

Intertwining in the Core of CLforJava

Jerry Boetje

College of Charleston

66 George St

Charleston, SC 29424 USA

+1 843-953-6601

boetjeg@cofc.edu

I

ABSTRACT

The original architecture of CLforJava concentrated on the orchestration of the interactions among a set of Java interfaces standing-in as Lisp types and a set of Java classes implementing lambda expressions. As the development evolved, that choice was proven to be strong and surprisingly resilient, leading, for example, to simplified (perhaps novel) implementations for **LOAD-TIME-VALUE**, **DEFSTRUCT** loading and initialization, and **DEFTYPE**. Another simplification of this intertwining enabled the use of standard Java jar files for storage for compiled code and the automatic loading of the contents. In some surprising twists, the use of interfaces to define types turns, at runtime, into a mechanism for controlling functions. Conversely, Java classes become the implementers of Lisp type instances. We expect that this architecture will also simplify the implementation of CLOS in the future.

Type definition generates a corresponding Java interface. Such a definition may be used in the compiler as a Java **instanceof** instruction or as a hint to optimize a computation. A type specification on the other hand is turned into a lambda form usable as a test as needed.

We have noticed a significant improvement in runtime behavior with the latest Java 5/6 versions. A casual user has noted that our use of Java interfaces for function invocation was significantly faster than expected. It also appears that the JVM hotspot compiler is applying a number of optimizations such as tail-call (which our compiler does not yet do) to the byte code.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features – *abstract data types, control structures, classes and objects, data types and structures, patterns.*

General Terms

Algorithms, Design, Experimentation, Languages, Design.

Keywords

Common Lisp, Types, Interfaces, Initialization, Java.

1. INTRODUCTION

CLforJava is a new, from-the-ground-up implementation of the Common Lisp specification. It is developed by CS undergraduates at the College of Charleston for their practical education in software engineering. This paper discusses some of the basic architectural features of CLforJava and how well they have fared over a several-year development.

2. Intertwining Types and Interfaces

When we use the term “intertwining”, we mean we can describe the design and execution of CLforJava equally well from the perspectives of the Java programmer and the Lisp programmer. It is loosely analogous to the wave/particle duality in subatomic physics. Each view is correct, but you can learn different things from each. This section is a quick primer in types and interfaces.

2.1. Atomic Types

For our purposes, we define an atomic type in CLforJava as a named Lisp type that can be defined by a unique Java interface which may also have possibly multiple superinterfaces. The root of the type DAG is called, of course, T. The Lisp type hierarchy is modeled as a DAG of such Java interfaces. It is common for an interface to extend one or other interfaces that define other Lisp types. A simple example is that of the **Vector** interface that extends the **Array** and **Sequence** interfaces.

This simple example demonstrates the intertwining starting from the Java perspective:

```
/** Defines an interface for Vector lisp type. */  
public interface Vector extends Array, Sequence {
```

To demonstrate the intertwining from the Java perspective, we note this line of code from the interface:

```
public static final Symbol typeName =  
    Package.CommonLisp.intern("VECTOR");
```

Java can retrieve the Lisp type name of a Lisp object by accessing its interface and the type name from the interface.

In a subsequent step, the interface object is stored in the Lisp symbol naming the type, enabling the Lisp environment to retrieve the Java type interface.

Examples:

In the **type-of** function, the systems accesses the **typeName** field present in any type interface. This returns the symbol that names the type, **(type-of 42) => FIXNUM**. A Java programmer, given the number 42, can obtain the Java class by use of the **getClass => lisp.common.type.Fixnum**.

The atomic form of **typep** uses Java reflection to determine if it is a member of the type. The system obtains the Java interface from the symbols naming the Lisp object and the Lisp type name. It uses these Java Class method **isAssignableFrom** to determine if one is a subclass of the other.

```
(typep 42 `integer) =>
  Integer.isAssignableFrom(Fixnum)
```

Note: the **lisp.common.type** prefix omitted for readability.

