Kawa - Compiling Dynamic Languages to the Java VM

Per Bothner
Cygnus Solutions
1325 Chesapeake Terrace
Sunnyvale CA 94089, USA
<`bothner@cygnus.com`>

Abstract:

Many are interested in Java for its portable bytecodes and extensive libraries, but prefer a different language, especially for scripting. People have implemented other languages using an interpreter (which is slow), or by translating into Java source (with poor responsiveness for `eval`). Kawa uses an interpreter only for "simple" expressions; all non-trivial expressions (such as function definitions) are compiled into Java bytecodes, which are emitted into an in-memory byte array. This can be saved for later, or quickly loaded using the Java `ClassLoader`.

Kawa is intended to be a framework that supports multiple source languages. Currently, it only supports Scheme, which is a lexically-scoped language in the Lisp family. The Kawa dialect of Scheme implements almost all of the current Scheme standard (R^{5}RS), with a number of extensions, and is written in a efficient object-oriented style. It includes the full "numeric tower", with complex numbers, exact infinite-precision rational arithmetic, and units. A number of extensions provide access to Java primitives, and some Java methods provide convenient access to Scheme. Since all Java objects are Scheme values and vice versa, this makes for a very powerful hybrid Java/Scheme environment.

An implementation of ECMAScript (the standardized "core" of JavaScript) is under construction. Other languages, including Emacs Lisp, are also being considered.

Kawa home page: `http://www.cygnus.com/~bothner/kawa.html`.

1. Introduction

While Java is a decent programming language, the reason for the "Java explosion" is largely due to the Java Virtual Machine (JVM), which allows programs to be distributed easily and efficiently in the form of portable bytecodes, which can run on a wide variety of architectures and in web browsers. These advantages are largely independent of the Java language, which is why there have been a number of efforts to run other languages on the JVM, even though the JVM is very clearly designed and optimized for Java. Many are especially interested in more high-level, dynamic "scripting" languages to use in a project in conjunction with Java. A language implemented on top of Java gives programmers many of the extra-linguistic benefits of Java, including libraries, portable bytecodes, web applets, and the existing efforts to improve Java implementations and tools.

The Kawa toolkit supports compiling and running various languages on the Java Virtual Machine. Currently, Scheme is fully supported (except for a few difficult features discussed later). An implementation of ECMAScript is coming along, but at the time of writing it is not usable.

Scheme [R^{5}RS] is a simple yet powerful language. It is a non-pure functional language (*i.e.* it has first-class functions, lexical scoping, non-lazy evaluation, and side effects). It has dynamic typing, and usually has an interactive read-evaluate-print interface. The dynamic nature of Scheme (run-time typing, immediate expression evaluation) may be a better match for the dynamic Java environment (interpreted bytecodes, dynamic loading) than Java is!

ECMAScript is the name of the dialect of JavaScript defined by ECMA standard 262 [ECMAScript], which standardizes JavaScript's core language only, with no input/output or browser/document interface. It defines a very dynamic object-based language based on prototype inheritance, rather than classes. A number of new or proposed Web standards are based on ECMAScript.

Information and source code will be available from `http://www.cygnus.com/~bothner/kawa.html`. (Note that Kawa is also the name of an unrelated commercial Java development environment.)

2. History

Starting in 1995 Cygnus (on behalf of the Free Software Foundation) developed Guile, an implementation of Scheme suitable as a general embedding and extension language. Guile was based on Aubrey Jaffar's SCM interpreter; the various Guile enhancements were initially done by Tom Lord. In 1995 we got a major contract to enhance Guile, and with our client we added more features, including threads (primarily done by Anthony Green), and internationalization.

The contract called for a byte-code compiler for Guile, and it looked like doing a good job on this would be a major project. One option we considered was compiling Scheme into Java bytecodes and executing them by a Java engine. The disadvantage would be that such a Scheme system would not co-exist with Guile (on the other hand, we had run into various technical and non-technical problems with Guile that led us to conclude that Guile would after all not be strategic to Cygnus). The advantage of a Java solution was leveraging off the tools and development being done in the Java "space", plus that Java was more likely to be strategic long-term.

The customer agreed to using Java, and I started active development June 1996. As a base, I used the Kawa Scheme interpreter written by R. Alexander Milowski. He needed an object-oriented Scheme interpreter to implement DSSSL [DSSSL], a Scheme-like environment for expressing style, formatting, and other processing of SGML [SGML] documents. DSSSL is an subset of "pure" Scheme with some extensions. Kawa 0.2 was a simple interpreter which was far from complete. It provided a useful starting point, but almost all of the original code has by now been re-written.

Kawa 1.0 was released to our customer and "the Net" September 1996. Development has continued since then, at a less intense pace! The long-term goal is an object-oriented environment that harmoniously integrates Scheme, Java, EcmaScript, and other languages.

3. Basic implementation strategy

There are three basic ways one might implement some programming language X using Java:

- One could write an interpreter for X in Java. First parse the source into an internal "abstract syntax tree", and then evaluate it using a recursive `eval` function. The advantage of using Java rather than C or C++ is having garbage collection, classes, and the standard Java class library makes it easier.

  The obvious down-side of the interpreter solution is speed. If your interpreter for language X is written in Java, which is in turn interpreted by a Java VM, then you get double interpretation overhead.

- You could write a compiler to translate language X into Java source code. You need to define a mapping for language X constructs into equivalent Java constructs, and then write a program that writes out a parsed X program using corresponding Java constructs. A number of implementations take this approach, including NetRexx, and various "extended Java" dialects.

  This only gives you the single a single (Java VM) layer of interpretation. On the other hand, most of the efforts that people are making into improving Java performance will benefit your implementation, since you use standard Java bytecodes.

  The biggest problem with that approach is that it is an inherently batch process, and has poor responsiveness. Consider a read-eval-print-loop, that is the ability for a user to type in an expression, and have it be immediately read, evaluated, and the result printed. If evaluating an expression requires converting it to a Java program, writing it to a disk file, invoking a separate java compiler, and then loading the resulting class file into the running environment, then response time will be inherently poor. This hurts "exploratory programming", that is the ability to define and update functions on the fly.

  A lesser disadvantage is that Java source code is not quite as expressive as Java bytecodes. While bytecodes are very close to Java source, there are some useful features not available in the Java language, such as goto. Debugging information is also an issue.

- Alternatively, you could directly generate Java bytecode. You can write out a .class file, which can be saved for later. You also have the option of writing to an internal byte array, which can be immediately loaded as a class using the `java.lang.ClassLoader.defineClass` method. In that case you can by-pass the file system entirely, yield a fast load-and-go solution, which enables a very responsive read-eval-print loop.

  This solution is the best of both worlds. The main problem is that more code needs to be written. Fortunately, by using Kawa, much that work has already been done.

I will discuss the compiler later, but first we will give an overview of the run-time environment of Kawa, and the classes used to implement Scheme values.

4. Objects and Values

Java [JavaSpec] has primitive types (such as 32-bit `int`) as well reference types. If a variable has a reference type, it means that it can contains references (essentially pointers) to objects of a class, or it can contain references to objects of classes that "extend" (inherit from) the named class. The inheritance graph is "rooted" (like Smalltalk and unlike C++); this means that all classes inherit from a distinguished class `java.lang.Object` (or just `Object` for short).

