transformed. Basically, this transformation introduces two classes for each transformed class. For example, in Figure 2, the class `Instance.java` becomes `BaseInstance.java` and a different version of `Instance.java`. The reflective interpreter relies on the transformed classes (here, for example: `BaseInstance.java`) as input in order to build levels of reflective towers. Note

Figure 2: System architecture



that this design imposes that the interpreter is self-applicable.

We implemented one version of this system architecture, the resulting reflective Java interpreter of which is called METAJ. The parser has been implemented by means of JavaCC and JJTree (versions 0.8pre2 and 0.3pre6, respectively). The runtime system is operational with the JDK1.1.6 and Java2.

Note that the approach can be applied to any object-oriented language.

3.2 A simple non-reflective interpreter

Smithsonian reflection is constructed on the basis of a non-reflective interpreter. We have implemented in Java a non-reflective interpreter for a subset of Java, which provides support for all essential object-oriented and imperative features, such as classes, objects, fields, methods, local variables, assignment statements, ...

Each syntactic construct of our Java subset is defined by a corresponding class. For example, assignment statement, method-call, and class declaration expressions are respectively encoded by the classes `ExpAssign`, `ExpMethod` and `ExpClass`. All of these classes define an evaluation method `eval()`.

Besides of the classes representing syntactic forms, the interpreter defines a few other classes to implement an operational semantics. Most of them are self-explanatory: `Method`, `MethodList`, `Data` (which implements mutable memory cells such as fields), `DataList`, `Class`,

Figure 3: Original class `Instance`

```
class Instance {
  public Class instanceLink;
  public DataList dataList;
  Instance(Class instanceLink, DataList dataList) {
    this.instanceLink = instanceLink;
    this.dataList = dataList;
  }
  Data lookupData(String name) {
    return this.dataList.lookup(name);
  }
  ...
}
```

`Instance` and `Environment` (mapping identifiers to values). For illustration, the main part of the class `Instance` (which is used as a running example below) is shown in Figure 3.

4 Generic reification (DIY)

Smith's reflective towers are the most convincing reflection model because of its generality and conceptual simplicity: reflective programming can be understood simply by considering the operation "*reify*," which reifies part of the current interpreter — thus piling up a new level on the tower — such that it can be manipulated as part of the reflective program.

4.1 Overview of the generic reification scheme

Reification of an object should not change its semantics but it changes the object representation and provides access to this new representation. For example, in our non-reflective interpreter it is impossible to access the field list of an object (although such a list exists in the memory of the underlying implementation). The reified representation of an object provides access to this list. Once the internal representation has been exposed, access to these structures allows the semantics of the object to be changed (e.g. adding a new field to an object's field list).

Basically, a reifiable entity can have two different representations: either a base representation or a reified representation. Since reification of an object does not change its semantics, the object should provide the same method interface in both representations. This common interface is implemented using a dispatch object (the `Instance` denoted by `pair` in Figure 4). Beware, in the previous sentence, `Instance` denotes an instance of the class `Instance`. This simplified notation also occurs in the rest of this paper.

The dispatch object points to the active representation: either the base representation (`BaseInstance` in Figure 4a) or the reified representation (`Instance` denoted by `pair.reify()`

and BaseInstance in Figure 4b). Whether an object is accessed through its dispatch interface or through its reified exported representation (Instance denoted by pair.reify() in Figure 4b) is irrelevant (more precisely, modification of the object through one access path pair is visible through the other access path pair.reify()). Obviously, the two paths provides different interfaces: e.g. pair.fst and pair.reify().dataList.lookup("fst") in the example. When the base representation is active, the dispatch simply *delegates* incoming method calls to it. When the reified representation is active, the dispatch *interprets* the incoming method call.

Based on this implementation idea, our generic reification scheme is a program transformation which can be applied to an arbitrary class (e.g. Instance) of the original interpreter. The transformation consists of two main steps:

1. Introduce the class BaseInstance which defines the base representation of an original Instance.

2. Introduce the class Instance which defines the corresponding dispatch objects. This class provides the same method interface as the original class Instance and implements a method reify() which creates the reified representation and switches from the base representation to the reified one.

