A Metaobject Protocol for CLforJava

Jay Cotton
MS Graduate
College of Charleston
Charleston, SC 29424
jay@fleeingrabbit.com

Jerry Boetje
Computer Science Department
College of Charleston
Charleston, SC 29424
boetjeg@cofc.edu

ABSTRACT

CLforJava is a new implementation of Common Lisp that intertwines its architecture and operation with Java. The authors describe a new architecture for a CLOS MOP that supports transparent, bi-directional access between Lisp and Java. The access requires no special techniques nor syntactic mechanisms on the part of the programmer - being either Java or Lisp. The core of the new MOP is a data structure that melds the fundamental structures of Java instances (N-tuples) and CLOS instances (2-tuples) in such a way that the respective object systems can interact without cumbersome translations. Methods from their respective object systems can interact freely. We discuss certain aspects of the respective MOPs that prevent a complete integration and replacement of one system by the other.

Category and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features – *abstract data types, polymorphism, control structures*

General Terms

Design, Languages, Experimentation.

Keywords

Lisp, Java, Objects, MOP, software

1. INTRODUCTION

1.1. The Overall Project

CLforJava is a new, ground-up implementation of Common Lisp (CL) built to run on the Java Virtual Machine. The project is the prime component of the capstone course in the CofC's CS educational program. Now in its fourth year, the development is carried on by undergraduate CS students at the College of Charleston.

The bulk of the work is done in the Software Engineering Practicum class [1 3]. Each semester the students design, implement, and test components of the CLforJava application. The education goal is to present them with a large, complex product and an industrial-like development environment and process. The side-effect is that they become familiar with advanced Java techniques and attain some fluency in Lisp. And of course, there will be a new CL implementation, available as Open Source.

The project has also attracted some advanced undergraduate and graduate students to work on some of the more sophisticated components of the product, e.g. the compiler, customizable documentation system, non-positional number support, and now, CLOS.

1.2. Goals and non-Goals

The goals of the project are:

- A complete implementation of the ANSI Common Lisp standard
- The intertwining of Java and Lisp (see next)

The non-goals are:

- Blazing speed
- Industrial-level support

1.3. Intertwining of Lisp and Java

The most interesting technical goal of this project is the "intertwining" of the Java and Lisp environments. The origin of this concept came from the author's prior experience with Foreign Function Interfaces (FFI) and their overt clumsiness and lack of esthetics.1 Our intertwining mantra is that a Java programmer is enabled to create and access Lisp entities as any Java library and a Lisp programmer is equally enabled to create and access Java entities using the usual CL mechanisms. In CLforJava, there is no explicit FFI, nor are there special syntactical rules. Some other Lisps such as JScheme and Common Larceny provide a clever syntactic sugaring to access Java entities. CLforJava attempts a semantic integration between the two languages by intertwining

[1] cf VAX Lisp™

their respective object systems. The following sections provide a brief description of the Java ⇒ Lisp direction. The Lisp ⇒ Java direction is discussed in .Accessing Java from CLforJava.

These intertwining mechanisms are not intended to replace the use of one or the other languages. For example, writing an equivalent Lisp program entirely in Java would be a long and tedious effort. On the other hand, creating complex Java classes from Lisp would be an equally difficult undertaking

1.3.1.Type Representation

Given the "tangled" nature of the CL type system, CLforJava models all CL types in Java interfaces, making use of the ability of interfaces to multi-inherit. The intertwining of the languages starts here. Each interface has a static field holding a Lisp symbol that names the Lisp type. The symbol holds a reference to the interface. For all interfaces that define an instantiable type (e.g. **`Integer`**), there is a nested **`Factory`** class containing one or more **`newInstance`** methods. This use of parallel hierarchy of interfaces for CL types provides the Java programmer a simple method of determining if an object is a Lisp entity: **`(someObject instanceof lisp.common.type.T)`**.

Many types carry the Java method signatures for the Lisp functions they support. For example, **`Number`** defines signatures such as **`plus`**, **`minus`**, etc. **`List`** defines for example, **`car`**, **`rest`**, **`push`**. For the convenience of the Java programmer, classes such as **`List`** implement corresponding Java utility method sets such as **`Collection`**.