Standard Scheme [R^{5}RS] has a fixed set of types, with no way of creating new types. It has run-time typing, which means that types are not declared, and a variable can contain values of different types at different times. The most natural type of a Java variable that can contain any Scheme value is therefore `Object`, and all Scheme values must be implemented using some class that inherits from `Object`.

The task then is to map each Scheme type into a Java class. Whether to use a standard Java class, or to write our own is a tradeoff. Using standard Java classes simplifies the passing of values between Scheme functions and existing Java methods. On the other hand, even when Java has suitable built-in classes, they usually lack functionality needed for Scheme, or are not organized in any kind of class hierarchy as in Smalltalk or Dylan. Since Java lacks standard classes corresponding to pairs, symbols, or procedures, we *have* to write some new classes, so we might as well write new classes whenever the existing classes lack functionality.

The Scheme boolean type is one where we use a standard Java type, in this case `Boolean` (strictly speaking `java.lang.Boolean`). The Scheme constants `#f` and `#t` are mapped into static fields (*i.e.* constants) `Boolean.FALSE` and `Boolean.TRUE`.

On the other hand, numbers and collections are reasonably organized into class hierarchies, which Java does not do well. So Kawa has its own classes for those, discussed in the following sections.

4.1. Collections

Kawa has a rudimentary hierarchy of collection classes.

```
class Sequence
{ ...;
  abstract public int length();
  abstract public Object elementAt(int i);
}
```

A `Sequence` is the abstract class that includes lists, vectors, and strings.

```
class FString extends Sequence
{ ...;
  char[] value;
}
```

Used to implement fixed-length mutable strings (array of Unicode character). This is used to represent Scheme strings.

```
class FVector extends Sequence
{ ...;
  Object[] value;
}
```

Used to implement fixed-length mutable general one-dimensional array of `Object`. This is used to represent Scheme vectors.

```
public class List extends Sequencw
{ ...;
  protected List () { }
  static public List Empty = new List ();
}
```

Used to represent Scheme (linked) lists. The empty list `'()` is the static (global) value `List.Empty`. Non-empty-lists are implemented using `Pair` objects.

```
public class Pair extends Sequence
{ ...;
  public Object car;
  public Object cdr;
}
```

Used for Scheme pairs.

```
public class PairWithPosition extends Pair
{ ...;
}
```

Like `Pair`, but includes the filename and linenumber in the file from which the pair was read.

Future plans include more interesting collection classes, such a sequences implemented as a seekable disk file; lazily evaluated sequences; hash tables; APL-style multi-dimensional arrays; stretchy buffers. (Many of these ideas were implemented in my earlier experimental language `Q` – see [Bothner88] and `ftp://ftp.cygnus.com/pub/bothner/Q/`. I will also integrate the Kawa collections into the new JDK 1.2 collections framework.

4.2. Top-level environments

```
class Environment
{ ...;
}
```

An `Environment` is a mapping from symbols to bindings. It contains the bindings of the user top-level. There can be multiple top-level `Environments`, and an `Environment` can be defined as an extension of an existing `Environment`. The latter feature is used to implement the various standard environment arguments that can be passed to `eval`, as adopted for the latest Scheme standard revision ("R^{5}RS"). Nested environments were also implemented to support threads, and fluid bindings (even in the presence of threads).

Environments will be combined into a more general name-table interface, which will also include records and ECMAScript objects.

4.3. Symbols

Symbols represent identifiers, and do not need much functionality. Scheme needs to be able to convert them to and from Scheme strings, and they need to be "interned" (which means that there is a global table to ensure that there is a unique symbol for a given identifier). Symbols are immutable and have no accessible internal structure.

Scheme symbols are reprented using interned Java `String`s. Note that the Java `String` class implements immutable strings, and is therefore cannot be used to implement Scheme strings. However, it makes sense to use it to implement symbols, since the way Scheme symbols are used is very similar to how Java `String`s are used. The method `intern` in `String` provides an interned version of a `String`, which provides the characters-to-`String` mapping needed for Scheme strings.

5. Numbers

Scheme defines a "numerical tower" of numerical types: number, complex, real, rational, and integer. Kawa implements the full "tower" of Scheme number types, which are all sub-classes of the abstract class `Quantity` discussed later.

```
public class Complex extends Quantity
{ ...;
  public abstract RealNum re();
  public abstract RealNum im();
}
```

`Complex` is the class of abstract complex numbers. It has three subclasses: the abstract class `RealNum` of real numbers; the general class `CComplex` where the components are arbitrary `RealNum` fields; and the optimized `DComplex` where the components are represented by `double` fields.

```
public class RealNum extends Complex
{ ...;
  public final RealNum re()
  { return this; }
  public final RealNum im()
  { return IntNum.zero(); }
  public abstract boolean isNegative();
}
```

```
public class DFloNum extends RealNum
{ ...;
  double value;
}
```

Concrete class for double-precision (64-bit) floating-point real numbers.

```
public class RatNum extends RealNum
{ ...;
  public abstract IntNum numerator();
  public abstract IntNum denominator();
}
```

`RatNum`, the abstract class for exact rational numbers, has two sub-classes: `IntFraction` and `IntNum`.

```
public class IntFraction extends RatNum
{ ...;
  IntNum num;
  IntNum den;
}
```

The `IntFraction` class implements fractions in the obvious way. Exact real infinities are identified with the fractions `1/0` and `-1/0`.

```
public class IntNum extends RatNum
{ ...;
  int ival;
  int[] words;
}
```

The `IntNum` concrete class implements infinite-precision integers. The value is stored in the first `ival` elements of `words`, in 2's complement form (with the low-order bits in `word[0]`).

There are already many bignum packages, including one that Sun added for JDK 1.1. What are the advantages of this one?

- A complete set of operations, including gcd and lcm; logical, bit, and shift operations; power by repeated squaring; all of the division modes from Common Lisp (floor, ceiling, truncate, and round); and exact conversion to `double`.
- Consistency and integration with a complete "numerical tower." Specifically, consistency and integration with "fixnum" (see below).
- Most bignum packages use a signed-magnitude representation, while Kawa uses 2's complement. This makes for easier integration with fixnums, and also

makes it cheap to implement logical and bit-fiddling operations.

- Use of all 32 bits of each "big-digit" word, which is the "expected" space-efficient representation. More importantly, it is compatible with the mpn routines from the Gnu Multi-Precision library (gmp) [gmp]. The mpn routines are low-level algorithms that work on unsigned pre-allocated bignums; they have been transcribed into Java in the MPN class. If better efficiency is desired, it is straight-forward to replace the MPN methods with native ones that call the highly-optimized mpn functions.

If the integer value fits within a signed 32-bit int, then it is stored in ival and words is null. This avoids the need for extra memory allocation for the words array, and also allows us to special-case the common case.

As a further optimization, the integers in the range -100 to 1024 are pre-allocated.

5.1. Mixed-type arithmetic

Many operations are overloaded to have different definitions depending on the argument types. The classic examples are the functions of arithmetic such as "+", which needs to use different algorithms depending on the argument types. If there is a fixed and reasonably small set of number types (as is the case with standard Scheme), then we can just enumerate each possibility. However, the Kawa system is meant to be more extensible and support adding new number types.