4.2 Example: making the class Instance reifiable

In this section, we apply this transformation in detail to the interpreter class Instance (cf. Figure 3 for its original definition).

First, we create the class BaseInstance, which defines the base representation of instances. As illustrated in Figure 5, we rename the original class and introduce an extra referent field. This field is used in some contexts (but not in the example considered here) to distinguish between the dispatch object and the active representation pointed to by the dispatch object. It is initialized by the constructor and points back from the BaseInstance to the dispatch object Instance.

Second, we create the dispatch class Instance shown in Figures 6 and 7. This class has three fields: baseRepresentation and refiedRepresentation that point to the two representations, and a boolean field isReified that discriminates the active representation. Its constructor initializes these fields and creates a base representation for the object by default.

The method lookupData() in Figure 6 has the same signature as its original version. When the base representation is active (i.e. isReified is false), the method call is delegated to the base representation. When the reified representation is active (i.e. isReified is true), the method call is interpreted: the method definition is fetched from the reified representation (lookupMethod()), a local environment is built from the method arguments (argsE. add()), and the method is evaluated (apply()).

Finally, the method reify() is defined (see Figure 7): a reified representation of the base representation is created by calling the method newNew() of an instance of the class Class whose definition is BaseInstance (cf. Figure 5). This operation actually creates an extra

Figure 4: Before and after reification of the object `pair`

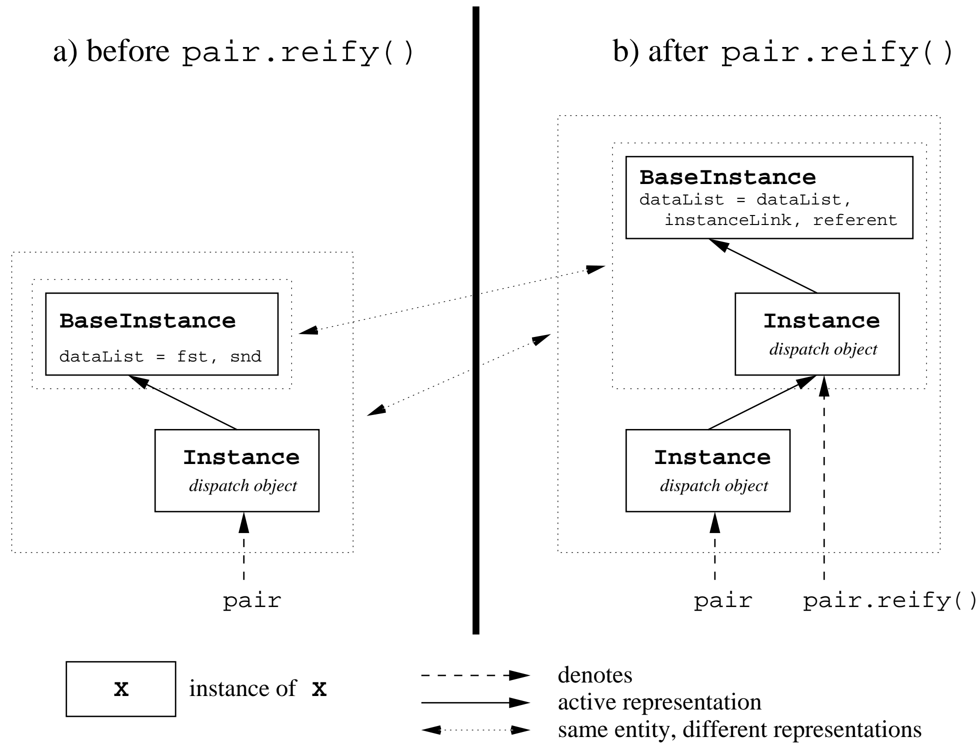


Figure 5: Class `BaseInstance`

```
class BaseInstance {
  public Class instanceLink;
  public DataList dataList;
  public Instance referent;
  BaseInstance (Class instanceLink, DataList dataList,
                Instance referent) {
    this.instanceLink = instanceLink;
    this.dataList = dataList;
    this.referent = referent;
  }
  Data lookupData(String name) {
    return this.dataList.lookup(name);
  }
  ...
}
```

