Mixing Lisps in Kawa

Per Bothner
<per@bothner.com>

Abstract

Kawa started as a Scheme implementation written in Java, based on compiling Scheme forms to Java byte-codes. It has developed into a powerful Scheme dialect whose strengths include speed and easy access to Java classes. It is Free Software that some companies depend on.

The Kawa compiler and run-time environment have been generalized to implement other languages besides Scheme, both in the Lisp family (Emacs Lisp, Common Lisp, and BRL), and outside it (XQuery, Nice). This paper focus on the differences and challenges of implementing Common Lisp (not usable yet) and Emacs Lisp, which supports the JEmacs editor.

1 Introduction

Kawa [2] is best known as an implementation of Scheme that compiles to Java byte-codes. However, it has evolved to a framework supporting multiple languages, also including XQuery [4], Emacs Lisp [1], and a start at Common Lisp. (XQuery [5] is an interesting language whose focus is on selecting and generating XML-like tree structures. It is in the process of being standardized by the World Wide Web Foundation.) In this article we will focus on the Lisp family of languages.

Of the Kawa languages, Scheme is the most mature and feature-full. It is being used by various projects and companies. Kawa's core supports a lot Common Lisp features, but very little of Common Lisp's syntax has been implemented. The Emacs Lisp support is intermediate, comprising enough to get some of the basic Emacs functionality working.

Data representation and calling conventions are largely the same for Scheme, Common Lisp, and Emacs Lisp. That is why some of the primitive Emacs Lisp and Common Lisp functions and syntax are currently written in Scheme, just because Scheme is more complete. Adding better support for type declarations and access to Java methods would make it easier to write low-level code in Emacs Lisp and Common Lisp; doing so would not be difficult.

In the following I will touch on some of the interesting challenges of a multi-language Java-based environment, focusing on Emacs Lisp and Common Lisp challenges and their differences from Scheme. In most ways we can view Emacs Lisp is a subset of Common Lisp with a few quirks, plus dynamic (fluid) binding in place of Common Lisp's default lexical binding.

2 Multiple Languages

Traditional "static" compilers that generate machine code often support "front-ends" for more than one language. This is rarer for functional or dynamic languages (ones that support `eval`). One reason is that such implementations are written by small groups that are primarily interested in a single language. Another is that higher-level languages may have more complicated run-time needs, which are harder to generalize to multiple languages. Even related languages like Scheme and Lisp have annoying differences that make life difficult, and we'll discuss some of them in this paper.

So why bother with multiple languages, rather than concentrating on just one? The reason is the same as for a multi-language traditional compiler like Gcc: Different people need or prefer different lan-

guages - and some people need to use multiple languages. Given that a non-trivial compiler and runtime environment is a large undertaking, it makes sense to share some of the code.

It does not follow that Kawa should support mixing multiple languages in the same application, but that too can be useful. Calling a function library written in one language from another language is often useful. A Foreign Function Interface is traditionally used to enable calling functions written in a lower-level language like C, but calling functions written in another high-level language is also sometime useful. One may also want to glue together modules written by different groups that use different languages.

Another case is migrating from one language to another. Most of the Emacs editor is written in Emacs Lisp. The FSF has a long-term goal of replacing Emacs Lisp by Scheme. (Personally, I think a Common Lisp subset might make more sense, as Emacs Lisp is closer to Common Lisp, and there exists packages that add more Common Lisp functionality.) That means there will be a need for mixing Emacs Lisp with Scheme and/or Common Lisp. (The FSF plan is to compile Emacs Lisp to Scheme but that's just an implementation detail.)

Kawa uses a `Language` class that contains various language hooks. For example the read-eval-print loop and the compiler are non-language-specific, but they call methods of the current `Language` instance to perform language-specific actions, such as parsing a source line or file.

3 Execution and compilation

Kawa is compiler-based, for good performance. However, for dynamic languages such as Lisp, it is also important to provide responsive implementations of `eval` and `load`. Kawa implements both an immediate-execution mode (which uses a combination of interpretation and compilation, depending on the input form), and a"batch-compilation" mode (where a module is compiled for future use).

