ChromaKey: Towards Extensible Reflective Architectures

Pablo Inostroza
Centrum Wiskunde & Informatica, The Netherlands
pvaldera@cwi.nl

Tijs van der Storm
Centrum Wiskunde & Informatica, The Netherlands
storm@cwi.nl

Abstract

Reflection allows programmers to inspect and modify the structural and runtime properties of a software system. Traditionally, the architecture of a reflective system has been a monolithic part of the runtime system, featuring a fixed semantics. Mirror-based reflective architectures decouple the base-level entities from their meta-level counterparts. In this work, we explore ChromaKey, a design to go yet one step further. ChromaKey enables the user to extend the reflective system in two dimensions: the semantics of reflective objects and reflection on syntax extensions of the host language. The first axis decouples the reflective system from a specific mirror interface. The second axis allows existing hierarchies of reflective objects to be extended. The key component is a generic reflecting component that "parses" class definitions according to a given semantics, specified by generic factories (Object Algebras).

1. Introduction

Reflection allows programmers to inspect and modify the structure and behavior of programs. In object-oriented systems, this applies to the structure of objects and classes (field, methods), and operations on objects (field access, field update, method invocation).

Well-known reflective architectures couple base-level objects with their meta-level counterparts. In Java, for instance, the getClass() method (that gives access to a reified class, and through it, to Java's reflective facilities) is part of the interface of an Object. In 2004, Bracha and Ungar proposed mirror-based architectures to decouple the base level from the reflective meta level. This allows VM programmers to provide different implementations for the reflective capabilities, as long as they conform to the mirror interfaces.

In this paper we explore ChromaKey, a mirror-based design that goes one step further, by decoupling the reflective system from predefined mirror interface. This allows programmers to supplant the reflective system with completely new hierarchies of "mirror"-like objects, for purposes that may go beyond the traditional use cases of reflection.

	Code example	#iface	#impl	Extensible?
Standard	Class cls= object.getClass();	1	1	no
Mirrors	ClassMirror clsMir = Ref.reflectCls(cls);	1	n	no
ChromaKey	MyClassMir chrKey= Ref.reflectCls(cls, sem);	n	n	yes

Table 1. Comparison of different reflective architectural styles, showing a code snippet for reifying a class; the number of different interfaces that a reification can produce; the number of supported implementations; and whether the design supports reflecting upon language extensions.

This flexbility is summarized in Table 1. The two upper rows of table show how the Java standard reflection model compares to mirror-based architectures. The code snippet in the second column shows how to obtain the reified representation of a class. In traditional reflection, the number of interfaces and implementations is fixed. In the case of mirror-based interfaces, the number of implementations is open-ended, but there still is a single interface.

The third row of Table 1 exemplifies ChromaKey. The mirror-style reflectCls operation receives an additional factory argument sem representing the custom semantics. The resulting reification can be of arbitrary type and does conform to the mirror interface provided by the language. Thus, in addition to decoupling the implementation of meta objects from the base level (as in mirror-based architectures), ChromaKey also allows the interfaces of the reified objects to be changed.

Apart from decoupling the reflective system from a concrete interface hierarchy, ChromaKey supports a powerful form of extensibility of reflective hierarchies. This has applications for extensible programming languages where the built-in reflective system cannot anticipate all language constructs a programmer might want to reflect upon.

We first introduce the ChromaKey design (§2) to make reflection more flexible by removing the "one-interface" constraint of mirror-based architectures. We show how mirror-

Meta'16 October 30, 2016, Amsterdam, Netherlands

```
// entry point for reflecting classes
<C,M,F,B> C reflect(Class c, ClassAlg<C, M, F, B> sem);

// type of "semantics factories" (Object Algebra)
interface ClassAlg<C, M, F, B> {
  C Class(String name, List<M> members);
  M Method(F formal, List<F> params, B body);
  M Field(F formal);
  F Formal(String name, Class type);
  B Method(java.lang.reflect.Method m);
}
```

Figure 1. ChromaKey: generic reflect, parameterized by an Object Algebra

based reflection falls out of the design, simply by instantiating ChromaKey to produce mirror objects. After introducing the architecture, we discuss examples of how custom semantics enable different object structures for the reification (§2.2, §2.3). In §2.4 we describe how ChromaKey provides for reflection on language extensions. Finally we describe an application of ChromaKey in §3, implementing Managed Data in Java (Loh et al. 2012; Zacharopoulos et al. 2016).