Figure 6: Dispatch class `Instance` (Part 1/2)

```
public class Instance {
  public BaseInstance baseRepresentation;
  public Instance reifiedRepresentation;
  public boolean isReified;
  Instance(Class instanceLink, DataList dataList) {
    this.isReified = false;
    this.reifiedRepresentation = null;
    this.baseRepresentation =
      new BaseInstance(instanceLink, dataList, this);
  }
  Data lookupData(String name) {
    if (this.isReified) {
      // lookup for the method definition
      Method m = this.reifiedRepresentation.lookupMethod("lookupData");
      // build a new local environment with parameters
      Environment argsE = new Environment(null, null, null);
      argsE.add("name", new Data(name));
      // apply the method to its arguments
      Data result = m.apply(argsE, this.reifiedRepresentation);
      // unpack result
      return (Data)result.read();
    } else
      return ((BaseInstance)this.baseRepresentation).lookupData(name);
  }
  ...
```

Figure 7: Dispatch class `Instance` (Part 2/2)

```
  ...
  Data reify() {
    if (!this.isReified) {
      // create a fresh copy of the base class BaseInstance
      Environment baseClasses = Main.expBaseClass.eval(null);
      Class aClass = baseClasses.lookup("BaseInstance");
      // create an instance
      Instance instance = aClass.newNew();
      // copy the object state
      instance.lookupData("instanceLink")
        .write(((BaseInstance)this.baseRepresentation).instanceLink);
      instance.lookupData("dataList")
        .write(((BaseInstance)this.baseRepresentation).dataList);
      instance.lookupData("referent")
        .write(((BaseInstance)this.baseRepresentation).referent);
      // set interface state
      this.isReified = true;
      this.baseRepresentation = null;
      this.reifiedRepresentation = instance;
    }
    return new Data(this.reifiedRepresentation);
  }
}
```

Figure 8: ExpMethod class

```
public class ExpMethod extends Exp {
  ...
  Data eval(Environment localE) {
    // evaluate the lhs (object part)
    Object o = this.exp.eval(localE).read();
    // evaluate the arguments to get a new local environment
    Environment argsE = new Environment(null,null,null);
    this.args.eval(localE, argsE);
      if ((o instanceof DataList) && (this.methodId.equals("lookup"))){
      // DataList.lookup(String dataName)
      String dataName = (String)(argsE.getData().read());
      return new Data(((DataList)o).lookup(dataName));
    } else {
      // Instance case
      Instance o1 = (Instance)o;
      // lookup for the method field
      Method m = o1.lookupMethod(this.methodId);
      // apply the method to its argument
      return m.apply(argsE, o1);
    }
  }
}
```

interpreter layer. The allocated reified representation is initialized with the fields values of the base representation, and the result is installed as the active reified representation. Ultimately, a reference to this reified representation is returned.

The textual definition of the class `BaseInstance` was parsed at interpreter creation time and stored in `Main.expBaseClass`. Each call to `reify()` constructs a fresh copy of this `Class`, so that the behavior of each `Instance` it creates can be specialized independently from modifications to its class. Indeed, an `Instance` points to its `Class` in the interpreter memory. If sharing is required it must be set up explicitly.

Finally, the class `ExpMethod` denotes method calls, such as `exp.methodId(`exp_1`, ...,` exp_n`)`. In the original non-reflective interpreter, `exp` denotes an `Instance` (i.e. a source-level object) and the method call is interpreted. With the introduction of reflection, `exp` may denote other types. For example, the `dataList` field of a reified `Instance` points to a `DataList` object. So, interpretation of the `lookupData()` method implies interpretation of a call to the method `DataList.lookup()` (see the definition of `lookupData()` in Figure 3). This method call must not be interpreted, but delegated. This additional case must be added to `ExpMethod.eval()` as shown in Figure 8. As needed, extra cases must be introduced for the other interpreter classes.

We applied this reification technique to the classes defining the object-oriented features of our Java interpreter. The imperative features can be tackled analogously.

5 Reflective Programming

In this section, we express several classic examples of reflective programming in our framework. The goal of this presentation is twofold. First, these executable examples provide concrete evidence that our goal — designing a generic reification technique for object-oriented languages — has been achieved. Second, we hope that detailed examples of our reflective interpreter at work help the reader's understanding of the system architecture described in the previous section.