Kawa processes a form or module with these steps:

- The source form is read, creating an S-expression in the usual way. Pairs contain a line/column-number annotation, so we can include source locations in messages, stack traces, and symbol tables.
- The input forms are rewritten to Kawa's internal format, which is a nested tree of `Expression` objects. Rewriting includes resolution of lexical names and macro expansion. Kawa supports both hygienic and non-hygienic macros.
- Some tree-walking passes gather information, perform optimizations, and select lambda representation.
- In immediate mode, if the form is simple enough, we now evaluate it to yield the result value.
- Otherwise, Kawa performs code generation. The top-level `Expression` (specifically an instance of a `ModuleExp`) is expanded to yield one or more Java classes including byte-code instructions.
- In immediate mode, we immediately load the generated classes to create "live" classes in the current Java run-time environment, using Java's `ClassLoader` mechanism. We invoke the `run` method on an instance of the new class. In batch-compile mode the generated classes are written out as files that can be loaded later.

4 Values and Objects

Java is a hybrid class-based object-oriented language. It has "unboxed" (non-heap-allocated) values, such as 32-bit signed integers. However, unboxed values have to be declared at compile-time using "primitive" types. Otherwise, all values are heap-allocated objects that are instances of some class or an array type. All classes and array types inherit from the root `Object` class.

Thus an important task in implementing a Lisp using Java is deciding on how the various Lisp values are represented as Java objects. The following sections discuss how Kawa does so. Kawa can use a standard Java class when that provides functionality close enough to that needed for a Lisp type. Otherwise, Kawa uses its own classes. Most of these are not

Lisp-specific, but can be used by any Kawa language, or directly from Java.

5 Threads and environment

Kawa implements *futures*, which originated in MultiLisp [3]. A future is implemented as a Java thread. Dynamic bindings in a future are shared with those in the parent thread, but within a `fluid-let` we get fresh bindings. Common Lisp stores the value of a dynamic variable in the "value cell" of a symbol. However, the value binding needs to be per-thread, so Kawa symbols don't use a value cell. Instead, the symbol is conceptually used as a key into the current thread's environment. The actual implementation does the name lookup at class loading time, allocating a `Location` object, and then using the Java thread-local mechanism to get the current value.

6 Symbols and Environments

Scheme's symbols are simple: All symbols are interned, and there is only a single unnamed package. Furthermore, there is only a single value binding, rather than separate value and function bindings, and there are no property list cells. Emacs Lisp has only a single package, but symbols have separate value and function bindings, as well a properties.

Kawa supports multiple packages or namespaces. As in Common Lisp, a package is a mapping from a print name to a symbol object. Kawa doesn't yet support the full Common Lisp package functionality, but it implements basic package "inheritance". Kawa symbols are stateless, with just a print name and a pointer to their home package.

As mentioned above, the "value" of a symbol isn't stored in the symbol itself, but it is found indirectly in the current environment, which allows multiple concurrent interpreters, and thread-local bindings. An environment is a two-dimensional mapping that maps a symbol and an arbitrary property object to a location, which may have thread-local bindings (adding a third dimension). To get the value binding of a symbol, look it up in the current environment, using null for the property. To get the function binding of a symbol, use instead for the property the uninterned `FUNCTION` symbol.

Property lists can be accessed via a special `PLIST` property. Alternatively, you can use the property directly, for constant-time access. Combining Common Lisp semantics with constant-time property-list access is a little tricky, but doable. (Early Lisp implementations stored the value and function binding using special properties on the property list, and that's essentially what Kawa does, too - except it uses hashing.)

Common Lisp function names are normally symbols, but can be of the special form: `(setf` *name*`)`. These are easily handled in Kawa by using a special `SETTER` property.

7 Sequences and Arrays

Kawa includes a set of Java classes that implement sequences and arrays. The class hierarchy is compatible with Common Lisp's type hierarchy.

Kawa's generalizes Common Lisp's arrays: An array is an affine mapping onto a sequence, typically a vector. The affine mapping is a linear combination of the array indexes plus a displacement; this generalizes Common Lisp displaced arrays. The vector can be of primitive type, which gives us multiple-dimension arrays of primitive type.

8 Nil

In Scheme the empty list, the symbol `nil`, and the Boolean false value (`#f`) are 3 distinct objects. Common Lisp and Emacs Lisp require that these all be the same object. We don't want to convert lists from one language representation to another when calling across languages, which dictates that we use Scheme's empty list value for `nil`. This means Boolean false differs between the languages, and so `(if x y z)` in Scheme compares `x` against the `Boolean.FALSE` object, and in Lisp it compares `x` against the empty list object. Similarly, the `nil` symbol in the `common-lisp` package is a special case: It's represented by the

empty list object, rather than an instance of the `Symbol` class.