2.2.Type Specifications

Type specifiers in Common Lisp are used for three purposes:

- Optimization in compilation
- Safety testing by interpreters or compilers
- Runtime partitioning

The atomic types, as we have seen, are the easiest to translate into the Java realm. If the expression, **(the fixnum (+ a b))** where **a** and **b** are declared to be fixnums, can be compiled to Java **int** arithmetic. Or, if safety is paramount, each of the components of the expression (and the expression itself) can be checked Java in compiled code by **(the-value-of-a instanceof lisp.common.type.fixnum)**

or by use of the **isAssignableFrom** method in interpreted code.

Handling most of the standard compound types is not much more difficult. For example, if the compiler encounters a type specifier such as **(integer 0 65535)** can be compiled into Java **char** type using unsigned arithmetic. The Java code for safety testing using this specifier on the symbol **a** would be

```
((a-value instanceof Integer) &&
 (a-value >= 0) && (a-value <=65535))
```

If one or more of the specifiers is *****, then to corresponding test is omitted. In compilation without testing, the corresponding value is treated as a **lisp.common.type.Integer**.

Implementing the **and**, **or**, or **not** type specifiers is correspondingly simple in the Java realm when compiling for safety. The compiler makes a conjunction or disjunction of the component types as needed. In principle, we could apply basic logic to optimize the tests, or indeed to know that anything or nothing can be a member of this type.

It is important to remember that all Lisp objects are instances of Java classes that implement one or more interfaces that define atomic types. The type is a conjunction of one or more atomic Lisp types but can be query with a conjunction or disjunction of these types. For example, there may be a **CLOS** object built from the types **GAS-STATION** and **POODLE-BOUTIQUE** where you could fill your tank or get your dog clipped.

The specifier **satisfies** cannot create a Java type that corresponds to it. Given its opaque nature, the specifier cannot be used in optimizations. When CLforJava encounters a **satisfies** specifier, it wraps the entire type specifier into a lambda form that takes an object and tests for its membership.

2.3.The DEFTYPE Macro.

While not yet implemented, this section outlines its operation. Expansion of **DEFTYPE** in CLforJava results in two distinct lambda forms. The first lambda, when applied to the arguments,

will return the declaration form. This form is used by the compiler when it encounters a declaration.

Example using the square-matrix from the HyperSpec:

```
(declare (square-matrix fixnum 100))
```

The expansion results in the core declaration if optimization is on:

```
(and (array fixnum (100 100))
 (satisfies equidimensional))
```

and which the compiler can handle as usual. In this case, it would ignore the **satisfies** clause and use the other information to speed access and arithmetic.

The second **deftype** lambda form is compiled as a normal parameterized function. This function is stored in the symbol naming the **deftype**, in this case **square-matrix**. The function does not override any prior function definition in the symbol. These are separate name spaces. When the enclosing form is compiled, the compiler will not deal directly with the **square-matrix** type. Instead it will insert a call to the **square-matrix** test function when a variable with that declaration is set or bound. This eliminates repetitive insertions of the type checking code.

3.Lambda

Lambdas are the active agents in the system. There are no executable code exposed beyond the boundaries of a lambda form. When the compiler encounters such code, it hastily wraps it in a lambda, compiles it as a new function, and arranges to have it executed at the proper time.

3.1.Basic Structure

Lambda forms are compiled into Java classes that implement the **lisp.common.type.Function** interface. The designation of **Function** means that every such class implements an **apply** method. This method takes a list of arguments, execute the body of the lambda, and returns the result. Multiple values are implemented as a primitive Java array. A simplification comes from the use of various **Function(N)** interfaces. They define the number of arguments the function may and define overloaded **funcall** methods - one for each number of arguments the function takes.

As the type system supports the Java/Lisp type intertwining ethos, so does the function supports intertwining in several ways. Functions are Java classes that implement the **Function** interfaces. This helps slide around some of the Java strong-typing features. They are invoked not by the JVM **invokevirtual** instruction as one might think but by the **invokeinterface** instruction. Lisp function classes can therefore be anything else as long they carry the **Function** interface.

We have had queries over our use of the Java interfaces in this way and its affect on runtime. Since lambdas are never subclassed, we tag these classes as **final**. This enables the JVM to optimize access to the functions.