5.1 Examples of reflective programming

In the following, our examples highlight an important feature of our design: since our reification scheme relies on the original interpreter definition, the meta-object protocol of the corresponding reflective interpreter (i.e. the interface of an reflective system) is quite easy to apprehend. It consists of half a dozen key classes of the interpreter and `reify()`.

The most basic use of reflection in object-oriented languages consists in examining the structure of objects (aka introspection). Let us consider the problem of testing the existence of a field (i.e. basically accessing to the underlying representation of an object). In Figure 9, a class `Pair` is defined, and in `main` a new instance `pair` is created. In the interpreter, the object `pair` is represented by an `Instance` (see Figure 4a). Our generic reification method provides access to a representation of this `Instance` which we name `metaPair`. Fields of an instance are stored in an object of class `DataList` which provides a method `Boolean member(String dataName)` to check membership of the field called `dataName` in the current list. In our example, the programs outputs `false` (`third` is not a field of `pair`).

In METAJ, reflective programming is not limited to introspection, but the internal state of the interpreter can also be modified (aka. intercession). The second example in `main` shows how the behavior of an instance can be modified by changing its class dynamically. Imagine, that we would like to be able to print pairs using a method called `toString`. One way to achieve this using reflection is detailed in Figure 9: we define a class `PrintablePair` which extends the original class `Pair` and implements a method `toString`. A given pair can then be made printable by dynamically changing its class from `Pair` to `PrintablePair` (the `Instance` field `instanceLink` holds the class, see Figure 5). Afterwards the object `pair` understands the method `toString`.

Our last example deals with method call tracing for debugging purposes. The class `BaseInstance` of the interpreter defines the method `Method lookupMethod(String name)` that returns the effective method to be called within the inheritance hierarchy. In our interpreter each `lookup()` is followed by an `apply()`. Thus, method call tracing can be introduced with `BaseInstanceWithTrace` which specializes the class `BaseInstance` with a method `lookupMethod()` which prints the name of its parameter (see Figure 9). In order to install the tracing of method calls of the instance `pair`, its standard behavior (defined by

Figure 9: Reflective Programming with the class Pair

```
class Pair {
  public String fst;
  public String snd;

  Pair(String fst, String snd) {
    this.fst = fst;
    this.snd = snd;
  }
}

class PrintablePair extends Pair {
  public string toString() {
    return "(" + this.fst + "," + this.snd + ")";
  }
}

class BaseInstanceWithTrace extends BaseInstance {
  Method lookupMethod(String name) {
    // trace method-called
    System.out.println("method called: " + name);
    return this.instanceLink.methodList().lookup(name);
  }
}

class Main {
  void main() {
    Pair pair = new Pair("1", "2");
    // 1 - introspection: test existence of a field
    Instance metaPair = pair.reify();
    System.out.println(metaPair.dataList.member("third"));
    // 2 - intercession: dynamic class change
    metaPair.instanceLink = PrintablePair;
    System.out.println(pair.toString());
    // 3 - intercession: change method-call semantics
    Instance metaMetaPair = metaPair.reify();
    metaMetaPair.instanceLink = BaseInstanceWithTrace;
    System.out.println(pair.toString());
  }
}
```

the class BaseInstance) must be replaced by BaseInstanceWithTrace. Reification of pair provides access to an Instance whose instanceLink field denotes the Pair class. A sequence of two reification operations of pair provides access to an Instance whose instanceLink denotes the class BaseInstance. This link can now be set to the class BaseInstanceWithTrace. A call to a method of the object pair then prints the name of the method. Therefore, "toString" is printed by our example program. Finally, note that our tower-based reflection scheme would make it easy to trace the tracing code if required (any number of levels may be created by a sequence of calls to reify()).

5.2 What's going on backstage?

In this subsection we present in some detail the states of the interpreter memory corresponding to the reflection example discussed above.