9 Streams

Input and output streams are implemented using Kawa classes that extends the standard Java `Reader` and `PrintWriter` classes. (Scheme uses the term *port* where Common Lisp uses *stream*.) Input streams are implemented using a Kawa class `InPort` that extends the standard Java `Reader` class. Output streams are implemented using a class `OutPort` that extends the standard Java `PrintWriter` class. Common Lisp bidirectional streams aren't currently supported, but would be trivial to add, as they're just a pairing of an input stream and an output stream. An Kawa interactive stream is an input stream that may be tied to an output stream (that is flushed before input), and may have a prompt procedure (whose result is printed at the start of a new line). A Common Lisp interactive stream is slightly different: a bidirectional stream that wraps the input stream and its tied output stream.

A *consumer* is a generalized output stream interface to write arbitrary values, not just characters. Output streams implement the consumer interface by formatting non-character objects. In addition, various other data structures also implement the consumer interface, which is used for a number of purposes, including "writing" multiple value results.

9.1 Readers and read tables

Kawa's Scheme/Lisp reader follows the Common Lisp specification, including using a programmable readtable.

9.2 Printing

Kawa includes a fairly complete implementation of `format` (written in Java). It also includes the pretty-printer from SBCL, translated into Java. (The re-implementation uses arrays rather than lists, and so should be a bit more efficient.) The Lisp programming interface, including the tables for pretty-printing Lisp source code, is mostly missing, but the low-level functionality works quite well. Cycle detection is not implemented yet.

10 Multiple values

Expressions in both Scheme and Common Lisp can return "multiple values". A big difference is that in Common Lisp multiple values can be coerced to a single value. XQuery expressions evaluate to sequences, which is in some ways are similar to multiple values, in that a sequence consisting of a single item is the same as that item. A major difference is that XQuery sequence can be concatenated and can become arbitrarily large, while Lisp expressions can only return a small number of values, explicitly enumerated in the program. Kawa represents XQuery sequences and Lisp multiple values the same way.

Kawa uses two basic representations for multiple values: An explicit representation stores the value in a data structure. The data structure is usually pre-allocated in a per-thread object, reducing the need for memory allocation. Multiple values can also be passed implicitly, as a stream of values. In this model the results of an expression are passed to the current `Consumer` as they are generated. Output streams implement the `Consumer` interface, so the values produced by top-level expression are printed as soon as they are generated. Such stream-based processing is very suitable for XQuery, but I have also experimented with Lisp dialects based on this model.

11 Types

Java has a standard "meta-object protocol" which allows you to query the class of an object and its properties. However, some Kawa languages (especially XQuery) need more extensive type information. Kawa has a separate "type" hierarchy for this reason, and also because a compiler needs to be able to talk about classes that don't yet exist. Kawa has extended Scheme's syntax to allow declaring the types of variables, parameters, and results. In the following example, the parameters `x` and `y` and the result value

are all native (unboxed) 32-bit integers:

```
(define (int-max (x :: <int>) (y :: <int>))
    :: <int>
  (if (> x y) x y))
```

This helps in generating faster code: The above functions compiles to byte-code instructions that operate on 32-bit unboxed Java integers. Kawa automatically converts arguments and results as needed. Type specifiers are also improve compile-time error detection, and makes it very convenient to call Java methods from Scheme.

Kawa doesn't yet support the Common Lisp declaration forms; adding those are probably the biggest priority for Common Lisp and Emacs Lisp, since a type declaration facility is very helpful in writing Lisp code that invokes Java features.

12 Functions

Kawa uses a number of different conventions, optimizations, and tricks for compiling function calls to Java code. When the called function is known, Kawa may emit a direct method invocation, or inline the function's body. The most general mechanism assumes a function is represented by a Java object that implements at least the following two methods:

- The `matchN` method takes an array containing the actual arguments. It returns a negative error code if the arguments have the wrong number or types. Otherwise, the arguments are copied (possibly coerced) to a per-thread parameter-storage-area, and `matchN` returns 0.

- The `apply` method evaluates the function body, using the parameters from the parameter save area. The function's result is "written" to a provided `Consumer`.

This separation handles proper tail-calling, even though Java doesn't. A tail-call evaluates the parameters, and calls `matchN`. If that returns non-zero, an exception is thrown. Otherwise, the function containing the tail-call returns. The `apply` method is called later, after the calling stack frame has been popped.