2. ChromaKey

2.1 Design

The core idea behind ChromaKey is semantics-parametric reflection, expressed by the definitions in Figure 1.

The reflect static method takes two parameters: the first corresponds to the class that is being reflected, and the second represents a generic factory object representing the semantics that the reflection process uses when traversing the class definition. The semantics is specified by implementing the Object Algebra interface ClassAlg shown at the bottom (see Oliveira and Cook 2012).

The methods of ClassAlg are called when constructing the reflective representation of the class in question. The reflect method acts as a custom "parser" of Java class definitions, that outputs specific objects, depending on the semantics specified by an implementation of the algebra. Because the method reflect is generic, it represents a *semantics-parametric reifier*. The implementation details of reflect are outside of the scope of this paper. The following section gives examples of how this design affords additional flexibility.

2.2 Regaining Original Mirrors

Using ChromaKey, it is naturally possible to regain the original mirror-based design by implementing the ClassAlg in such a way that mirror objects are produced that conform to the existing mirror interface:

```
class JavaMirrors
  implements ClassAlg<ClassMirror, MemberMirror,
                      FormalMirror, BodyMirror> {
  ...
}
```

JavaMirrors instantiates the generic interface ClassAlg to return objects conforming to the mirror interfaces (e.g., ClassMirror). Given that the VM would provided causally connected implementations of the mirror interfaces, it is now possible to get mirror-based reflection facilities:

```
mirror = reflect(Point.class, new JavaMirrors());.
```

2.3 Alternative Semantics: Reflecting onto Strings

Since the semantics of reflection is now arbitrary, it is possible to use the same generic infrastructure to "reflect" classes into different kinds of object structure, possibly defined by the programmer. For instance, classes could be reflected onto text to inspect their syntactic contents. The following algebra would achieve this:

```
class ToString implements ClassAlg<String,String,String,String> {
  public String Class(String name, List<String> members) {
    return "class " + name + "{" +  ... + "}";
  }
  ...
}
```

The following client code outputs a textual representation of the class Point.

```
String s = reflect(Point.class, new ToString());
System.output.println(s) // outputs "class Point{ ... }"
```

2.4 Reflection in the Presence of Syntax Extension

As we have pointed out, the ChromaKey reflect method supports a form of custom "parsing" of Java classes or interfaces.

Consider, for instance, this hypothetical Java extension for primary key fields:

```
class Person{
  key String email;
  String name;
}
```

In this class, the email field is tagged with the key modifier, but this is a (user level) language extension. The base reflection system does not know about this construct, so reflecting Person would lead to an incorrect representation of the classes, or worse, an error would be thrown.

ChromaKey leverages the extensibility of Object Algebras to address this situation. First, we assume that reflect will map unknown constructs (such as key) to specific methods in the provided algebra. Second, both the algebraic interface (i.e., ClassAlg) and the factories need to be extended to deal with the new construct.

First, the signature:

```
interface KeyAlg<C, M, F, B> extends ClassAlg<C, M, F, B> {
  M Key(F formal);
}
```

Given this new interface, existing semantic factories can be modularly extended as well. For instance, the ToString factory is extended as follows:

```
class KeyToString extends ToString
   implements KeyAlg<String, String, String, String> {

  public String Key(String formal) {
    return "key " + formal + ";";
  }
}
```

Since KeyToString is a subtype of ClassAlg, it can be passed to reflect:

```
String s = reflect(Person.class, new KeyToString());
// s = "class Person{ key String email; ... }"
```

In our proof-of-concept implementation of ChromaKey, syntax extensions are realized through Java annotations. For instance, the key modifier is represented as a @Key annotation. Our reflect method interprets these annotations and calls the Object Algebra methods using reflection. In future work we will explore approaches where the reflect "parser" can be made extensible as well, in the style of Biboudis et al. (2016).

