A Novel Approach for Porting Perl to the Java Virtual Machine

Bradley M. Kuhn
Free Software Foundation
59 Temple Place, Suite 330
Boston, MA 02111-1307 USA
bkuhn@gnu.org

Abstract

At the fourth Perl Conference, two possible approaches for porting Perl to the Java Virtual Machine (JVM) were presented. Both these approaches have unfortunate drawbacks that render them cumbersome for all but trivial subsets of Perl. This paper discusses a novel approach for porting Perl to the JVM. Due to an "impedance mismatch" between `perl`'s intermediate representation (IR) and the bare JVM, the more generalized GNU Kawa IR is used as a companion to `perl`'s IR. The `perl` IR is massaged into Kawa's "middle-layer" IR. In turn, Kawa's IR compiles directly to JVM bytecode. While Kawa's IR does not hand us a ready-made port of Perl to the JVM, it does provide more infrastructure than is available with any other system. Using Kawa's IR, together with the `B` module, brings us much closer to a full Perl port to the JVM than has hitherto been possible.

Introduction

At the fourth Perl Conference, two possible approaches for porting Perl to the Java Virtual Machine (JVM) were presented [6, 8]. One approach sought to compile Perl directly to Java source code. The other used the `B` module to facilitate direct compilation from the `perl` intermediate representation (IR) to JVM assembler. Both these approaches have unfortunate drawbacks that render them cumbersome for all but trivial subsets of Perl. In particular, these two approaches share their most fatal drawback. They both must effectively emulate the entire `perl` back-end (the Perl Virtual Machine, or PVM) in the Java environment.

The PVM is discussed in [1, 2, 6, 7] and in the *perlguts* manpage. The PVM can take the IR generated by *perl*'s front-end, along with a set of Perl data type implementations, and evaluate the IR (and in doing so, execute the code given by the Perl programmer).

When porting to the JVM, previous approaches needed to effectively emulate parts of the PVM using hand-coded Java. Specifically, both approaches presented last year do so by using hand-coded Java classes to emulate Perl's native data structures on the JVM. Such work is easy for small examples, but as a full port of Perl is pursued, such work becomes unwieldy. It amounts to reimplementation of the entire PVM in Java.

Parallel to previous work on porting Perl to the JVM, Per Bothner had continued to develop the GNU Kawa system for Scheme [3, 4]. As he worked, he began to abstract some of the components of Kawa that were not Scheme-specific, and he developed them into an interesting layer of abstraction. The end result was a high level and generalized intermediate representation (IR) that compiled to the JVM.

This new Kawa IR filled a large gap in the JVM porting community. Namely, it provided an infrastructure to port many languages to the JVM. This paper discusses how that IR can be used to facilitate a better, easier and more robust port of Perl to the JVM than those methods previously attempted.

A New Layer of Abstraction

The JVM is indeed generalized insofar as it provides a low level, object oriented, platform independent assembler. However, most compilers today are not written to target some specific assembler directly. Usually, a higher level IR is used to represent the program. Since the IR is simpler to work with than the assembler, ports of new languages are easier. For example, the GNU Compiler Collection, GCC, was one of the first compilers to extensively use such an IR. GCC now supports six different languages on the front-end, and dozens of architectures on the back-end [9].

As discussed, `perl` does have its own IR. [6] discussed how this IR can be used to generate JVM assembler directly. It was discovered, however, that `perl`'s IR simply did not map well onto the JVM.

In hindsight, this is not surprising. The `perl` IR was not designed, as GCC's was, to ease the burden of creating new front-ends and back-ends. In fact, `perl`'s IR was actually designed specifically to work with and depend on the PVM. Thus, it makes sense that using `perl`'s IR to port to new architectures is difficult.

Given this reality, the next step is to find a way to still leverage the useful `perl` front-end IR in a way that will better facilitate a port to the JVM. The solution proposed in this paper considers using a *second* IR that is specifically designed to function with the JVM. To finish the job, a translator is being written that massages `perl`'s IR into the other IR.

This approach is easier, because an IR designed to be general will have better facilities to implement various language features. For example, features like lexically-scoped variables and anonymous subroutines are common in many languages. If the IR supports these features, translating from `perl`'s IR to the new IR will be easier. And, even for those features that are unique to Perl, a good IR provides facilities (better than those available on the bare JVM) to implement those additional features.

