Kawa

Compiling Dynamic Languages to the Java VM

Per Bothner
`<bothner@cygnus.com>`

Cygnus Solutions

Usenix/Freenix 1998
(Draft version)

Other languages on JVM

Java can be thought of as two different things:

- Java as a programming language

(object-oriented, syntax similar to C/C++)

- Java as a machine/environment

(libraries + portable byte-codes run virtual machine)

- Need for other languages to co-exist in the Java environment.

High-level "scripting" languages especially useful.

- Can use extensive Java libraries, and portable bytecodes.

Benefit from Java optimization efforts.

- Many examples: NetRexx, Tcl, Ada, ...

Implementation Strategies

- Interpreter written in Java:

Good responsiveness;
Slow execution time.

- Translate to Java source:

Poor responsiveness;
Fast execution time.

- Translate to Java bytecode:

Good responsiveness;
Fast execution time.

Kawa does the last:

- Translating Scheme directly to bytecodes is much faster.
- Required for interactive response (in read-eval-print loop).
- Bytecodes are more general (bytecodes have `goto`, which is not in Java language).

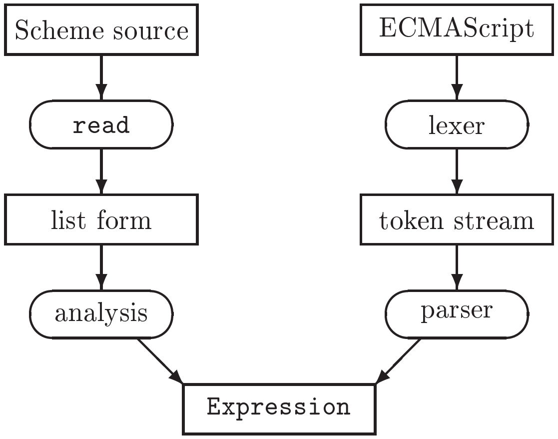
Eval – Immediate execution

- Scripting languages have `eval`:

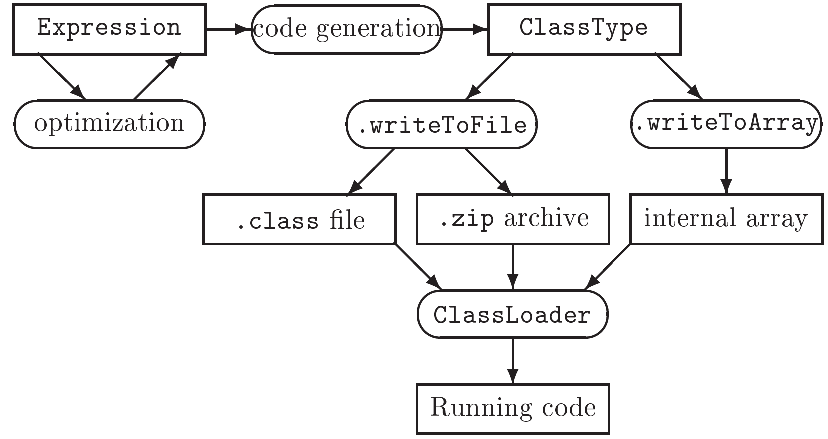
Take a command typed interactively or constructed on the fly, and immediately execute it.

- A "batch" translator does not support `eval` well.
- Kawa compiles directly to in-memory bytecodes, so is highly responsive.
- Uses a simple interpreter for "simple" expressions for even faster response.

Compilation Front-end



Compilation Back-end



Loading a Scheme source file

Read expressions until EOF – normally yields a sequence of `PairWithPosition` lists.

Wrap expressions in a dummy no-argument `lambda`.

The analysis phase does macro expansion, resolves lexical bindings. (Could do optimizations.)
Result is a `LambdaExp` expression object.

Compile to internal byte arrays containing bytecodes and class definitions.
(Uses the same format as Java `.class` files.)

Use a `ClassLoader` to convert byte arrays to loaded classes.

Apply the result, which evaluates top-level expressions.

Expressions

- Abstract `Expression` class represents (language-independent, partially processed) expressions.

- `IfExp` — conditional expression
- `ApplyExp` — application (call)
- `LambdaExp` — anonymous procedures
- `LetExp` — block with lexical bindings
- `QuoteExp` — literal (constant)
- `ReferenceExp` — variable reference
- `ErrorExp` — syntax error seen
- ...

- `compile` method compiles the expression into bytecode instructions.

- `eval` is only used to evaluate simple expressions interactively.

Code Generation

```
public abstract class Expression
{
  public abstract void
    compile(Compilation state, Target target);
  ...;
}
```

- A `Compilation` manages the state for a compilation unit, and manages one or more `gnu.bytecode.ClassType` objects, one for each generated `.class` file.
- To compile an `Expression`, invoke its `compile` method. This generates bytecodes to evaluate the `Expression`. Calls `compile` recursively for sub-expressions.
- A `Target` specifies where to leave the result.

The bytecode Package

The `gnu.bytecode` package is an efficient intermediate-level library for working with Java `.class` files:
Code generation, reading, writing, printing, disassembling.

- `ClassType` – Information about a class as a whole.
- `CpoolEntry` (abstract) – A slot in the constant pool.
- `Variable` – Local variable.
- `Field`
- `Attribute` (abstract) – Used for miscellaneous info.
- `Method` – Handles methods and byte-code instructions.
- `CodeAttr` – A `Method`'s bytecode instructions.
- ...