The solution is straight-forward in the case of a one-operand function such as "negate", since we can use method overriding and virtual method calls to dynamically select the correct method. However, it is more difficult in the case of a binary method like "+," since classic object-oriented languages (including Java) only support dynamic method selection using the type of the first argument ("this"). Common Lisp and some Scheme dialects support dynamic method selection using all the arguments, and in fact the problem of binary arithmetic operations is probably the most obvious example where "multi-dispatch" is useful.

Since Java does not have multi-dispatch, we have to solve the problem in other ways. Smalltalk has the same problems, and solved it using "coercive generality": Each number class has a generality number, and operands of lower generality are converted to the class with the higher generality. This is inefficient because of all the conversions and temporary objects (see [Budd91Arith]), and it is limited to what extent you can add new kinds of number types.

In "double dispatch" [Ingalls86] the expression x-y is implemented as x.sub(y). Assuming the (run-time) class of x is Tx and that of y is Ty, this causes the sub method defined in Tx to be invoked, which just does y.subTx(x). That invokes the subTx method defined in Ty which can without further testing do the subtraction for types Tx and Ty.

The problem with this approach is that it is difficult to add a new Tz class, since you have to also add subTz methods in all the existing number classes, not to mention addTz and all the other operations.

In Kawa, x-y is also implemented by x.sub(y). The sub method of Tx checks if Ty is one of the types it knows how to handle. If so, it does the subtraction and returns the result itself. Otherwise, Tx.sub does y.subReversed(x). This invokes Ty.subReversed (or subReversed as defined in a super-class of Ty). Now Ty (or one of its super-classes) gets a chance to see if it knows how to subtract itself from a Tx object.

The advantage of this scheme is flexibility. The knowledge of how to handle a binary operation for types Tx and Ty can be in either of Tx or Ty or either of their super-classes. This makes is easier to add new classes without having to modify existing ones.

5.2. Quantities

The DSSSL language [DSSSL] is a dialect of Scheme used to process SGML documents. DSSSL has "quantities" in addition to real and integer numbers. Since DSSSL is used to format documents, it provides length values that are a multiple of a meter (*e.g.* 0.2m), as well as derived units like cm and pt (point). A DSSSL quantity is a product of a dimension-less number with an integral power of a length unit (the meter). A (pure) number is a quantity where the length power is zero.

For Kawa, I wanted to merge the Scheme number types with the DSSSL number types, and also generalize the DSSSL quantities to support other dimensions (such as mass and time) and units (such as kg and seconds). Quantities are implemented by the abstract class Quantity. A *quantity* is a product of a Unit and a pure number. The number can be an arbitrary complex number.

```
public class Quantity extends Number
{ ...;
  public Unit unit()
  { return Unit.Empty; }
  public abstract Complex number();
}
```

```
public class CQuantity extends Quantity
{ ...;
  Complex num;
  Unit unt;
  public Complex number()
  { return num; }
  public Unit unit()
  { return unt; }
}
```

A `CQuantity` is a concrete class that implements general `Quantities`. But usually we don't need that much generality, and instead use `DQuanity`.

```
public class DQuantity extends Quantity
{ ...;
  double factor;
  Unit unt;
  public final Unit unit()
  { return unt; }
  public final Complex number()
  { return new DFloNum(factor); }
}
```

```
public class Unit extends Quantity
{ ...;
  String name; // Optional.
  Dimensions dims;
  double factor;
}
```

A `Unit` is a product of a floating-point `factor` and one or more primitive units, combined into a `Dimensions` object. The `Unit` name have a name (such as "`kg`"), which is used for printing, and when parsing literals.

```
public class BaseUnit extends Unit
{ ...;
  int index;
}
```

A `BaseUnit` is a primitive unit that is not defined in terms of any other `Unit`, for example the meter. Each `BaseUnit` has a different `index`, which is used for identification and comparison purposes. Two `BaseUnits` have the same `index` if and only if they are the same `BaseUnit`.

```
public class Dimensions
{
  BaseUnit[] bases;
  short[] powers;
}
```

A `Dimensions` object is a product and/or ratio of `BaseUnits`. You can think of it as a data structure that maps every `BaseUnit` to an integer power. The `bases` array is a list of the `BaseUnits` that have a non-zero power, in order of the `index` of the `BaseUnit`. The `powers` array gives the power (exponent) of the `BaseUnit` that has the same index in the `bases` array.

Two `Dimensions` objects are equal if they have the same list of `bases` and `powers`. `Dimensions` objects are "interned" (using a global hash table) so that they are equal only if they are the same object. This makes it easy to implement addition and subtraction:

```
public static DQuantity add
  (DQuantity x, DQuantity y)
{
  if (x.unit().dims != y.unit().dims)
     throw new ArithmeticException
       ("units mis-match");
  double r = y.unit().factor
     / x.unit().factor;
  double s = x.factor + r * y.factor;
  return new DQuantity (s, x.unit());
}
```

The `Unit` of the result of an addition or subtraction is the `Unit` of the first operand. This makes it easy to convert units:

```
(+ 0cm 2.5m)    ==>    250cm
```

Because Kawa represents quantities relative to user-specified units, instead of representing them relative to primitive base units, it can automatically print quantities using the user's preferred units. However, this does make multiplication and division more difficult. The actual calculation (finding the right `Dimensions` and multiplying the constant factors) is straight-forward. The problem is generating the new compound unit, and later printing out the result in a human-friendly format. There is no obvious right way to do this. Kawa creates a `MulUnit` to represent a compound unit, but it is not obvious which simplifications should be done when. Kawa uses a few heuristics to simplify compound units, but this is an area that could be improved.

6. Procedures

Scheme has procedures as first-class values. Java does not. However, we can simulate procedure values, by overriding of virtual methods.

```
class Procedure
{ ...;
  public abstract Object applyN
    (Object[] args);
  public abstract Object apply0();
  ...;
  public abstract Object apply4
    (Object arg1, ..., Object arg4);
}
```

We represent Scheme procedures using sub-classes of the abstract class `Procedure`. To call (apply) a procedure with no arguments, you invoke its `apply0` method; to invoke a procedure, passing it a single argument, you use its `apply1` method; and so on using `apply4` if you have 4 arguments. Alternatively, you can bundle up all the arguments into an array, and use the `applyN` method. If you have more than 4 arguments, you have to use `applyN`.

Notice that all `Procedure` sub-classes have to implement all 6 methods, at least to the extent of throwing an exception if it is passed the wrong number of arguments. However, there are utility classes `Procedure0` to `Procedure4` and `ProcedureN`:

```
class Procedure1 extends Procedure
{
  public Object applyN(Object[] args)
  {
    if (args.length != 1)
        throw new WrongArguments();
    return apply1(args[0]);
  }
  public Object apply0()
  { throw new WrongArguments();}
  public abstract Object apply1
    (Object arg1);
  public Object apply2
    (Object arg1, Object arg2)
  { throw new WrongArguments();}
  ...;
}
```

Primitive procedures are generally written in Java as sub-classes of these helper classes. For example:

```
class car extends Procedure1
{ // Return first element of list.
  public Object apply1(Object arg1)
    { return ((Pair) arg1).car; }
}
```

A user-defined Scheme procedure is compiled to a class that is descended from `Procedure`. For example, a variable-argument procedure is implemented as a sub-class of `ProcedureN`, with an `applyN` method comprising the bytecode compiled from the Scheme procedure body. Thus primitive and user-defined procedure have the same calling convention.