The GNU Kawa IR serves as this new IR. Originally designed for Scheme, Kawa's IR has been generalized to support basic generic features that are common in many very high level languages. In addition, Kawa has extensible parts, too. For example, the user of the IR can implement a class that controls variable binding lookup. Yet, the IR's object oriented interface hides the details of how that variable binding lookup operates internally. This feature alone can help simplify one of Perl's most complex features—tied variables.

The Kawa IR

The Kawa IR is provided by the Java package, `gnu.expr.*`. This package, described in [3, 4], provides classes that represent nodes in a parse tree. Each node in that tree is a subclass of the abstract base class, `Expression`.

Each `Expression` has two key methods: `eval` and `compile`. The former is used when the source program is run interactively. When invoked, `eval` evaluates the current expression (and often subexpressions) in the context of the current run-time environment. The `compile` method is used to compile the expression (and often subexpressions) to a JVM class file for later use.

One of the most important subclasses of `Expression` is the `LambdaExp`, which provides the basic semantics of a `lambda` expression in Scheme. However, the `LambdaExp` class is an abstraction of `lambda`, and thus does not contain anything specific to Scheme. The `LambdaExp` can be used to enclose functions as well as objects. `LambdaExp` has straightforward methods for handling function parameters, localized and lexically scoped variables and variables internally captured by a closure.

The `ModuleExp` is a subclass of `LambdaExp`. A `ModuleExp` maps onto a JVM class, and can include declarations of both static and instance variables, as well as static and instance methods. When translating `perl`'s IR to Kawa's IR, a `ModuleExp` is used to represent each Perl `package`.

An "Add and Print" Example with Kawa

To best exemplify how Kawa is used to represent a Perl program, consider the following simple Perl program that adds and prints the result:

```
$x = 5;
$y = $x + 7;
print "RESULT: $y", "\n";
```

Figure 1 shows a diagram representing how this "add and print" example is compiled to Kawa's IR.

The top-level Perl `package` is compiled to a `ModuleExp`. Contained by that `ModuleExp` are two `Declaration` objects for the two scalar variables used in the program.

The `ModuleExp` contains one subexpression, a `BeginExp`. A `BeginExp` is used to organize a set of subexpressions, and evaluate them for their side effects.

One such subexpression seen in this example is the `ApplyExp`. This expression is a generalized way to evaluate a function. The functions can be existing `LambdaExp`s, or, as in this case, a reference to some known procedure. In this example, we refer to a `PrimProcedure` object, which simply refers to a known function that is written in Java. However, an `ApplyExp` can just as easily refer to unnamed functions compiled by Kawa, or named functions written in Perl or Scheme. This flexibility is a great advantage over other methods for compiling non-Java languages to the JVM.

Another particularly flexible mechanism is variable binding. The `SetExp` is used to set a variable to a particular value. Each `SetExp` refers to some variable. However, the binding of that variable need not be specified directly. The variable is looked up by Kawa in the current context. Thus, worrying about how the variable is bound is left up to the IR. The compiler from perl's IR to Kawa's IR need not specifically worry about that issue.

The complement to the `SetExp` is the `RefExp`. A `RefExp` refers back to a variable so its value can be used as an r-value. Variable binding and lookup are handled completely by Kawa. The programmer who generates the IR need not be concerned with those details; the programmer need only tell Kawa the scoping style of variable (lexical or dynamic) and attach the declaration to the right Kawa expression. Kawa does the rest.

Complex Perl Features with Kawa

It should be noted that while in the previous example, we are dealing only with simple scalars, `SetExp`s and `RefExp`s can also be used to solve the issues with Perl's `tie`. Kawa has a binding mechanism, whereby variable declarations can have particular constraints associated with their use. `SetExp`s and `RefExp`s are evaluated or compiled with references to these constraint objects. At runtime, the constraint objects are resolved. In this manner, JVM code can automatically be generated to handle complex variable access mechanisms, such as exist with Perl's `tie`.

This flexibility is typical of Kawa's facilities. Kawa is specifically designed as an IR for very high level languages. Thus, those features one expects in such a language are completely native and natural to Kawa. Kawa provides abstractions for features that required careful and copious hand-coding in previously attempted ports.

As another example, consider Kawa's core expression type, the `LambdaExp`. This expression natively provides full support for what Perl calls anonymous subroutines, and even provides support for closures. These are complex language features that require careful consideration to implement. Kawa's IR provides an abstraction of these features to the compiler writer, so that she can focus only on the specific details of how the given source language handles and uses those features.