In Figure 10, the memory footprint after the creation of the instance pair is shown. At this point of time the active representation of pair is the base representation denoted by the BaseInstance object **BI** (as already shown in Figure 4a). **BI** contains references to the class Pair and to the field list of pair. For the sake of simplicity, we focus on pair and do not detail the implementations of other objects: dispatch objects *and* representations of classes and data lists are collapsed in oval boxes.

In Figure 11, the memory footprint after the first reification operation pair.reify() is shown. At this point of time the active representation of pair is the reified representation denoted by the content of the dashed curve (cf. Figure 4b). In particular, the fields of **BI** (cf. Figure 10) are represented now by the three **DL** cells. Note that the class of **BI** of Figure 10 is now explicitly represented by a new Class object created from the original Java definition of BaseInstance.

In Figure 12, the memory footprint after the second reification operation pair.reify().reify() is shown. At this point of time the active representation of pair.reify() is the reified representation denoted by the content of the dashed curve. Here, the class of **BI** of Figure 11 is now also represented explicitly by a new Class object created from the original BaseInstance. Note that a fresh copy of **C** is allocated by each reification operation.

6 Conclusion and future work

In this paper, we presented a generic reification mechanism for object-oriented interpreters based on program transformation techniques. The most salient property of our approach is that it cleanly introduces Smith's reflective towers in object-oriented interpreters. A second interesting property, which is orthogonal to the first one, is that it can be applied to different base interpreter definitions in order to get different reflective interpreters. Each resulting reflective implementation provides a different meta-object protocol: it consists of *some* of the classes of the non-reflective interpreter and the reification operation reify(). We could, for example, allow reification of instances but forbid reification of classes by transforming the original definition of Instance and preserving the original definition of Class. Or, in order to get finer

