Scheme, XQuery, and JavaFX: Compiling using Kawa or Javac

Per Bothner
GNU; Sun Microsystems

Preview

- Kawa background.
- Kawa Scheme.
- Using `gnu.bytecode` for bytecode generation.
- Higher levels of Kawa: `gnu.expr`.
- Optimization machanisms.
- Compiling JavaFX Script using `javac`.

Kawa Languages

- Kawa started as a Scheme compiler in 1996.
- Scheme is a dynamically-typed strict (impure) functional language, in the Lisp family.
- Emacs Lisp support added in 1999, for the Swing-based "JEmacs" clone of the Emacs editor.
- XQuery support added in 2001. XQuery is a W3C language for processing XML.
- Others have implemented Nice, BRL, ...

Implementation strategies

- Interpreter — slow.
- Compile to Java source, which is compiled with `javac` — unresponsive, hard to get debugging etc quite right.
- Compile to Java classes directly — fast, responsive, hard [Kawa].
- Compile to Java AST, compile that using `javac` as library [JavaFX].
- Provide both a compiler and an interpreter [Many].

Kawa Scheme - a "better Java" (blue pill)

- Declare classes, interfaces, methods; extend from Java classes and interfaces.
- A dynamic language with optional Java-compatible typing.
- Create/access Java objects and arrays; call Java methods; access fields; perform arithmetic on unboxed primitive - all **as efficiently as in Java**.

Beyond Java (red pill)

- Complete implementation (except for continuations) of standard (R5RS) Scheme with common extensions (SRFIs).
- Efficient true multiple inheritance.
- A numeric hierarchy, including units and efficient bignums.
- Syntactic extension (macros).
- Keyword parameters - convenient for compact object construction.
- Read-eval-print loop.

Colon notation

- Notation: *`expr:name`*
  means get part named *`name`* in *`expr`*.
- Instance field: *`obj:fldname`*
- Static field: *`classname:fldname`*
- Instance method call: `(`*`obj:methname arg ...`*`)`
- Static method call: `(`*`classname:methname arg ...`*`)`
- Compound name: *`namespace:name`*
- XML qualified name: `'`*`prefix:name`*

Named parameters - object construction

- Object creation - works for any class name and bean property names:

```
(Button text: "Yes"
        action: (lambda (e) (whatever)))
```

- Also XML/HTML element:
  `(html:a href:"link.html" "Click here!")`

Execution overview

1. Read/parse source code, yielding internal Expression data structure.
2. Tree-walk the Expression for analysis, optimimization, and selecting representations.
3. Generate code, using gnu.bytecode.
4. Load-and-execute generated code, or save for future use.

Code-generation with gnu.bytecode

- gnu.bytecode is a self-contained library for bytecode generation

and manipulation.
- Like ASM and BCEL, but focused more on generation.
- Slightly higher-level than ASM or BCEL.
- Generate efficient code - and do it easily and fast.
- ClassType etc can be generated by compiler;
  read from .class file; or derived from reflection.
  (Annoying duplication of java.lang.reflect APIs because latter classes are final.)

Code-generation snippet

```
CodeAttr code = method.startCode();
code.pushScope();
Variable r = code.addLocal(Type.intType, "result");
code.putLineNumber(123);
code.emitLoad(code.getArg(0));
code.emitPushInt(10);
code.emitMul();
code.emitStore(r);
...
code.popScope();
```

Code output

- emitXxx routines append instructions to byte array.
- The emitted code is tentative, since we support post-generation code re-arranging:
  - Eliminating redundant gotos.
  - Picking narrow or wide jumps as needed.
  - Re-ordering fragments to avoid gotos.
- Similar to "relaxation" in traditional assemblers.

Compiler-writer conveniences

- API is compact and straight-forward.
- Chooses best instruction based on operands and types.
- Keeps track of types.
- Handles line-numbers and other debug information.
- On-the-fly consistency checking.

Control structures

- Switch/case expression helper.
- Try-catch-finally expression support.

- Both switch and try can be value-returning expressions, not just statements.
- Loops are handled as tail-calls to inlined functions.

Example: optional parameters

```
// if (i < argsArray.length)
code.emitPushInt(i);      // param number
code.emitLoad(argsArray); // incoming args
code.emitArrayLength();
code.emitIfIntLt();
// then argsArray[i]
code.emitLoad(argsArray);
code.emitPushInt(i);
code.emitArrayLoad();
code.emitElse();          // else
// evaluate default value
defaultArgs[i-fixedArgs].compile(comp, paramType);
code.emitFi();
```

StackMapTable

- On Java6 automatically generates StackMapTable.
- Fast on-the-fly generation, using available type information, rather than bytecode analysis.
- try-finally generates an inline subroutine rather than duplicating finally body — but now uses conditional return rather than jsr.