1.3.2.Function Representation

Lisp functions are defined as Java classes that implement the **`lisp.common.type.Function`** interface. This interface specifies an **`apply`** method that takes a list of values and returns an **`Object`**. It also defines a static field that holds the symbol naming the function (named and unnamed functions). Named function classes have an additional static field that holds an instance of the class; they also have **`private`** constructors enforcing singleton instances. An optimization is the inclusion of one of more **`Function`**N (where N is 0 to a specified limit) interfaces. Each of these interfaces defines a **`funcall`** method that takes N arguments. Most of the CL functions have a well-defined set of parameters, making it possible to make direct calls to the **`funcall`** methods, thereby improving speed.

1.4. CLOS in CLforJava

The design and implementation of CLOS in CLforJava evolved under the same intertwining constraints as the rest of the system:

- Types are defined by Java interfaces
- Factory classes create instances
- Java accessor methods are defined for each type
- Executable instances implement the **`Function`** interface

The primary challenge was the implementation of the complete CLOS system in an existing, classical object system. "Classical" here means "derived from or influenced by the object model of Simula 67. [2]" The authors considered these approaches:

- Write the entire system in Lisp. This would be the preferred method if we were only considering the Lisp programmer.
- Write the core of the system in specialized Java code that ignored the rest of the CLforJava architecture.
- Attempt to be smart by examining the Meta Object Protocols of both languages and building a core Lisp MOP functionality within the limitations of the basic Java MOP. Of course, being Lisp, there are different Lisp MOPs. We chose the implementation described in AMOP as the target.

In the main, the effort was successful. We were able to construct the complete metaobject class structure within our Java-based type system including the meta-circular aspect of the class **`standard-class`**.

2. DEFINING THE CLFORJAVA MOP

2.1.Goals

Simply put, the goal is to create a core implementation of a CLOS MOP in Java that adheres to the CLforJava intertwining principle. By "core implementation", we mean implementations of the most basic components of the AMOP:

- the metaobject class structure,
- the meta-circular component **`StandardClass`**,
- the metaobject classes **`GenericFunction`** and **`StandardMethod`**, accessible to Java programmers as any Lisp function

2.2.Defining Transparency

In the CLforJava intertwined environment, transparency is the ability of a programmer fluent in one language to access, inspect, and manipulate objects created in the other language and to create instances of classes written in the other language. Transparency does not imply that one system can completely replace the other. For example, The CLforJava MOP cannot define arbitrary Java classes.

2.3.Restrictions Due To The Java MOP

While risking the destruction of some suspense, there are a few aspects of the respective MOPs that we were unable to map completely. In Java, class definition includes slots and active code (methods) to manipulate them. The creation of the class is an atomic action that binds the slots and code together. CLOS on the other hand explicitly separates the slots from the code. In CLOS it is always possible to create a new method specialized to this class at any time after class definition. While the Java **`Instrumentation`** class supports some level of redefinition of existing classes, it is not yet as powerful as CLOS. However, our overall goal is just intertwining, not replacement.

---

[2] The authors are indebted to Reviewer 02 for this delightful turn of a phrase.

3. CORE IMPLEMENTATION OF THE CLFORJAVA MOP

3.1.Classes and Instances

3.1.1.Structure of CLOS Instances

The authors explored a variety of Java constructs to represent and optimize the behavior of CLOS instances. Several early prototypes used a different Java class to represent each CLOS defined metaobject. Under this system, an instance of **`standard-class`** (from the CLOS perspective) had a one-to-one correspondence with an instance of the Java class **`StandardClassImpl`** which implemented the **`StandardClass`** interface (in accordance with general CLforJava protocol). This Java object had fields corresponding to each of the slots of the CLOS object (or metaobject), plus a **`classOf`** field pointing to the CLOS class from which it was instantiated, and, in the case of an instance of **`standard-class`**, an extra slot referencing a Java Class object which was called upon to accomplish instantiation. The appearance of this structure on whiteboard after whiteboard, a vertical bar with a set of horizontal tines representing required instance data, earned its nickname as the "backbone" structure.

[The backbone became synonymous with the concept of a unified CLOS instance structure, and the name stuck despite the fact that the current structure bears little resemblance to its first iterations. Early versions of the backbone, while perhaps more immediately intuitive and invitingly simple, proved difficult to conceptualize consistently over the long term. CLOS object slots were mixed with object metadata and both were represented across a flat set of Java instance fields, which provided a misleading equivalence between Java fields and CLOS slots. This required a complicated and inconsistent treatment of Java fields during the instantiation process depending on which was being instantiated: a metaobject class, a regular class, or a regular object. The challenge was to mesh a traditional CLOS instance 2-tuple representation (class + slots) with a Java instance N-tuple representation (N fields). The former would allow for a more consistent treatment in its role as a CLOS metaobject, while the latter would let Java programmers process the instance with standard Java techniques.