If a nested procedure references a lexical variable in an outer procedure, the inner procedure is implemented by a "closure". Kawa implements a closure as a `Procedure` object with a "static link" field that points to the inherited environment. In that case the lexical variable must be heap allocated, but otherwise lexical variables use local Java variable slots. (This is conceptually similar to the "Inner classes" added in JDK 1.1.)

```
class ModuleBody extends Procedure0
{ ...;
  public Object apply0()
  { return run(Environment.current());}
  public abstract Object run
    (Environment env);
}
```

Top-level forms (including top-level definitions) are treated as if they were nested inside a dummy procedure. A `ModuleBody` is such a dummy procedure. When a file is `loaded`, the result is a `ModuleBody`; invoking `run` causes the top-level actions to be executed.

7. Overview of compilation

These are the stages of compilation:

Reading

The first compilation stage reads the input from a file, from a string, or from the interactive command interpreter. The result is one or more Scheme forms (S-expressions), usually lists. If reading commands interactively, only a single form is read; if reading from a file or string, all the forms are read until end-of-file or end-of-string; in either case, the result is treated as the body of a dummy function (*i.e.* a `ModuleBody`).

Semantic analysis

The source form is rewritten into an `Expression` object, specifically a `ModuleExp`. This stage handles macro expansion and lexical name binding. Many optimizations can be done in this phase by annotating and re-arranging `Expressions`.

Code generation

The resulting `ModuleExp` is compiled into one or more byte-coded classes. This is done by invoking the virtual `compile` method recursively on the `Expressions`, which generates instructions (using the `bytecode` package) to evaluate the expression and leave the result on the Java operand stack. At the end we ask the `bytecode` package to write out the resulting classes and methods. They can be written to a file (for future use), or into byte arrays in memory.

Loading

The compiled bytecodes are loaded into the Kawa run-time. In the case of code that is compiled and then immediately executed, the compiled code can be immediately turned into Java classes using the Java `ClassLoader` feature. (That is how the read-eval-print loop works.) An instance of the compiled sub-class of `ModuleBody` is created and `run`, which normally produces various side-effects.

8. Expressions

The abstract `Expression` class represents partially processed expressions. These are in principle independent of the source language, though there are still some Scheme assumptions wired in.

```
class Expression
{ ...;
  public abstract Object eval
    (Environment e);
  public abstract void compile
    (Compilation comp, Target targ);
}
```

The `eval` method evaluates the `Expression` in the given `Environment`. The `compile` method is called when we are compiling the body of a procedure. It is responsible for generating bytecodes that evaluate the expression, and leave the result in a result specified by the `Target` parameter. This is usually the Java evaluation stack, but we will go into more detail later.

```
class QuoteExp extends Expression
{ ...;
  Object value;
  public QuoteExp(Object val)
  { value = val; }
  public Object eval(Environment env)
  { return value; }
  public void compile
    (Compilation comp, Target target)
  { comp.compileConstant (value, target); }
}
```

A `QuoteExp` represents a literal (self-evaluating form), or a quoted form.

```
class ReferenceExp extends Expression
{ ...;
  Symbol symbol;
  Declaration binding;
}
```

A `ReferenceExp` is a reference to a named variable. The `symbol` is the source form identifier. If `binding` is non-null, it is the lexical binding of the identifier.

```
class ApplyExp extends Expression
{ ...;
  Expression func;
  Expression[] args;
}
```

An `ApplyExp` is an application of a procedure `func` to an argument list `args`.

```
class ScopeExp extends Expression
{ ...;
  ScopeExp outer;  // Surrounding scope.
  public Declaration add_decl(Symbol name)
  { ...Create new local variable... }
}
```

A `ScopeExp` is a abstract class that represents a lexical scoping construct. Concrete sub-classes are `LetExp` (used for a `let` binding form) and `LambdaExp`.

```
class LambdaExp extends ScopeExp
{ ...;
  Symbol name; // Optional.
  Expression body;
  int min_args;
  int max_args;
}
```

The Scheme primitive syntax `lambda` is translated into a `LambdaExp`, which represents anonymous procedures. Each `LambdaExp` is compiled into a different bytecoded class. Invoking `eval` causes the `LambdaExp` to be compiled into a class, the class to be loaded, an instance of the class to be created, and the result coerced to a `Procedure`.

Other sub-classes of `Expression` are `IfExp` (used for conditional expressions); `BeginExp` (used for compound expressions); `SetExp` (used for assignments); and `ErrorExp` (used where a syntax error was found);

9. Semantic analysis

The translation phase takes a top-level form (or body), and generates a `ModuleExp`, which is a top-level expression. This is done using a `Translator`, which keeps track of lexical bindings and other translation state.

```
class Translator
```

```
{ ...;
  public Expression rewrite(Object exp)
  { ... }
  public Expression syntaxError
   (String message) { ... }
}
```

The `rewrite` method converts a Scheme source form to an `Expression`. The `syntaxError` method is called when a syntax error is seen. It prints out the current source filename and line number with the given `message`.

9.1. Syntax and Macros

```
class Syntax
{ ...;
  public abstract Expression rewrite
    (Object obj, Translator tr);
}
```

The `rewrite` method in `Translator` checks for syntactic keywords and macros. If the `car` of a "call" is a `Syntax` or if it is a `Symbol` that is bound to a `Syntax`, then its `rewrite` method is called.

As an example, this trivial class implements `quote`:

```
class quote extends Syntax
{ ...;
  public Expression rewrite
    (Object obj, Translator tr)
  { // Error-checking is left out.
    return new QuoteExp(((Pair)obj).car);
  }
}
```

Much more complicated is the `Syntax` that implements `define-syntax`.

```
class define_syntax extends Syntax
{ ...;
  public Expression rewrite
    (Object obj, Translator tr)
  {  enter (new SyntaxRules (...)); }
}
```

The result is a `SyntaxRules` object, which contains an encoded representation of the patterns and templates in the `syntax-rules`. This is in its own right a `Syntax` object.

```
class SyntaxRules extends Syntax
{ ...;
  SyntaxRule[] rules;
  public Expression rewrite
    (Object obj, Translator tr)
  {
    Object[] v = new Object[maxVars];
    for (int i = 0;  i < rules.length;)
    {
      SyntaxRule r = rules[i++];
      if (r.match (obj, v))
        return r.execute_template(v, tr);
    }
    return tr.syntaxError
      ("no matching syntax-rule");
  }
}
```

Contrast evaluating a procedure definition (`lambda`), which causes a new *sub-class* of `Procedure` to be created and compiled, while evaluating a `define-syntax` only causes a new *instance* of `SyntaxRules` to be created.

10. Interpretation: Eval

Many people think of Scheme, Lisp, and ECMAScript as "interpreted" languages. However, many of these languages have compilers. What these languages do have is `eval` - that is a command that at run-time takes a source program, and evaluates it. They may also have an interactive read-eval-print interface. For such uses a traditional interpreter is easiest and most responsive. Therefore, high-end Lisp systems traditionally provide both a compiler and an interpreter. Such duplication is expensive, in terms of size, development effort, and testing. If one has load-and-go capabilities, that is the ability to efficiently load a compiled program into a running application, then one can simply implement `eval` as a compile followed by a load.