---

try-finally example

```
    invokestatic f.useFile()int
    istore_0
    aconst_null
 5: astore_1
    invokestatic f.closeFile()void
    aload_1
    ifnull 20
    aload_1
    athrow
handle(Throwable) range 0-5:
    iconst_0
    istore_0
    goto 5
20: iload_0
    ireturn
```

The gnu.expr package

- Core of Kawa compiler.
- Expression: Represent expressions and blocks (AST).
- Language: Abstract class with language-specific hooks.
- ExpWalker: Expression visitors.
- Literal: Possibly-structured constant data.
- Target: Where to put expression result - by default the JVM stack.

Expression sub-classes

- QuoteExp - constants.
- ReferenceExp - variable and function references.
- ApplyExp - function/method call.
- LambdaExp - function value/definition.
- IfExp - conditional.
- TryExp - try/catch/finally, possibly value-returning.
- SetExp - definitions and assignments.
- ... etc ...

First-class functions

```
(define (foo x) ...)

compiles to:

class ... extends  ModuleBody {
  public static Objectfoo foo(Object x) { ... }
  public ModuleMethod foo =
    new MethodMethod(this, FOO, "foo", 1);
  public Object apply1(ModuleMethod method, Object arg1) {
    switch (module.selector) {
      case FOO: return foo(arg1);
      default: return super.apply1(method, arg1);
  }
}
```

Tail-calls

- Internal tail-calls compiled to goto.
- By default Kawa doesn't handle general tail-calls.
- An option modifies function-calls to take hidden "context" parameter.
- This supports tail-calls; plus some other uses, including SAX-like output sink.

- Default: off - for better performance and Java compatibility.
- The two kinds of functions can be freely mixed.

- Only limited continuation support so far.

Optimizing builtin functions

- Operations like addition are represented as calls (ApplyExp) to a known (constant) function.
- Such functions may implement Inlineable, which provides a compile method, that takes an ApplyExp and Target.
- It can emit *arbitrary* bytecode instructions.
- Example: If the operands to + have primitive-int arguments, we emit iadd instruction.
- Functions can also have hooks into tree-walking, which allows custom type inference and re-writing at the tree level.

Adding control structures

- Language-specific control structures can be translated to calls to builtin functions.
- Sub-expressions can be encapsulated as function-valued parameters.
- Example: a loop is a call to a map function.
- These builtins can implement Inlineable, and thus compile to custom bytecode, and also have hooks into the tree-walker.

Fast compilation

- Compiler is fast enough to not need an interpreter.
- Used to interpret "simple expressions", but no longer do.
- Supports read-eval-print-loop - compiles each line.

JavaFX Script - based on javac

- Sun decided to build a compiler for JavaFX Script.
- Considered using Kawa, but decided to build on javac.

JavaFX compiler flow

- ANTLR parser creates JavaFX-specific JFXTree AST nodes, which extend javac's JCTree
- "Attribution" does name and type resolution.

(Currently, rather simple-minded type inference.)
- Then we translate JFXTree to javac AST.
- Re-attribute translated AST, using javac, enhanced with "block expressions".
- "Lower" and code-generate as in javac.

Benefits of using Javac

- Compatibility in command-line processing, character set decoding, profile/target support.
- Useful components: Message catalogs; localization, builtin support for debugging, StackMapTable, annotations, generic types.
- Useful data structures reduce design and busy-work: Symbols, Types, vistoror framework.
- Baby steps in terms of generalizing javac to library useful for other languages.

Problems of using Javac

- Javac AST very Java-centric - assumes a statement vs expression distinction.
- Ended up with a new JavaFX-specific AST hierarchy.
- Extra time spend doing name-lookup and type-checking twice. (Could be avoided.)
- Have to fit into valid Java tree structure. For example: Cannot mark a method as synthetic.

Last words: Avoid excessive dynamism?

- Language implementors fighting with dynamic objects, classes, source includes, etc. Lots of hard work and cleverness to achieve acceptable efficiency.
- To what end?
- C++'s goal was "you only pay for it if you use it". "Modern" languages are moving in the opposite direction.
- Until implementors learn the benefits of lexical scoping, static binding, etc.
- Perhaps what we need is something like Scala, but simpler, with optional typing and type inference?

Links

Kawa
    http://www.gnu.org/software/kawa/

Qexo (Kawa XQuery)
    http://www.gnu.org/software/qexo/

JavaFX Script compiler
    https://openjfx-compiler.dev.java.net/

Per Bothner <per@bothner.com> <per.bothner@sun.com>
    http://per.bothner.com/