The revised backbone structure is always an instance of the Java class **`ClosInstance`**, itself an implementation of the **`StandardObject`** Java interface. The presence of the **`StandardObject`** interface enables a Java program to immediate determine that CLOS semantics are in order. The backbone always contains two fields: **`closClass`** and **`javaInstance`**. The **`closClass`** field implements the link to the CLOS class object from which it was instantiated. The **`javaInstance`** field points to a Java object which implements slot storage, providing a more limited and specific equivalence between Java fields and CLOS slots, and in the case of Lisp classes, executable code to produce instances.

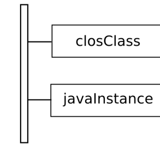

Figure 1: Basic Backbone Structure

This structure allows all CLOS instances to be represented and processed uniformly: detection of differences between types of instance (eg., object vs. metaobject) is shifted from reliance on what the fields are to the *values* of the fields are.

.

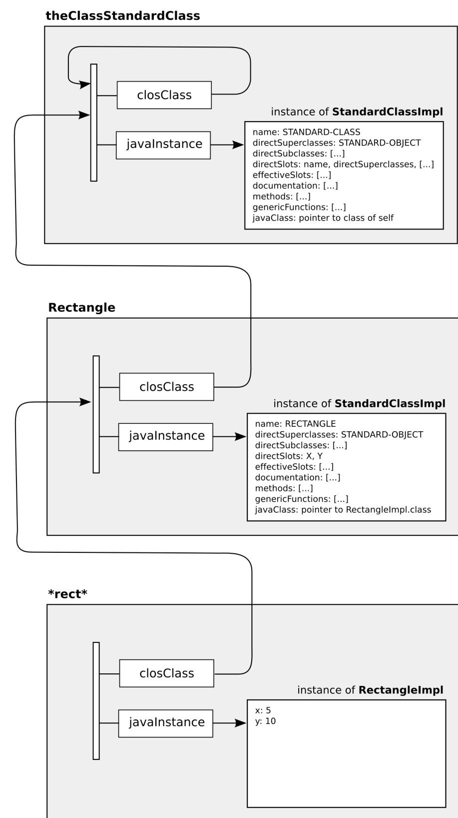

Figure 2. The 3 forms of the backbone.

3.1.2. Structure of CLOS Classes

Figure 2 depicts the relationship between three CLOS instances: **CLOS.theClassStandardClass** (the global and only instance of standard-class named **standard-class**), **Rectangle** (an example of a user defined instance of **standard-class**), and ***rect*** (an instance of **Rectangle**). Note that the Java fields of the **javaInstance** in both **theClassStandardClass** and **Rectangle** include all the traditional slots of standard-class with the addition of a field/slot called **javaClass** which points to a Java class object providing the mechanism for JVM instantiation. In CLforJava, the **JAVA-CLASS** slot is part of the specification for **standard-class**. Together with the **javaInstance** field of **ClosInstance**, **JAVA-CLASS** both makes use of existing JVM machinery for instance creation and instance data storage and also serves as a conceptual bridge from CLOS to the Java programmer.

3.1.3.Instantiating Objects

Instances are created by the Lisp programmer with the make-instance generic function. The Java programmer instantiates CLOS objects in the same way she instantiates other Lisp types such as **Integer**: she accesses the **Factory** class embedded in the Java interface representing the CLOS type. The **ClosInstance** for CLOS class instances are represented by a **ClosInstance** subclass called **InstantiableClosInstance**. This class implements a special interface that provides a predictable **Factory** class for the convenience of the Java programmer.

3.1.4.Constructing Standard Class

CLOS in CLforJava defines a class called **Defclass** which handles the canonicalization tasks identified in the AMOP as well as making it possible to pass around proxy **defclass** forms in Lisp-like ways while deferring the actual task of implementing the CLOS macro expansion layer. The class definition information for **standard-class** arrives in a **Defclass** object, from which are extracted the slot descriptions which are individually fed into **Clos.makeEffectiveSlotDefinition** to produce a collection of **EffectiveSlotDefinition** objects. A new uninitialized **ClosInstance** object is assigned to the static variable **theClassStandardClass**, and **standard-class**'s circular class-of link is established by assigning the **closClass** field of **theClassStandardClass** to itself. As in Closette, the installation of a class association for **standard-class** is required before its slots can be installed with the static method call **Clos.setClassSlots(theClassStandardClass**, **EffectiveSlotDefinition**).