When we compile to Java bytecodes, we create one or more files in the `.class` format. There is a standard Java method `java.lang.ClassLoader.defineClass` that takes a byte array laid out in the format of a `.class`, and from it dynamically creates a new class in the existing Java run-time. (This facility is used for "applets" downloaded accross the Network.) Kawa uses this scheme to implement `eval`, and it works well. Because `ClassLoader.defineClass` takes an array, rather than a file, we can compile and load entirely inside the Kawa run-time, without having to go via the filesystem for temporary files, as a traditional compiler batch does. The result is near-instant response.

There is a tradeoff, though. Doing a compile+load is a very heavy-duty operation, compared to a simply interpreting an expression. It creates a lot of temporary objects. Worse, it also creates some temporary classes, and many Java environments do not garbage collect unused classes.

Kawa uses a compromise strategy. If the `Expression` is “simple”, it is interpreted directly, using the `Expression.eval`. Otherwise, it is compiled. Simple expressions include literals, (global) variable access, assignment, and function application. Implementing `eval` in those cases is trivial. Expressions that define new local bindings (such lambda expressions and `let` forms) do not implement `eval`. If the user types in such an expression, it is wrapped inside a dummy function, compiled to bytecodes, and immediately executed. This is to avoid dealing with lexical binding in the evaluator.

A `ModuleExp` represents a top-level form:

```
class ModuleExp extends LambdaExp
{ ...;
  public Object eval_module
      (Environment env) {
    if (body_is_simple) // Optimization
      return body.eval (env);
    Object v = eval (env);
    return ((ModuleBody) v).run (env);
  }
}
```

`ModuleExp` is a sub-class of `LambdaExp`, since it is actually a dummy function created by wrapping the top-level forms in an implicit `lambda`. The `eval_module` method evaluates the top-level forms. If the body is not simple, it invokes the `eval` in `LambdaExp` (which invokes the compiler). The result of `eval` is a `ModuleBody`, which we can `run`.

11. Code generation

A `Compilation` object manages the classes, methods, and temporary state generated as a result of compiling a single top-level `ModuleExp`.

```
class Compilation
{ ...;
  ClassType[] classes;
  boolean immediate;
  public ClassType addClass
    (LambdaExp lexp, String name)
  { ... }
  public ClassType(ModuleExp exp, ...)
  { ...; addClass (exp, ...); }
}
```

Each `Compilation` may create one or more `ClassType` objects, each of which generates the bytecodes for one class. Each `ClassType` is generated from a `LambdaExp`, including the top `ModuleExp`. The boolean `immediate` is true if we are compiling for immediate loading, and is false if the target is one or more `.class` files.

The `addClass` method does all the work to compile a given `LambdaExp`. It creates a `ClassType`, adds it to `Compilation`’s `classes` array, and generates `Method` objects for the constructor and the main `apply`*`X`* method. Once the `apply`*`X`* `Method` has been created, `addClass` emits some bytecodes to set up the incoming parameters, and then invokes the virtual `compile` method on the body of the `LambdaExp`, which generates the code that does the actual work of the procedure.

The `Compilation` constructor gets a `ModuleExp`, which it passes to `addClass`. The `compile` method of `LambdaExp` (which gets called for all `lambda`s except the dummy top-level) also calls `addClass` to generate the class corresponding to the `lambda`, and then it emits instructions to create a new instance of the generated `Procedure` class, and pushes it on the Java stack.

11.1. Targets

Most operations in the Java VM leave their result on the VM stack, where they are available for succeeding operations. The obvious and general way to compile an expression is therefore to generate bytecode instructions that leave the result (in the form of a `Object` reference) on the stack. This handles most cases quite well, but we can do better. We specify a `Target` parameter when invoking the `compile` method; the `Target` specifies where to leave the result.

```
public abstract class Target
{ ...;
  public abstract void compileFromStack
    (Compilation comp, Type stackType);
  public static final Target Ignore
  = new IgnoreTarget();
}
```

The `compileFromStack` method supports the least-common-denominator: A `compile` method can generate code to leave the result on the VM stack, and then invoke `compileFromStack`, which is responsible for moving the result to the actual target.

The simplest `Target` is an `IgnoreTarget`. It is used when the result of an expression will be ignored, but we still need to evaluate it for possible side-effects. The implementation of `IgnoreTarget.compileFromStack` just emits an instrcution to pop a value from the VM stack. Expressions that have no side-effects can check if the target is an `IgnoreTarget`, and then immediately resturn. This saves a useless push-pop pair.

The usual `Target` is an `StackTarget`. This specifies that an expression should leave the result on the VM stack. Normally, the type of the result is `Object`, but a `StackTarget` can specify some other expected type, when that can be determined. The implementation of `StackTarget.compileFromStack` is also trivial: If the type of the result on the stack is a sub-type of the expected target type, nothing needs to be done; otherwise, it generates code to do the type conversion.

Things get more interesting when we come to `ConditionalTarget`.

```
public class ConditionalTarget
    extends Target
{ ...;
  public Label ifTrue, ifFalse;
}
```

A `ConditionalTarget` is used when compiling the test expression in a conditional. The expression is evaluated as a boolean value; if the result is true, control transfers to `ifTrue`; otherwise control transfers to `ifFalse`. Using `ConditionalTarget` makes it straight-forward to generate optimal code for nested conditionals, including `and` and `or` macros, and (when inlining) functions such as `not` and `eq?`.

Finally, `TailTarget` is like a `StackTarget`, but in "tail" position. (*I.e.* it is the last thing done in a function.) It is used to do (restricted) tail-recursion-elimination.

11.2. The bytecode package

The `ClassType` and `Method` classes are in a separate `gnu.bytecode` package, which is an intermediate-level interface to code generation and Java `.class` files. It is essentially independent of Scheme or the rest of Kawa.

```
class ClassType extends Type
{ ...;
  CpoolEntry[] constant_pool;
  Method methods; // List of methods.
  Field fields; // List of fields.
  public Field addField
    (String name, Type type, int flags)
  { ...Create new field... }
  public method addMethod(String name,...)
  { ...Create new method... }
  public void writeToStream
    (OutputStream stream) { ... }
  public void writeToFile(String filename)
  { ... }
  public byte[] writeToArray()
  { ... }
}
```

The `ClassType` class is the main class of the `bytecode` package. It manages a list `Fields`, a list of `Methods`, and the constant pool. There are utility methods for adding new fields, methods, and constant pool entries.

When the `ClassType` has been fully built, the `writeToFile` method can be used to write out the contents into a file. The result has the format of a `.class` file [JavaVmSpec]. Alternatively, the class can be written to an internal byte array (that has the same layout as a `.class` file) using the `writeToArray` method. The resulting byte array may be used by a `ClassLoader` to define a new class for immediate execution. Both of the these methods are implemented on top of the more general `writeToStream`.

Each method is represented by a `Method` object.

```
class Method implements AttrContainer
{ ...;
  Type[] arg_types;
  Type return_type;
  Attribute attributes;
}
```

An `AttrContainer` is an object that contains zero or more `Attributes`. The Java `.class` file format is quite extensible. Much of the information is stored in named *attributes*. There are standard attributes, but an application can also define new ones (that are supposed to be ignored by applications that do not understand them). Each class file may have a set of top-level attributes. In addition, each field and method may have attributes. Some standard attributes may have nested sub-attributes.

```
public abstract class Attribute
{ ...;
  AttrContainer container;
  String name;
}
```

An `Attribute`'s `container` specifies who owns the attribute. The attribute also has a `name`, plus methods to gets its size, write it out, etc.