Some Implementation Details

Kawa's facilities are extensive and greatly ease the back-end compiler work when porting a new language to the JVM. However, this approach still requires separate work to handle all front-end issues.

Also, Kawa is written almost entirely in Java. Thus, for Kawa to be useful to the compiler writer, the front-end must be able to link with Java code easily. This poses a problem when trying leverage the existing `perl` front-end, as it is written in C.

In addition, as is noted in [2, 5, 6, 7], the easiest way to leverage `perl`'s front end is to use the `B` and `O` modules. Of course, these modules are written for access via Perl itself. Thus, we are presented with a bootstrapping problem: linking C and Java together is not yet particularly easy (GCC 3.0 will likely make this easier, but not trivial), and Perl modules cannot yet interface with Java easily until we are done a full JVM port.

To solve this problem, the Java-Perl Lingo (JPL) is used. The JPL is a part of the core *perl* distribution that eases the integration of Java and Perl. The JPL allows Java and Perl code to integrate somewhat seamlessly via the Java Native Interface (JNI). (More details on the JPL are available in [6, 7].)

Thus, the core of the new `perljvm` implementation is a `B` module that uses the JPL to make calls to Kawa's API. This `B` module instantiates various Kawa expression objects, builds a Kawa IR tree from the internal `perl` IR tree, and finally tells Kawa to write the compiled JVM bytecode.

Massaging the `perl` IR into the Kawa IR is relatively straightforward (at least compared to previous methods). Of course, `perl`'s IR is on a very high level, but so is Kawa's IR. Thus, Kawa mitigates the "impedance mismatch" between the `perl` IR and the lower level JVM code.

Drawbacks to Using Kawa

Of course, Kawa is not a panacea to all JVM porting woes. Compiler writing and porting remains a complex issue on all fronts, and Kawa can only solve some of the problems.

The major drawback is the complexity of Kawa itself. A programmer must have deep knowledge of the Kawa IR to use it effectively. Good overview documentation is available, but detailed examples of porting new languages are lacking. To effectively use Kawa, detailed reading of its source code is necessary.

Meanwhile, since the front-end is still done with a `B` module, the programmer must still understand the intimate details of the canonical `perl` implementation. For a programmer to effectively work on the project, she must have a depth of knowledge of both Kawa and `perl`. Currently, there a few programmers who have both the time and inclination to acquire such knowledge, and thus progress on the project is slow.

Not only is Kawa complex, but also its API has not yet solidified. As active porting efforts of non-Scheme languages to Kawa continue, Kawa's features are being generalized more and more. This is advantageous in the long run, as it will make Kawa more generally useful. However, in the short term, those who use Kawa as a porting infrastructure must keep up closely with Kawa development. This factor further impedes the ability for new programmers to join the project.

Finally, some have voiced a concern that given the slow speed of current Java environments, and the high level of abstraction used by Kawa, there will be serious efficiency problems with this approach. This particular drawback can be answered in two ways. First, it is clearly better to have an inefficient port rather than no port at all. This work is the first of the many attempts of porting Perl to the JVM that has shown real promise to work for non-trivial Perl programs. We must continue forward, and then focus on optimizing for speed once the job is done.

Second, native compilation of both Java source and JVM bytecode is an active area of development. Such native compilation could speed Kawa-generated bytecodes up considerably. In addition, since this port centers around Kawa, the port will benefit from optimizations added directly to Kawa. Such optimizations are one focus of current Kawa development.

Conclusions and Future Work

Porting Perl to the JVM is a unique challenge. The complexity of Perl's native data types and of parsing Perl cripple most typical methods of compilation for the JVM. This paper presents the most likely candidate method for finally achieving a port of Perl to the JVM.

Clearly, it made sense to mitigate the parsing issues by using `perl`'s existing front-end. Use of that front-end leverages the many person-years that went into that canonical implementation of Perl, while limiting the problem scope to a much more manageable task.

However, using `perl`'s front-end was not without its own challenges. Earlier approaches implicitly assumed that the PVM could be easily mapped onto the JVM (or Java) directly. The `perl` IR proved too inflexible for this method, as it was not designed to be a generalized IR. Attempting to use it as such quickly took `perl`'s IR to its limits. In the case of a JVM port, this meant that far too much new Java code was required to support even the simplest of features.