3.1.5.Instantiating Classes

Classes are instantiated by passing a canonicalized **Defclass**, **dc**, to **Clos.EnsureClass** which in turn calls the appropriate instantiation method based on the value of **dc.getMetaclass()**. There is a static method, **Clos.makeInstanceStandardClass()**, used to initialize classes without needing to call CLOS methods for slot access. Its purpose is identical to the similarly named function in Closette, and it only works with classes having the default metaclass **standard-class**.

3.1.6.Meta-Circular Closure

The bootstrapping process guides in the construction of standard-class (see 'Constructing Standard Class,' above), as well as the manual creation of the metaobject **T** (which must not have a superclass) and the processing of the **Defclass** object which defines **standard-object**. With these places filled in the metaobject hierarchy, the meta-circular loop can be closed by passing the **standard-class Defclass** object to **Clos.EnsureClass** and processing it normally. The resulting instance of **standard-class** becomes the permanent value for **theClassStandardClass**, the variable created at the beginning of the bootstrap process.

3.1.7.Change Class and Class Redefinition

CLOS classes are changed or redefined with the appropriate generic functions according the the CLOS specification. The affected **ClosInstance** Java objects are updated by reassigning the value of **javaInstance** (in the case of the **ClosInstance** representing the modified class) and **closClass** (in the case(s) of **ClosInstance** representing the *instances* of the modified class)

3.2. Functions and Methods

Just as CLOS class instances in CLforJava utilize a specialized type of **ClosInstance** called **InstantiableClosInstance**, generic function instances are represented by a different specialization called **FuncallableClosInstance**. This specialized class implements the **Function** interface and allows both Java and Lisp programmers to use **apply** and **funcall** with generic functions identically to calling regular functions.

In a piece of serendipity, the set of Java superinterfaces for a interface is stored in the identical order of CLOS superclasses. Since all CLOS classes have their own interfaces, we can perform the topological sort on the sets of superinterfaces, obviating the need of making our own data structure.[3]

Generic function invocation in CLforJava uses standard **method combination** (currently implemented in the Java backstage). The complete specification for arbitrary method-combination metaobjects remains an area for future research. The **method-combination** metaobject is responsible for computing the effective method resulting from a call to a generic function with specific arguments. Its primary responsibilities are ordering the various applicable methods and correctly accumulating the

[3] This may just the obvious way to store superinterfaces,, but Guy Steele was also one of the designers of Java...

method return values, and as such it must be intimate with method metadata.

4. ACCESSING JAVA FROM CLFORJAVA

Much of the preceding dealt with building the MOP from Java. This section discusses the approach used to make Java accessible from CLOS.

4.1. A New Metaobject Class

Our goal is to give the Lisp programmer access to Java objects and methods in much the same way they access instances and methods in CLOS. While some CL implementations have created syntactic constructs that translate directly to Java, we have taken a different path: a semantic mapping between the MOPs. Simply put, methods are methods, classes are classes, and slots are slots. In CLforJava, Java classes are treated as CLOS classes, having named slots and pre-defined primary methods.

To make this bridge, we have defined a new class metaobject (a subclass of the class metaobject **`Class`**) parallel to the **`StandardClass`** called **`StandardJavaClass`**. Like other class metaobjects, **`StandardJavaClass`** is itself an instance of the class **`StandardClass`**. Instances of the class **`StandardJavaClass`** act to Lisp as proxies for the Java class. They determine their slots via Java introspection. Unlike the creation of standard CLOS classes, **`StandardJavaClass`** also defines generic functions and primary methods for the associated Java class using its public methods. Lisp naming follows the method described in [CLforJava] using Java packages.

To facilitate the use of Java classes, the **`defclass`** macro recognizes the form

```
(defclass java-name () ()
  (:metaclass StandardJavaClass))
```

as a reference to an existing Java class. At the end of this macro evaluation, the CLOS methods: **`slot-value`**, **`direct-slot`**s, **`make-instance`**, and the methods of the Java class are available. This class may also be a superclass of a CLOS s**`tandard class`**.