The most interesting (and large) standard `Attribute` occurs in a method and has the name `"Code"`. It contains the actual bytecode instructions of a non-native non-abstract method, and we represent it using `CodeAttr`.

```
class CodeAttr extends Attribute
{ ...;
  Variable addLocal(Type t, String name)
  { ... }
  public void emitLoad(Variable var)
  { ... }
  public void emitPushInt(int i)
  { ... }
  public void putLineNumber(int lineno)
  { ... }
}
```

As an example of the level of functionality, `emitPushInt` compiles code to push an integer *i* on stack. It selects the right instruction, and if *i* is too big for one of the instructions that take an inline value, it will create a constant pool entry for `i`, and push that.

The method `addLocal` creates a new local variable (and makes sure debugging information is emitted for it), while `emitLoad` pushes the value of the variable on the stack.

Kawa calls `putLineNumber` to indicate that the current location corresponds to a given line number. These are emitted in the `.class file`, and most Java interpreters will use them when printing a stack trace.

We mainly use `gnu.bytecode` for generating `.class` files, but it also has classes to read `.class` files, and also classes to print a `ClassType` in readable format. The combination makes for a decent Java dis-assembler.

There are other toolkits for creating or analyzing `.class` files, but `gnu.bytecode` was written to provide a lot of support for code generation while having little overhead. For example, some assemblers represent each instruction using an `Instruction` instance, whereas `CodeAttr` just stores all the instruction in a byte array. Using a linked list of `Instructions` may be more "object-oriented", and it does make it easier to do peep-hole optimizations, but the time and space overhead compared to using an array of bytes is huge. (If you do need to do peephole optimizations, then it makes sense to use a doubly-linked list of `Instructions`, but to use that in conjunction with `CodeAttr`. You will in any case want a byte-array representation for input and output.)

11.3. Literals

A Scheme quoted form or self-evaluating form expands to a `QuoteExp`. Compiling a `QuoteExp` would seem a trivial exercise, but it is not. There is no way to embed (say) a list literal in Java code. Instead we create a static field in the top-level class for a each (different) `QuoteExp` in the body we are compiling. The code compiled for a `QuoteExp` then just needs to load the value from the corresponding static field. The tricky part is making sure that the static field gets initialized (when the top-level class is loaded) to the value of the quoted form.

The basic idea is that for:

```
(define (foo) '(3 . 4))
```

we compile:

```
class foo extends Procedure0
{
  Object static lit1;
  public static
  { // Initializer
    lit1 = new Pair(IntNum.make(3),
                    IntNum.make(4));
  }
  public Object apply0()
  { return lit1; }
}
```

When the compiled class `foo` is loaded, we do:

```
Class fooCl = Class.forName("foo");
Procedure fooPr
  = (Procedure) fooCl.newInstance ();
// Using foo:
Object result = fooPr.apply0 ();
```

How does the Kawa compiler generate the appropriate `new Pair` expression as shown above? A class whose instances may appear in a quoted form implements the `Compilable` interface:

```
interface Compilable
{
  Literal makeLiteral(Compilation comp);
  void emit(Literal l, Compilation comp);
}
```

The `makeLiteral` creates a `Literal` object that represents the value of `this` object. That `Literal` is later passed to `emit`, which emits bytecode instructions that (when evaluated) cause a value equal to `this` to be pushed on the Java evaluation stack.

This two-part protocol may be overkill, but it makes it possible to combine duplicate constants and it also supports circularly defined constants. (Standard Scheme does not support self-referential constants, but Common Lisp does. See section 25.1.4 "Similarity of Constants" in [CommonLisp2].)

It is possible that the `Compilable` interface will be replaced or augmented with JDK 1.1's serialization feature.

If we are compiling for immediate execution, we do not need to generate code to regenerate the literal. In fact, we want to re-use the literal from the original source form. The problem is passing the source literal to the byte-compiled class. To do that, we use the `CompiledProc` interface.

```
interface CompiledProc
```

```
{
  public abstract void setLiterals
    (Object[] values);
}
```

An immediate class compiled from a top-level form implements the `CompiledProc` form. After an instance of the `ModuleBody` has been created, it is coerced to a `CompiledProc`, and `setLiterals` is called. The argument to `setLiterals` is an array of the necessary literal values, and the method that implements it in the compiled code causes the array of literal values to be saved in the `ModuleBody` instance, so it can be accessed by the compiled code.

12. Class, types, and declarations

Java support "reflection", that is the ability to determine and examine the class of an object, and use the class at run-time to extract fields and call methods using names specified at run-time. Kawa, like some other Scheme implementations, also supports reflection.

It seems plausible to represent a type using a `java.lang.Class` object, since that is what the Java reflective facility does. (Nine static pseudo-classes represent the primitive non-object types.) Unfortunately, there are at least three reasons why Kawa needs a different representation:

- We may need to refer to classes that do not exist yet, because we are in the process of compiling them.
- We want to be able to specify different high-level types that are represented using the same Java type. For example, we might want to have integer subranges and enumerations (represented using `int`), or different kinds of function types.
- We want to associate different conversion (coersion) rules for different types that are represented using the same class.

Kawa represents types using instances of `Type`:

```
public abstract class Type
{ ...;
  String signature;  // encoded type name
  int size;
  public final String getName() { ... }
  public boolean isInstance(Object obj)
  { ... }
  public void emitIsInstance(CodeAttr c)
  { ... }
}
```

The method `isInstance` tests if an object is a member of this type, while `emitIsInstance` is called by the compiler to emit a run-time test. Note that the earlier mentioned `ClassType` extends `Type`.

Kawa follows the convention (used in RScheme [RScheme] and other Scheme dialects) that identifiers of the form `<`*`typename`*`>` are used to name types. For example Scheme vectors are members of the type `<vector>`. This is only a convention and these names are regular identifiers, expect for one little feature: If such an identifier is used, and it is not bound, and *`typename`* has the form of a Java type, then a corresponding `Type` is returned. For example `<java.lang.String[]>` evaluates to a `Type` whose values are references to Java arrays whose elements are references to Java strings.

As a simple example of using type values, here is the definition of the standard Scheme predicate `vector?`, which returns true iff the argument is a Scheme vector:

```
(define (vector? x)
  (instance? x <vector>)
```

The primitive Kawa function `instance?` implements the Java `instanceof` operation, using `Type`'s `isInstance` method. (In compiled code, if the second operand is known at compile-time, then the compiler uses `Type`'s `emitIsInstance` method to generate better code.)

The traditional benefits of adding types to a dynamic language include better code generation, better error checking, and a convenient way to partially document interfaces. These benefits require either type inference or (optional) type declarations. Kawa so far has neither, but the compilation framework is being gradually made more type-aware. There are some hooks to support "unboxed" types, so the compiler could (potentially) use raw `double` instead of having to allocate a `DFloNum` object.

Kawa includes the record extension proposed for R^{5}RS. This allows a new record type to be specified and created at run-time. It is implemented by creating a new `ClassType` with the specified fields, and loading the class using `ClassLoader.defineClass`. The record facility consists of a number of functions executed at run-time. Many people prefer an approach based on declarations that can be more easily analysed at compile-time. (This is why the record facility was rejected for R^{5}RS.) A more declarative and general class definition facility is planned but not yet implemented.