3. Application: Managed Data

ChromaKey provides flexibility to the programmer to implement custom reflection systems. However, supporting behavioral and structural intercession requires support from the VM. In this section we sketch an approach based on ChromaKey, which circumvents this problem by focusing on Java 8 interfaces and dynamic proxies to implement powerful forms of reflection semantics. One application area is managed data, where the behavior of data objects is determined by runtime interpreters of data descriptions (Loh et al. 2012; Zacharopoulos et al. 2016).

For instance, a simple Point "class" could then be defined as follows:

```
interface Point {
  int x(int ...x);
  int y(int ...y);
}
```

The x and y methods are intended as fields: calling them with zero arguments is interpreted as field access, while calling them with exactly one argument is interpreted as field update.

Managed Data interprets an interface like this as a description of a data structure, which is provided with an implementation by interpreting it at runtime. A simplified ChromaKey implementation is shown in Figure 2. The interface Class is the carrier type for class reflection; it declares a single method New to create new objects. The class Managed implements the ClassAlg binding the generic type parameters C and M to Clazz and Member (which is not shown).

Whenever a class is reflected through Class a new Clazz is returned which realizes New by creating a new proxy for the Java class corresponding to name. The invocation handler is obtained through the extension point manager, which in this case returns a DefaultManager.DefaultManager implements the InvocationHandler interface to intercept any method invocation. It will parse the requests and then delegate to various helper methods such as setField. This architecture resembles the design of *mirages* (Mostinckx et al. 2007), which are base objects whose semantics are defined by mirrors. In our case, the proxy acts as the base object which is backed by a mirror-like manager that defines the runtime semantics.

```
interface Clazz { Object New(Object ...args); }

class Managed implements ClassAlg<Clazz, Member, ...> {
  public Clazz Class(String name, List<Member> members) {
    Class cls = Class.forName(this.name);
    return new Clazz() {
      public Object New(Object ...args) {
        return Proxy.newProxyInstance(cls.getClassLoader(),
            new Class[] {cls}, manager(args, name, members));
      }
    };
  }
  ...
  protected InvocationHandler
      manager(Object[] args, String n, List<Member> ms) {
    return new DefaultManager(args, n, ms);
  }
}

class DefaultManager implements InvocationHandler {
  ...
  @Override
  public Object invoke(Object p, Method m, Object[] args) {
    ... // emulate method invocation.
  }
  protected void setField(String x, Object val) {...}
}
```

Figure 2. Instantiating ChromaKey to implement Managed Data

Here is an example of how to create a simulated Point object using this infrastructure:

```
Clazz point = reflect(Point.class, new Managed());
Point p = (Point)point.New();
p.x(3) // Assigns 3 to the simulated field x
```

Simply emulating default Java behavior via interfaces and proxies, however, does not provide much benefit. The point is that the factories can be extended and modified through inheritance. For instance, the following manager instruments the setField method to log all assignments.

```
class LoggingManager extends DefaultManager {
  @Override
  protected void setField(String x, Object val) {
    System.out.println("Assigning " + x);
    super.setField(x, val);
```

```
  }
}
```

This custom data manager can be inserted into ChromaKey by override the manager method of Managed. For instance, the above client code can be modified as follows in order to produce a point object which log their field assignments:

```
Clazz point = reflect(Point.class, new Managed() {
  protected InvocationHandler manager(...) {
    return new LoggingManager(...);
  }
});
Point p = (Point)point.New();
p.x(3) // Assigns 3 to x, and logs "Assigning x"
```

ChromaKey allows us to go yet one step further by combining managed data with syntax extension as introduced in §2.4. Let's assume we want to support immutable fields, which would be declared as follows:

```
interface Point2 {
  immutable int x(int ...x);
  immutable int y(int ...y);
}
```

The data manager should now reject assignment to the fields x and y after they have been initialized. The pattern to implement this, is a combination of a key-like extension of ClassAlg (see §2.4) and overriding the setField method.