3.2.LOAD-TIME-VALUE

The implementation of **LOAD-TIME-VALUE** illustrates the Lisp/Java intertwining in functions. The specification mandates that the form be evaluated in the **null** environment and at load time. Using an interaction between the Lisp compiler and the Java class loader, we will demonstrate how the compiler handles the presence of a **LOAD-TIME-VALUE** in a function **BAR** that contains the **LTV** form **(a b 42)**.

1. During the processing of the function **BAR**, the compiler encounters the **LTV**. It is important to remember that the compilation of the **LTV** results in a Java class that implements the **Function** interface.
2. The compiler adds a static, final field (ex **BAR123_field**) to the **BAR** class. Such fields are initialized during class loading and cannot be changed thereafter.
3. Now, the compiler transforms the **LTV** form into a lambda that can create the desired value. In this example, it would be `(lambda () (list 'a 'b 42))`. As per the specification, the compilation is done in the null environment.
4. The transformation is not yet done. The compiler creates another small lambda, **LTV_bridge**, that has but one use. This lambda serves to ensure that the classes are initialized in the proper sequence.

Here is the sequence of events:

- The function **BAR** is loaded, and the Java class executes the class initialization.
- The **BAR** class initialization code makes an instance of the **LTV_bridge**
- Making the instance of **LTV_bridge** causes it to execute its class initialization which in turn initializes the **LTV** lambda. The **LTV_bridge** class initialization finishes by storing the value into a static field.
- Now, the instance of **LTV**, and indeed, the class can be trashed.
- The **LTV_bridge** class is now initialized, and the **BAR** class initialization calls the **LTV_bridge funcall** method that just returns the value in the static field from the prior step.
- The **LTV_bridge** is now complete and can be collected.
- The last step in the **BAR** class initialization is to store the **LTV** value in the static field defined at the beginning of the compilation.
- Further use of the **LTV** value in **BAR** is a one-instruction access, and the use of the Java class initialization ensures the proper order of evaluation.

4.Compile-File

Our reliance on Java conventions and practices led us to a novel (for Lisp systems) approach to the store and access of compiled Lisp code. CLforJava does not create a “standard” **FASL** file, rather it packages the byte code into a Jar file. Use of the Jar format is natural for byte code, resources, and any metadata we care to store. But our method for extracting and loading the byte code is another intertwining between Java and Lisp.

The usual file of Lisp code is a series of lambda forms (after the necessary macro expansions) and other forms among them. For example:

```
(setf (symbol-function 'burp)
      (lambda (x) (blah foo 1 x)))
(print (burp "We're all here"))
(setf (symbol-function 'blah)
      (lambda (y) (burp -1)))
```

Lambda forms are compiled to Java classes as you would expect. However, the compiler also extracts the “other” forms, including placeholder references to the explicit lambdas, into an implicit

lambda form with no parameters. This lambda is of course is compiled into a Java class along with the other lambdas. The placeholder forms are compiled to create an instance of each compiled lambda within the implicit lambda.

Example:

```
(lambda ()
  (setf (symbol-function 'burp)
        (->ref %%lambda-1770))
  (print (burp "We're all here"))
  (setf (symbol-function 'blah)
        (->ref %%lambda-1789)) )
```

Once all of the lambdas are compiled and stored in the Jar file, the file compiler also records the name of the Java class of the implicit lambda into the Jar file’s manifest. The Lisp loading process then becomes very simple. The Lisp loader obtains the class name of the implicit lambda from the manifest, uses the class loader to load the lambda, locates the no-argument constructor, creates an instance, and calls the **funcall** method. This causes the non-lambda code to be executed which loads the other Lisp functions as a side -effect. All of that takes about 8 lines of Java code.

5.DEFSTRUCT

Implementation of the DEFSTRUCT facility was one of the most challenging subsystem in CLforJava. Maintaining our model that Java interfaces define types, interfaces contain factories to create instances, and defstructs may be redefined at runtime was difficult but our basic architecture supported our needs.

According to our architecture, the defstruct facility requires an interface that defines the type and a nested Factory that will create instances of the implementing class. Immediately we were stymied by Java in its inability (or deliberate prevention) of recoding the nested factory class. We were loath to make a special exception from our pattern that worked so well to then.