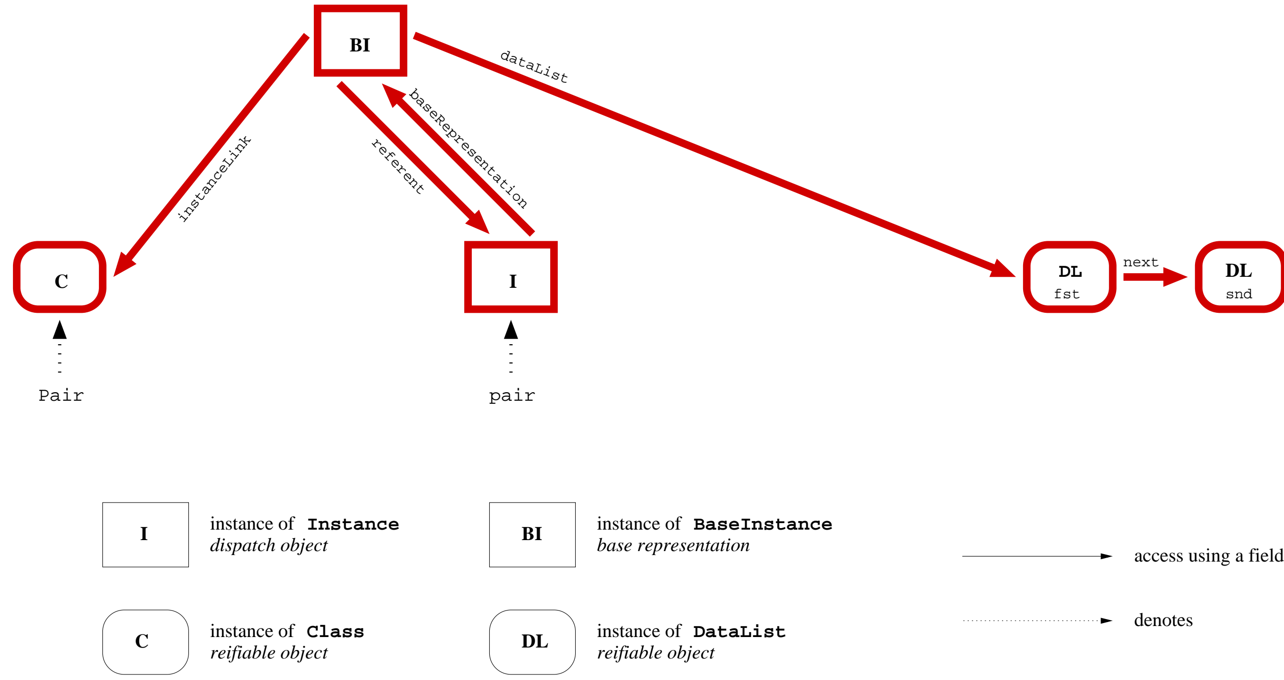


Figure 10: Memory footprint: after `new Pair()`

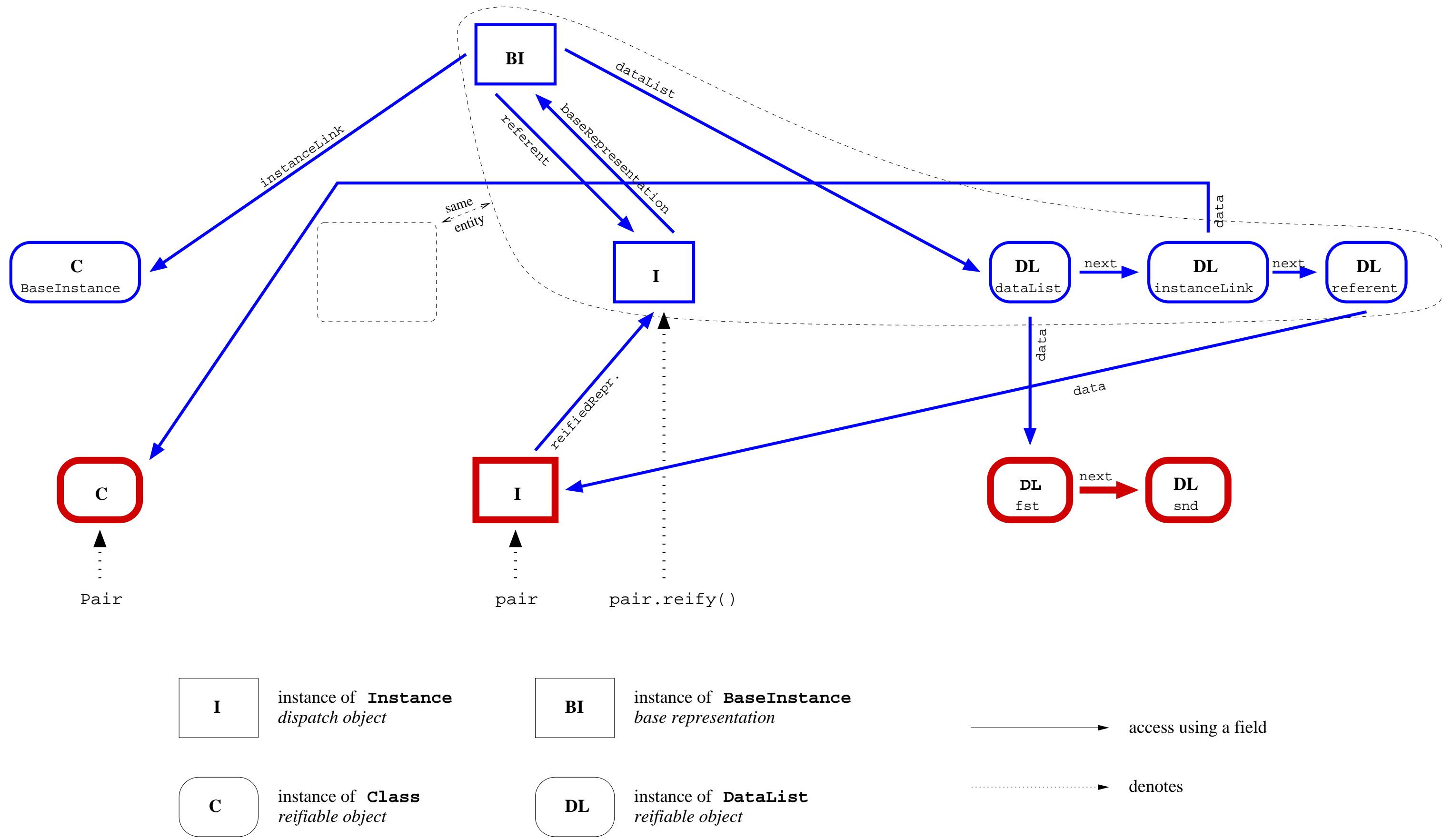


Figure 11: Memory footprint: after `pair.reify()`

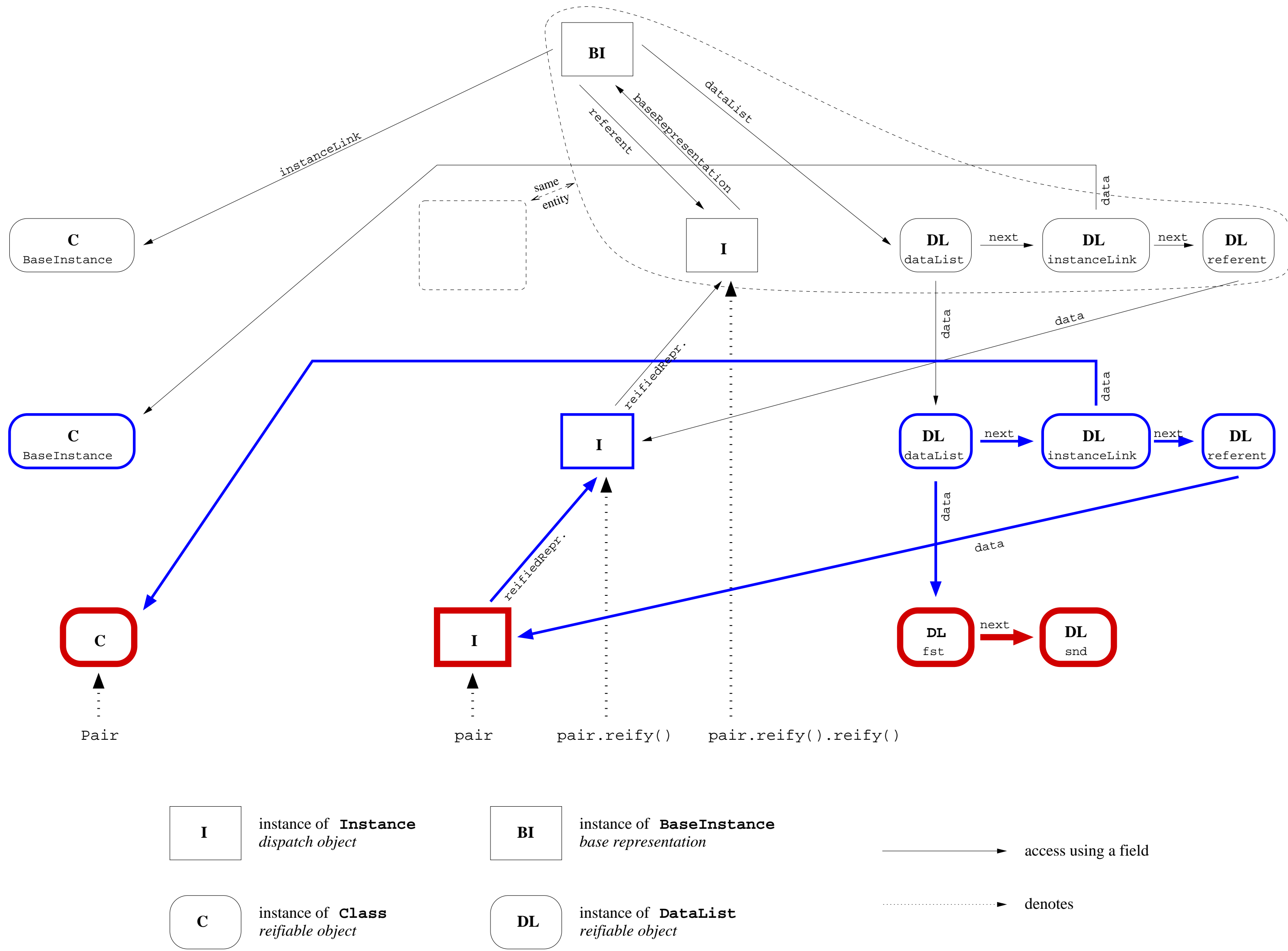


Figure 12: Memory footprint: after `pair.reify().reify()`

control over method application, for example, an alternative interpreter definition could split the operation `apply()` into two parts: `send()` and `receive()`. We believe this technique paves the way to a systematic study of reflective object-oriented language implementations.