13. Low-level Java access

Many implementations of high-level language provide an interface to functions written in lower-level language, usually C. Kawa has such a "Foreign Function Interface", but the lower-level langauge it targets is Java. A `PrimProcedure` is a `Procedure` that invokes a specified Java method.

```
public class PrimProcedure
    extends ProcedureN
{ ...;
  Method method;
  Type retType;
  Type[] argTypes;
}
```

The following syntax evaluates to a `PrimProcedure` such that when you call it, it will invoke the static method named *`method-name`* in class *`class`* with the given *`arg-type`*s and *`result-type`*:

```
(primitive-static-method class method-name
  return-type (arg-type ...))
```

When such a function is called, Kawa makes sure to convert the arguments and result between the Scheme types and Java types. For example:

```
(primitive-static-method
   <java.lang.Character> "toUpperCase"
   <char> (<char>))
```

This is a function that converts a Scheme character (represented using a `<kawa.lang.Char>` object), to a Java `char`, applies the standard `java.lang.Character.toUpperCase` method, and converts the result back to a Scheme character. Normally, the Java reflection features are used to call the specified method. However, if the `primitive-static-method` is used directly in the function position of an application, then the compiler is able to inline the `PrimProcedure`, and emit efficient `invokestatic` bytecode operations. That is the usual style, which is used to define many of the standard Scheme procedures, such as here `char-upcase`:

```
(define (char-upcase ch)
  ((primitive-static-method
    <java.lang.Character> "toUpperCase"
    <char> (<char>))
   ch))
```

Similar forms `primitive-virtual-method` and `primitive-virtual-method` are used to generate virtual method calls and interface calls, while `primitive-constructor` is used to create and initialize a new object.

You can access instance and static fields of an object using similar macros. For example, to get the time-stamp from an `Event`, do:

```
((primitive-get-field <java.lang.Event>
  "when" <long>)
 evt)
```

Kawa also has low-level operations for working with Java arrays. All these primitive operations are inlined to efficient byte code operations when the compiler knows that the procedure being called is a primitive; otherwise, the Java reflection features are used.

14. Scheme complications

Scheme has a few features that are difficult to implement, especially when you cannot directly manipulate the call stack. Some people think that first class functions with lexical scoping is one complication, but it is actually straight-forward to implement a "closure" as an object. In Kawa, such an object is a sub-class `Procedure` that includes an instance variable which references to the surrounding environment. The JDK 1.1 "inner classes" feature is (intentionally) quite similar to closures.

The two features that really cause problems are "continution capture", and tail-call elimination. The next two sub-sections will discuss how we currently implement very restricted (but useful) subsets of these feature, and then I will briefly discuss a planned more complicated mechanism to handle the general cases.

14.1. Continuations

Scheme continuations "capture" the current execution state. They can be implemented by copying the stack, but this requires non-portable native code. Kawa continuations are implemented using Java exceptions, and can be used to prematurely exit (throw), but not to implement co-routines (which should use threads anyway).

```
class callcc extends Procedure1
{ ...;
  public Object apply1(Object arg1)
  {
    Procedure proc = (Procedure) arg1;
    Continuation cont
       = new Continuation ();
    try { return proc.apply1(cont); }
    catch (CalledContinuation ex)
      {
        if (ex.continuation != cont)
             throw ex;  // Re-throw.
        return ex.value;
      }
    finally
      {
        cont.mark_invalid();
      }
  }
}
```

This is the `Procedure` that implements `call-with-current-continuation`. It creates `cont`, which is the "current continuation", and passes it to the incoming `proc`. If `callcc` catches a `CalledContinuation` exception it means that `proc` invoked some `Continuation`. If it is "our" continuation, return the value passed to the continuation; otherwise re-throw it up the stack until we get a matching handler.

The method `mark_invalid` marks a continuation as invalid, to detect unsupported invocation of `cont` after `callcc` returns. (A complete implementation of continuations would instead make sure the stacks are moved to the heap, so they can be returned to an an arbitarry future time.)

```
class Continuation extends Procedure1
{ ...;
  public Object apply1(Object arg1)
  {
    throw new CalledContinuation
      (arg1, this);
  }
}
```

A `Continuation` is the actual continuation object that is passed to `callcc`'s argument; when it is invoked, it throws a `CalledContinuation` that contains the continuation and the value returned.

```
class CalledContinuation
    extends RuntimeException
{ ...;
  Object value;
  Continuation continuation;
  public CalledContinuation
    (Object value, Continuation cont)
  {
    this.value = value;
    this.continuation = cont;
  }
}
```

`CalledContinuation` is the exception that is thrown when the continuation is invoked.

14.2. Tail-calls

Scheme requires that tail-calls be implemented without causing stack growth. This means that if the last action of a procedure is another function call, then this function's activation frame needs to be discarded before the new function's frame is allocated. In that case, unbounded tail-recursion does not grow the stack beyond a bounded size, and iteration (looping) is the same as tail-recursion. Making this work is easy using a suitable procedure calling convention, but this is difficult to do portably in Java (or for that matter in C), since implementing it efficiently requires low-level porcedure stack manipulation.

Compiler optimizations can re-write many tail-calls into `gotos`. The most important case is self-tail-calls or tail recursion. Kawa rewrites these to be a simple goto to the start of the procedure, when it can prove that is safe. Specifically, it does optimize Scheme's standard looping forms `do` and named-`let`.

14.3. Re-writing for tail-calls

Implementing general tail-calls and continuations require being able to manipulate the procedure call stack. Many environments, including the Java VM, do not allow direct manpulation of stack frames. You have the same problem if you want to translate to portable C, without assembly language kludges. Hence, you cannot use the C or Java stack for the call stack, but instead have to explicitly manage the call graph and return addresses. Such re-writing has been done before for ML [MLtoC] and Scheme [RScheme].

In Java we have the extra complication that we do not have function addresess, and no efficient way to work with labels. Instead, we simulate code labels by using switch labels. This is more overhead than regular method calls, so the regular `Procedure` interface discussed earlier will probably remain the default. Thus some procedures use the regular calling convention, and others the "CPS" (Continuation Passing Style) calling convention. The rest of this section explains the planned CPS calling convention.

```
public abstract class CpsFrame
{
  CpsFrame caller;
  int saved_pc;
}
```

Each `CpsFrame` represents a procedure activation. The `caller` field points to the caller's frame, while `saver_pc` is a switch label representing the location in the caller.

There is a single "global" `CpsContext` which owns the generalized call "stack". There may be many `CpsContext` if there are multiple threads, and in fact one `CpsContext` is allocated each time a regular (non-CPS) method calls a procedure that uses the CPS calling convention.

```
public class CpsContext
{
  CpsFrame frame;
  int pc;
  Object value;

  Object run()
  {
    while (frame != null)
      frame.do_step(this);
    return value;
  }
}
```

Each `CpsContext` has a `frame` which points to the currently executing procedure frame, and `pc` which is a case label for the code to be executed next in the current procedure. The result of a function is left in the `value` field. All of these these fields may be consider to be like global (or per-thread) registers, which is how you would ideally like to implement a CPS calling convention if you had access to machine registers. The `frame`, `pc`, and `value` fields simulate the frame pointer register, the program counter, and the function result register in a typical computer. After creating a `CpsContext` with an initial `frame` and `pc`, you would call `run`, which uses the `do_step` method to execute each step of a function until we return from the initial `frame` with a final `value`.