To call a generic function, we invoke the `matchN` methods of the generic's constituent method functions. If needed, we select the most specific matching method, and call its `apply`.

Kawa supports optional, keyword, and rest parameters, in Scheme as well as Common Lisp and Emacs Lisp.

13 Defining new classes

Kawa Scheme provides a `define-class` form which is similar to that in Stk and Guile, which in turn are derived from Common Lisp's `defclass`. You can use it to define a Java class using Scheme syntax. It supports multiple inheritance fairly efficiently, by making use of Java interfaces.

The `define-simple-class` has the same syntax as `define-class`, but is restricted to single inheritance. This allows a direct translation into a Java class, without needing to define helper interfaces. The result is slightly more efficient, but more importantly make it easier to use the generated classes from Java.

Both forms allow you to define methods belonging to a class, as an alternative to Common Lisp's generic function mechanism, which can also be used.

Some features of CLOS such as `change-class`, may be difficult to implement without adding extra overhead that may be hard to justify.

14 Conditions; continuations

Scheme's `call-with-current-continuation` function can be used to perform general control transfers. Kawa currently only implements limited "exiting" continuation calls, implemented using Java exceptions. General continuations are planned, but not yet implemented. Non-local exits can be implemented using Scheme exceptions, or in the future using continuations. This can be used to implement Common Lisp condition handlers.

15 Modules

Kawa supports separately compiled modules. Normally a source file gets compiled into a "module class" plus sometimes some auxiliary helper classes. Each exported top-level definition gets compiled to a Java field. Importing (requiring) a module works by importing the values bound to the fields. Macros are also compiled into macro objects. Macro "hygiene" works across modules: an exported macro may expand to a form that references an non-exported definition.

16 Emacs types

The core of the Emacs Lisp language is one of many dynamically scoped Lisp extension languages. What makes it interesting is its embedding in Emacs, and the special data types used by Emacs. These include buffers, windows, frames, and key-maps. Kawa includes basic implementations of classes for these Emacs values, written from scratch in Java. Actually, currently there are two implementations of some of these classes. The initial implementation used the standard Swing toolkit. Recently, Christian Surlykke has contributed support for the SWT toolkit (from the Eclipse IDE), and we've made the JEmacs core classes platform-independent.

17 Editing and debugging

Kawa emits standard Java debug information, including line numbers and local variable names. Thus Java stack traces contain line numbers referencing the Kawa input file. It is also possible to debug Kawa programs (at least Scheme) using an IDE like Eclipse. The latter is helped by an Eclipse plugin written by Dominque Boucher, which includes a nice Scheme/Lisp editor with support for Kawa extensions. The result is the beginnings of a Scheme debugger, but it isn't terribly friendly yet. One issue is that Scheme/Lisp symbol names need to be "mangled" (translated) into valid Java names. (This is unfortunately required by the Java Virtual Machine, for no good reason I know of.) The IDE doesn't have support for producing the reverse mapping. Printing Lisp values is less then ideal, though tolerable. There is no way to input Scheme/Lisp expressions, for example in conditional breakpoint predicates. The IDE knows nothing about how closures and lambdas are translated in Kawa classes, which means the programmer has to know this instead. Still, this is a good step towards good Scheme/Lisp support in one of the world's most popular IDEs.

18 Summary

Kawa is a full-featured and mature environment for compiling and running high-level languages on the Java platform. The Scheme implementation is the more popular and complete, but other languages are also being implemented. Kawa is especially convenient for efficient implementations of Lisp variants. My time to devote on Emacs Lisp and Common Lisp has been limited; collaborators will be very welcome.

References

[1] Per Bothner. *JEmacs - The Java/Scheme-based Emacs* Free Software Magazine (original incarnation). 2002. (`http://per.bothner.com/papers/JEmacs02`).

[2] Per Bothner. *Kawa: Compiling Scheme to Java* Lisp Users Conference (Berkeley). 1998. (`http://www.gnu.org/software/kawa`).

[3] Robert Halstead. *MultiLisp: A Language for Concurrent Symbolic Computation* TOPLAS 7(4):501-538. 1985.

[4] Per Bothner. *Compiling XQuery to Java bytecodes* First International Workshop on XQuery Implementation Experience and Perspectives (XIME-P). 2004. (`http://per.bothner.com/papers/Qexo04`).

[5] *XQuery 1.0: An XML Query Language* (`http://www.w3c.org/XML/Query`).