Future work. First, our motto is: "the interpreter is the meta-object protocol." So, the design space of reflective object-oriented implementations should be explored by defining different non-reflective interpreters. Moreover, high-level reflective libraries encapsulating our primitive `reify()` would provide a means to ensure consistency conditions (such as "one instance and its class always have the same fields"). These techniques should be useful in the context of component-based systems, where adaptability is a prime requirement.

Second, reflection is deeply related to interpretation. However, the full expressiveness of reflective programming is seldom used in all applications. For example, the Java reflective API allows introspection and minimal intercession because this restricted model is sufficient for JavaBeans. So, specialization techniques like partial evaluation are prime candidates for efficiency improvements.

Finally, our current implementation revealed a number of technicalities, such as syntax extensions denoting a method without defining a class or the scope of interpreter classes (e.g. one definition of `BaseInstance` per reflective level). These questions should be formally studied by means of an appropriate formal semantics (e.g. [aba96], [cas97]).

References

[aba96] M. Abadi, L. Cardelli. A Theory of Objects. Monographs in Computer Science, Springer, 1996.

[baw88] A. Bawden. Reification without Evaluation. In Proceedings of ACM Lisp and Symbolic Computation, 1988.

[bri89] J.P. Briot, P. Cointe. Programming with Explicit Metaclasses in Smalltalk. In Proceedings of ACM Object Oriented Programming, Systems, Languages, Applications, pp 419-431, 1989.

[cas97] G. Castagna. Object Oriented Programming: A Unified Foundation. Progress in Theoretical Computer Science, Birkhäuser, 1997.

[coi87] P. Cointe. Metaclasses are First Class Objects: the ObjVLip Model. In Proceedings of ACM Object Oriented Programming, Systems, Languages, Applications, pp 156-167, 1987.

[dan88] O. Danvy, K. Malmkjær. Intensions and Extensions in a Reflective Tower. In Proceedings of ACM Symposium on Lisp and Functional Programming, pp 327-341, 1988.

[fri84] D.P. Friedman, M. Wand. Reification without Metaphysics. In Proceedings of ACM Symposium on Lisp and Functional Programming, pp 348-355, 1984.

[gol83] A. Goldberg, D. Robson. Smalltalk 80, the Language and its Implementation. Addison-Wesley, 1983.

[jef92] S. Jefferson, D.P. Friedman. A Simple Reflective Interpreter. In Proceedings of ACM New Models for Software Architecture, Reflection and Meta-Level Architecture, JSSST, IPSJ, 1992.

[kic91] G. Kiczales, J. des Rivières, D. Bobrow. The Art of the Metaobject Protocol. MIT Press, 1991.

[mae87] P. Maes. Computational Reflection. Ph.D. Thesis, Vrije University, Brussel, 1987.

[mae87b] P. Maes. Concepts and Experiments in Computational Reflection. In Proceedings of ACM Object Oriented Programming, Systems, Languages, Applications, pp 147-155, 1987.

[ref99] Proceedings of Reflection'99, Session on Middleware/Multimedia, LNCS 1616, to appear, Springer Verlag, 1999.

[riv96] F. Rivard. Smalltalk: a Reflective Language. In Proceedings of Reflection'96, pp 21-38, 1996.

[riv96b] F. Rivard. Evolution du comportement des objets dans les langages à classes réflexifs. Ph.D. Thesis, Université de Nantes, in French, 1996.

[riv84] J. des Rivières, B.C. Smith. The implementation of procedurally reflective languages. In Proceedings of ACM Symposium on Lisp and Functional Programming, pp 331-347, 1984.

[smi84] B.C. Smith. Reflection and Semantics in Lisp. In Proceedings of ACM Symposium on Principles of Programming Languages, pp 23-35, 1984.

[wan86] M. Wand, D.P. Friedman. The mystery of the Tower Revealed: a Non-Reflective Description of the Reflective Tower. In Proceedings of ACM Lisp and Symbolic Computation, 1986.