Consider a simple Scheme source file, which defines two functions:

```
(define (f)
  (g)
  (h))
(define (g)
  ...)
```

This would get compiled into:

```
public foo extends CpsFrame
{
  void do_step(CpsContext context)
  {
    CpsFrame fr;
    switch (context.pc)
    {
      case 0: // top-level code
        define("f", new CpsProc(this, 1);
        define("g", new CpsProc(this, 3);
        return;
      case 1: // beginning of f
        // do a (non-tail) call of g:
        fr = g.allocFrame(context);
        fr.caller = this;
        fr.saved_pc = 2;
        context.frame = fr;
        return;
      case 2:
        // then do a tail call of h:
        fr = h.allocFrame(context);
        fr.caller = this.caller;
        fr.saved_pc = this.saved_pc;
        context.frame = fr;
        return;
      case 3:  /* beginning of g */
        ...;
    }
  }
}
```

The entire code of the Scheme compilation unit is compiled into one large switch statement. Case 0 represents the top-level actions of the program, which defines the functions `f` and `g`. Next comes the code for `f`, followed by the (omitted) code for `g`. When `f` is called, a new `foo` frame is allocated, and the context's `pc` is set to 1, the start of `f`.

The body of `f` makes two function calls, one a non-tail function, and finally a tail-call. Either call allocates a `CpsFrame` and makes it the current one, before returning to the the main loop of `CpsContext`'s `run` method. The regular (non-tail) call saves the old current frame in the new frame's return link. In contrast, the tail call makes the return link of the new frame be the old frame's return link. When we return then from do_step, the old frame is not part of the call chain (unless it has been captured by `callcc`), and so it has become garbage that can be collected.

At the time of writing, the CPS calling convention has not been implemented, but I am filling in the details. It has some extra overhead, but also a few side benefits. One is that we compile an entire source file to a single Java class, and it is more convenient when there is a one-to-one correspondence between source files and binary files (for example in `Makefiles`).

Another exciting possibility is that we can write a debugger in pure Java, because we can run `do_step` until some condition (break-point), examine the `CpsFrame` stack, and optionally continue.

15. Current and Future Work

The main current priorities of Kawa are making it fully compatible with standard (R^{5}RS) Scheme, and making the ECMAScript support usable. The major tasks for R^{5}RS-compatibility are the rewrite to support general continuations and tail-calls, plus a redesign of how macros are implemented.

The ECMAScript implementation in May 1998 includes a complete lexer, and an almost-finished recursive-descent parser (with an optimization to handle binary operators). A few operations are implemented, enough to demonstrate a very limited read-eval-print loop. ECMAScripts objects are dynamic mappings from names to properties, and will be implemented using a framework that generalizes ECMAScript objects, Scheme environments and records, database records, and more (just like a collections framework unifies lists and vectors). Someone else has an an existing ECMAScript implementation; if the lawyers let us, I plan to integrate their standard objects and functions, with my `Expression` and compilation framework.

Implementing ECMAScript requires moving Scheme-specific code out of the Kawa core. We also need a more general interface to plug in new parsers, pre-defined functions, data types, and output formatting. That will make it easier to add new languages and dialects. Of special interest is re-implementing some of the ideas and syntax from my earlier Q language [Bothner88]. These include a line-oriented syntax with fewer parentheses, and high-level sequence and array operations (as in APL).

Also of interest is support for Emacs Lisp. This would require an extensive library to implement the Emacs data types (such as buffers and windows), in addition to the challenges of the Emacs Lisp language itself (it has different data types and name binding rules than Scheme), but may be a good way to build a next-generation Emacs.

There is very preliminary threads support in Kawa. It provides an interface to Java threads that looks somewhat like `delay`, except that the delayed expression is evaluated in a new thread. (The model is similar to to the "futures" concept of MultiScheme [Miller87], but there is no implicit force, at least yet.) Recent re-implementation of core classes (such `Environment` and `Translator`) has been done to support threads with optionally separate top-level environments.

An interface to graphics primitives is needed. The new Swing toolkit seems like a more powerful base then the old Abstract Windowing Toolkit.

More sophisticated Scheme code rewriting, optimizations, and inlining are also on the wishlist.

16. Conclusion

Kawa is a solid implementation of Scheme with many features. It is portable to any environment that can run Java applications. It has active use and development, a 75-member mailing list, and is used for a number of different projects. Most people seem to be using it as a scripting language for Java packages. Other people just prefer to use Scheme, but have to co-exist with Java. Scheme is beginning to get wider notice, partly because it is the basis for DSSSL [DSSSL], the standard style and formatting language for SGML. Kawa implements a number of the DSSSL extensions.

There are no benchmark results in this paper, because the state of the art in Java implementation is in such flux, because many optimizations are planned but have not been implemented, and because the different feature sets of the various Scheme implementations makes them difficult to compare fairly. But Kawa has the potential of being a reasonably fast Scheme implementation (though probably never among the very fastest), and will reap benefits from current efforts in Java compilation [GccJava].

Bibliography

[Bothner88] Per Bothner. *Efficiently Combining Logical Contraints with Functions*. Ph.D. thesis, Department of Computer Science, Stanford University, 1988.

[Budd91Arith] Timothy Budd. *Generalized arithmetic in C++*. Journal of Object-Oriented Programming, 3(6), 11-22, February 1996.

[CommonLisp2] Guy L. Steele Jr.. *Common Lisp – The Language*. Second edition, Digital Press and Prentice-Hall, 1990.

[DSSSL] International Standards Organization. *Document Style Semantics and Specification Language*. 1996, International Standard ISO/IEC 10179:1996(E).

[ECMAScript] ECMA. *ECMAScript Language Specification*. `http://www.ecma.ch/stand/ecma-262.htm`.

[GccJava] Per Bothner. *A Gcc-based Java Implementation*. IEEE Compcon 1997 Proceedings, 174-178, February 1997, See also `http://www.cygnus.com/product/javalang`.

[gmp] Torbjörn Granlund. *The GNU Multiple Precision Arithmetic Library*. 1996, (`Gmp` and its manual are available on most GNU archives.).

[Ingalls86] Daniel Ingalls. *A Simple Technique for Handling Multiple Polymorphism*. ACM SIGPLAN Notices, 21(11), 347-349, November 1986.

[JavaSpec] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[JavaVMSpec] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[Kaffe] Tim Wilkinson. *Kaffe - a free virtual machine to run Java code*. `http://www.kaffe.org/`.

[Kawa] Per Bothner. *Kawa, the Java-based Scheme System*. `http://www.cygnus.com/~bothner/kawa.html`.

[Miller87] James Miller. *MultiScheme: A Parallel Processing System based on MIT Scheme*. Ph.D. thesis, Department of Electrical Engineering and Computer Science, MIT, 1987.

[MLtoC] David Tarditi, Peter Lee, and Anurag Acharya. *No Assembly Required: Compiling Standard ML to C*. ACM Letters on Programming Languages and Systems, 1992, 1(2), 161-177.

[R^{5}RS] *Revised^5 Report on the Algorithmic Language Scheme*. Richard Kelsey, William Clinger, and Jonathan Rees (editors). 1998.

[RScheme] Donovan Kolbly, Paul Wilson, and others. `http://www.rscheme.org/`.

[SGML] International Standards Organization. *SGML (Standard Generalized Markup Language) ISO 8879*.