`CodeAttr` has many medium-level methods for generating bytecode instructions. For example:

codeattr`.emitPushInt(`*i*`);`

Generates code to push `int` literal *i* on JVM stack.

Calling Java methods

Kawa makes it convenient to call Java methods form Scheme:

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

Converts Scheme character value to Java primitive `char`, calls `toUpperCase` method in `java.lang.Character`, and converts result back.

Compiler can inline call to known primitive method.
Otherwise, Java reflection feature is used.

Similar access to array elements and object fields.

Scheme vs Java

Scheme and Java are very different kinds of languages:

- Scheme is dynamically typed, while Java is statically typed.
- Java is an object-oriented language.
- Scheme is a (non-pure) functional language: Procedures are first-class objects; lexical scoping requires closures.
- Scheme defines arithmetic on a tree of number types. Java normally uses raw machine-level numbers.

How should we map Scheme constructs into Java constructs?

Dynamic Types

Java (like Smalltalk and unlike C++) has a "rooted" inheritance graph: All classes are derived from the class `Object` (which is an alias for `java.lang.Object`).

- Scheme (and ECMAScript) have variable declarations, but without type specifications.
- hence all Scheme values belong to some sub-class of `Object`.
- Some latitude when to use standard (builtin) Java classes for Scheme values, or write our own.

Objects and Values

- Scheme booleans are represented using Java `Booleans`.
- Symbols are represented using interned Java `Strings`.
- Scheme lists, vectors, and strings use new classes in a `Sequence` (colections) hierarchy.

Sequences

`abstract class Sequence`

`class Vector extends Sequence`
Used for Scheme vectors.

```
class List extends Sequence
List.Empty = new List (); // Empty list
```

`class Pair extends List`
Has `car` and `cdr` fields.

`class PairWithPosition extends Pair`
A `Pair` with line-number information.

Numbers

- `gnu.math` package implements Scheme numbers and more.
- Forms a coherent class hierarchy.
- Provides infinite-precision rationals.
- Complex numbers provided.
- Has quantities, with units and dimensions.

Procedures

```
public abstract class Procedure
{
  public abstract Object applyN(Object[] args);
}
```

- A primitive procedure is written in Java as a sub-class of `Procedure`. (*E.g.* `+` defines `applyN` to add arguments.)
- A Scheme function is compiled into a subclass of `Procedure`, with the Scheme body compiled into body of `applyN`.
- Allocating instance of a `Procedure` sub-class creates Scheme procedure value.
- Nested procedure has a field for the "static context" (inherited lexical environment).
A closure is an instance of such a class.

Tail-calls

Scheme language requires "tail-call-elimination", which is a generalization of "tail-recursion-elimination":

`CALL F` followed by `RETURN` should be same as `GOTO F`.

I.e. current frame must be popped before `CALL`, to avoid stack growth. Allows iteration and state machines to be expressed using recursion.

Best handled with suitable calling convention, which is not portable in Java.

Kawa implements tail-recursion-elimination only: If compiler sees a call to the current function, it replaces it by a goto.

General tail-call-elimination will be implemented using a new heap-based calling convention; can co-exist with the current faster calling convention.

Continuations

Continuations provide a way to "capture" the current stack environment.
They can be used to implement new control structures: co-routines, backtracking, exception handling, and more.

Can be implemented by copying the stack – but this requires assembly-level code.

Kawa implements limited continuations, sufficient to implement catch/throw and exception handling. It uses the Java exception handling mechanism.

Full support will be added, based on the new tail-call convention,

Other languages

Most of the Kawa is independent of Scheme.

The same techniques, and most of the code, could be re-used to implement other languages, especially dynamically typed ones, such as Tcl, Rexx, Smalltalk, APL, ...

New languages may require new data types – just write the appropriate Java classes.

Most languages require their own parser. This would translate text into `Expression`s.

Each language has a different standard environment containing pre-defined values, types, and functions.

ECMAScript

- ECMA standard no. 262 for the core of JavaScript.
- Very dynamic object-based language with prototype inheritance, but no classes.
- Lexer and parser written. Most pre-defined functions and objects missing.
- Generates `Expression` and compiles to Java bytecodes.

DSSSL and SGML

- Industry migrating to SGML for document storage and manipulation.
- HTML is an instance of SGML; XML is simplified SGML.
- DSSSL is the ISO standard language for specifying style and processing of SGML.
- DSSSL is basically standard Scheme with no side-effects plus some extensions, plus lots of standard "classes" for manipulating text flow etc.
- The original (pre-Cygnus) Kawa was written to process DSSSL. Kawa implements some of the DSSSL extensions. DSSSL seems be the cause of re-newed interest in Scheme (and Kawa).

Status and Future work

Kawa is used for a number of commercial and academic projects, and has a 75-member mailing list.

Kawa is still being actively developed and enhanced.

Plans include:

- Implement missing features of standard Scheme (R^5RS).
- Finish ECMAScript implementation.
- Implement Emacs Lisp (core).
- Play with an array language.
- Design a graphics interface.
- Add a module system.
- Support optional type declarations and (local) type inference. Support using raw Java data types in Scheme.
- ...

Fine print

- Not a Cygnus product.
- No connection with Tek-Tools' Java IDE of the same name.
- Home page: `http://www.cygnus.com/~bothner/kawa.html`.