The signature we wished to preserve was this:

```
myStruct.Factory.newInstance(...args...);
```

where **myStruct** is the name of the type-defining interface. What we needed was a way to maintain an immutable factory in the interface while pulling a Houdini-like transformation without Java noticing.

Our solution was to broaden the definition of nested Factory. The important aspect was the Factory signature. But being unable to replace the nested one, we resorted to indirection. There would be a nested Factory class in the interface, but the real one would live elsewhere. If the struct were redefined, the factory would change but not the type. Here are the pieces:

All structs implement the **Defstruct** interface. This is a superinterface of

```
public interface Defstruct {
    public Object getSlot(Symbol sym); }
```

Now we have to define the abstract factory that is the meta-interface for the factory types.

```
public interface DefstructFactory {
    public Defstruct newInstance(Object... args)
}
```

Here is the interface that provides the `FooStruct.Factory.newInstance` method although it's not really a factory. It's an indirection to a real one that is defined in the `FooStruct` implementation.

```
public interface FooStruct extends Defstruct {
    public static final FooStruct.AbstractFactory factory = new FooStruct.AbstractFactory();
    public static class Factory {
        public static Defstruct newInstance(Object... args) {
            return factory.trueFactory.newInstance(args); }
    static class AbstractFactory {
        DefstructFactory trueFactory = null;
    }
}
```

Here is the implementing class for the instances of the `FooStruct`. The interesting trick is that this class contains the real factory to create instances of that class. And when that class is loaded, all of the loose ends tie up neatly. The slot names are made up for this presentation. Real ones would be substituted at compile time. Note the 2nd and 3rd lines where the class creates an instance of the `Impl.Factory` nested in the `Impl`. The 3rd line sets that factory into the defining interface for `FooStruct`. If the class `FooStructImpl` were ever refined, the interface remains, but it would get a new `Factory` automatically.

```
public class FooStructImpl extends DefstructImpl implements FooStruct {
    public static final FooStructImpl.Factory trueFactory = new FooStructImpl.Factory();
    static { FooStruct.factory.trueFactory = trueFactory; }
    static final Symbol[] slotNames = new Symbol[] {Kludge.symA, Kludge.symB};
    public static class Factory implements DefstructFactory {
        public Defstruct newInstance(Object... args) {
            return new FooStructImpl(args[0], args[1]);
        }
    }
}
```

This has been an interesting stroll some interesting Java tricks, but where does Lisp intertwine in all of this Java? The simple answer is that `defstruct` appeared to be an anomaly in our pattern of object typing and creation. But with a bit of slight-of-class, the pattern is preserved. Although not discussed here, the rest of the `defstruct` code is written in Lisp via all of the intertwining patterns developed in the project.

6. Conclusion

The CLforJava project started with the explicit goal of creating a new version of Common Lisp that runs on the Java Virtual Machine and that intertwines with the Java language. It still follows those goals and shows promise of becoming a useful research tool as continuing to be an excellent teaching vehicle.

7. Acknowledgements

My great thanks to the Computer Science department of the College of Charleston and its chair Chris Starr for encouraging and supporting my efforts in teaching software engineering through the CLforJava project. I am proud for the more than 150 students who have worked so hard over 5 years to make the project a success. My thanks also go particularly to Pascal Constanza and JonL White who have encouraged my efforts.

8. References

- [1] Steele, Guy. *Common Lisp: the Language*, Second Edition. Cambridge, MA, Digital Equipment Corp., 1990
- [2] Boetje, J "Common Lisp for Java: A New Implementation ,Intertwined with Java", Proceedings of the International Lisp Conference, 2005
- [3] Kiczales, Gregor, Jim des Rivieres, and Daniel G. Bobrow. "The Art of the Metaobject Protocol." MIT Press, 1991.
- [4] Boetje, Jerry "Foundational Actions: Teaching Software Engineering When Time Is Tight", Proceedings of the Conference on Innovation Technology in Computer Science Education (ITiCSE), 2006
- [5] Cotton Jay, Boetje Jerry, "A Metaobject Protocol for CLforJava", International Lisp Conference, Cambridge UK, 2007
- [6] Bloch, Joshua "Effective Java, Second Edition", Prentice Hall PTR, 2008