The following code shows how the simulation of **const** works in practice:

```
Clazz point = reflect(Point2.class, new ImmManaged());
Point2 p = (Point2)point.New();
p.x(2) // initializes x to 2
p.x(3) // throws exception since "field" is immutable
```

4. Open Questions

Reflecting on Objects Mirror-based reflection architecture also support the creation of object mirrors, obtained directly from an object. Object mirrors are an aspect of mirror systems that is orthogonal to ChromaKey. In essence, ChromaKey requires some base, "source like" representation, the structure of which is defined in the Object Algebra interface. It is currently unclear whether this makes sense to do for the structure of an actual object as well.

Type safety and Generic Mirrors Looking at the signature of reflect one may observe that the types do not track the actual type of the the class that is reflected on. Unfortunately, Java's type system lacks type constructor polymorphism which is needed to accurately represent this. In pseudo-notation, the signature would then be:

```
<T, C<_>, M, F, B> C<T> reflect(Class<T> c,
    ClassAlg<C<T>, M, F, B> alg);
```

An additional consequence is that our implementation of managed data requires down-casts when invoking the New method. We are currently experimenting with a technique to simulate higher-kinded polymorphism in Java (Biboudis et al. 2015), which provides type safety at the cost of considerably boilerplate. Further research is needed to explore the ramifications of this encoding for the other aspects of ChromaKey, such as extensibility.

Applications Our primary goal is to design reflective capabilities for extensible languages where the runtime behavior of objects can be (re)defined by the programmer. It is instructive to ponder whether ChromaKey has other applications, besides managed data. Arguably, reflecting class definitions on to text might not be very useful. We expect, however, that reflecting on to different class hierarchies has numerous applications in defining custom views, translations, or editors for class definitions. A key question, however, is how much causality the host language would allow in order to manipulate class definitions through these derived views.

5. Conclusion

We have introduced ChromaKey, an extensible design for mirror-based reflection APIs. ChromaKey allows programmers to define a custom canvas for reflecting classes on. Instantiating ChromaKey using a given mirror hierarchy produces the regular, mirror-based reflection design. Thus, mirror-based reflection can be seen as but a special case of ChromaKey. In addition to decoupling the reflective system from the meta object type hierarchy, the pattern supports reflecting on classes which use language construct not anticipated by the built-in reflection. As such, ChromaKey can be seen as first step toward reflection for (user) extensible languages. The key component behind ChromaKey is a semantics-parametric reifier that generically builds class representations based on factories (Object Algebras).

References

A. Biboudis, N. Palladinos, G. Fourtounis, and Y. Smaragdakis. Streams a la carte: Extensible pipelines with object algebras. In *ECOOP*, pages 591–613, 2015.

A. Biboudis, P. Inostroza, and T. van der Storm. Recaf: Java dialects as libraries. In *GPCE*. ACM, 2016.

G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA*, pages 331–344, 2004.

A. Loh, T. van der Storm, and W. R. Cook. Managed data: modular strategies for data abstraction. In *Onward!*, pages 179–194. ACM, 2012.

S. Mostinckx, T. Van Cutsem, S. Timbermont, and E. Tanter. Mirages: Behavioral intercession in a mirror-based architecture. DLS '07. ACM, 2007.

B. C. d. S. Oliveira and W. R. Cook. Extensibility for the masses: practical extensibility with Object Algebras. In *ECOOP*, pages 2–27. Springer, 2012.

T. Zacharopoulos, P. Inostroza, and T. van der Storm. Extensible modeling with managed data in Java. In *GPCE*. ACM, 2016.