Note that the evaluation of this **`defclass`** does not create a Java class. It generates a pseudo-class that contains the information to mesh the CLOS and Java class semantics. It is possible to generate instances of this Java class via **`make-instance`** for classes instantiable via the Java **`new`** instruction. For Java classes that are instantiated via a **`Factory`** class, that **`Factory`** class must be defined to CLOS and its **`newInstance`** method (or other as defined in the API) called on the class.

Example

This fragment illustrates the use of the CLforJava MOP. Assume that **`readtable-case`** is **`:preserve`**.

```
(DEFCLASS java.lang:System () ()
  (:METACLASS STANDARD-JAVA-CLASS))
(DEFCLASS java.io:PrintStream () ()
  (: METACLASS STANDARD-JAVA-CLASS))
;; both the System and PrintStream
;; classes are now defined in CLOS
(java.io:println
  (SLOT-VALUE java.lang:System
    'java.lang:out)
  "Hello World")
```

And if we apply the **`use-package`** function to the two Java packages, the form simplifies to

```
(println (SLOT-VALUE System 'out)
  "Hello World")
```

4.2. Behavior of Generic Functions and Methods

All of the generic functions and primary methods generated by **`defclass`** with a metaclass of **`StandardJavaClass`** have a single required argument and an **`&rest`** argument. The required argument is an instance of the specified Java class. Additional arguments are passed directly to the Java instance.

Example

```
;; some of the generic functions
;; defined from the prior example
(DEFGENERIC java.io:println
  ((WRITER java.io:PrintWriter)
    &REST ARGS))
```

Calls to a primary method of a **`StandardJavaClass`** instance are compiled as direct calls to the underlying Java method. Also, the class methods can be used in **`standard method`** combinations such as **`:before`**, **`:after`**, and **`:around`**. The function **`call-next-method`** works correctly in primary and around methods. The final application of **`call-next-method`** can be compiled to a Java method call. Note that an instance of **`StandardJavaClass`** (a Java class) can be the argument to generic functions. The semantics of this form call a static method in the Java class.

Example

```
(defclass counted-system (|System|)
  ((count :type integer
    :allocation :class
    :initform 0)))
;; Define a method that keeps track
;; of the # of times called
(defmethod |println|
  ((sys counted-system) &rest args)
    (incf (count sys))
    (call-next-method))
```

5. CONCLUSIONS AND FUTURES

5.1. Transparency Goal

5.1.1. Wins

Java programmers are able to consistently instantiate CLOS objects in the same way that they instantiate regular Lisp types: by calling make-instance on the embedded **`Factory`** class of the interface. Java programmers also have a consistent method for calling both functions and generic functions. The full power of the Lisp system is accessible to the Java programmer. Slot storage and object instantiation are implemented using existing JVM mechanisms but which do not limit CLOS flexibility and give Java programmers intuitive access to these features.

5.1.2. Losses

CLOS programmers cannot define Java classes and CLOS classes in the same way because of the inability to flexibly compile a Java class separately from its associated methods. For the same reason, a CLOS programmer cannot redefine a Java class the same way that she can redefine a CLOS class.

5.2. Future Research

In the current work, access to the slots of a CLOS instance is entirely based on Lisp functions, i.e. the programmer must locate the SLOT-VALUE function and call its **`funcall`** methods. A more natural access method would be to provide common get/set accessor methods on the implementing class. A direct access might undermine the integrity of a carefully designed set of CLOS classes. Our next experiment is to create Java proxies that can preserve the CLOS semantics.

Also in the current CLOS implementation, the method combinations are hand-crafted in Java. Determining how best to support a move to Lisp-based combinations is high on our list.

6. ACKNOWLEDGMENTS

Our thanks to the Computer Science department of the College of Charleston for their support and encouragement for this project and the broader project that is Common Lisp for Java. Our thanks also to the ILC and the reviewers for their helpful suggestions and the subsequent improvement of this paper.

7. REFERENCES

(1)J. Boetje, “Common Lisp for Java: A New Implementation Intertwined with Java”, Proceedings of the International Lisp Conference, 2005

(2)Kiczales, Gregor, Jim des Rivieres, and Daniel G. Bobrow.”,The Art of the Metaobject Protocol.” MIT Press, 1991.

(3)J.Boetje, “Foundational Actions: Teaching Software Engineering When Time Is Tight”, Proceedings of the Conference on Innovation Technology in Computer Science Education (ITiCSE), 2006