In addition, reliance on the JVM to act in the same manner as the PVM proved to be another problematic assumption. The JVM is not designed to be a perfectly general virtual architecture. This is not necessarily a flaw in the JVM, but it does indicate that using the JVM in such a general way is not the best approach.

Fortunately, the Kawa system provides a more generalized method for compiling non-Java languages to the JVM. Kawa introduces a layer of abstraction that is absolutely necessary if the JVM is to be used as a general architecture for non-Java languages. Other projects that port non-Java languages to the JVM would do well to revisit Kawa in its current state, and perhaps migrate to it. Such migration would not only alleviate problems faced in those projects, but standardizing on Kawa would also ease the task of integrating the object models of the various JVM ports.

In the specific case of our Perl port, Kawa solved some even more difficult problems. Using Kawa overcomes the deficiencies inherent in `perl`'s IR and its tight coupling with the PVM. By providing a higher-level IR, Kawa eases the reuse of `perl`'s IR. The minutiae of book-keeping required when trying to compile `perl`'s IR directly to JVM bytecode disappears when Kawa is used. Semantic mapping is the sole focus, and the common details of compilation are handled internally by Kawa's compilation process.

The most open area for future work is to continue porting more of Perl to the JVM via Kawa. Currently, only a subset of Perl is supported, but the path is clear. Kawa's infrastructure makes the task of porting Perl to the JVM much more feasible. It is hoped that more developers will become interested in the project now that this work has laid out a clear path to the goal.

Also, this work benefits more than just the Perl community. Already, the work of porting Perl to the JVM via Kawa has inspired enhancements to Kawa itself. It is hoped that continued efforts to port a unique language like Perl via Kawa will help Kawa to become even more generalized and robust.

As Microsoft's .NET system looms on the horizon, The Kawa/JVM environment can be a real competitor to it. Of course, a Kawa/JVM system has the added advantage that it is completely open and free software, while Microsoft's .NET will no doubt remain proprietary. It is hoped that this advantage can carry a Kawa/JVM-based language system, along with a Perl port to Kawa/JVM, to success for users and programmers alike.

Software Availability

The `perljvm` software is licensed under the same license as `perl`: a disjunction of the Artistic License and the GNU General Public License.

The `perljvm` software is available via *savannah*, the GNU project development server. See the URL, *http://savannah.gnu.org/projects/perljvm/*, for details.

More information about Kawa is available at *http://www.gnu.org/software/kawa/*.

Affiliation Note

This work is being done as part of the Free Software Movement, under the auspices of the GNU project. Of course, since we in the Free Software Movement by and large agree with the Open Source Movement on matters of technology, we are happy to present this work at an Open Source event. However, we would like to note for the record we are not directly affiliated with "open source".

References

[1]
Aas, Gisle. "Perl Guts Illustrated, Version 0.09". [Online] Available at *http://gisle.aas.no/perl/illguts/*. November 1999.

[2]
Beattie, Malcolm. "The Perl Compiler". *The Perl Journal*. Volume 1, issue 2 (Summer 1996), pages 34–36.

[3]
Bothner, Per. "Kawa — Compiling Dynamic Languages to the Java VM". *USENIX 1998 Annual Technical Conference: Invited Talks and FREENIX Track*, pages 225–272. New Orleans, Louisiana, USA. June 1998.

[4]
Bothner, Per. "Kawa: Compiling Scheme to Java". *Proceedings of the 1998 Lisp Users Conference*. Berkeley, CA, USA. November 1998.

[5]
Jepson, Brian. "Taking Perl to the Java Virtual Machine". *The Perl Journal*. Volume 4, issue 4 (Winter 1999), pages 53–59.

[6]
Kuhn, Bradley M. "`perljvm`: Using B to Facilitate a Perl Port To the Java Virtual Machine". *Proceedings of the Perl Conference 4.0*, pages 17–23. Monterey, CA, USA. July 2000.

[7]
Kuhn, Bradley M. "Considerations on Porting Perl to the Java Virtual Machine". Master's Thesis. Department of Electrical and Computer Engineering and Computer Science. University of Cincinnati, Cincinnati, OH, USA. January 2001.

[8]
Mccrae, Raymond, et al. "PerlCaffeine: Compiling Perl to Java". *Proceedings of the Perl Conference 4.0*, pages 127–135. Monterey, CA, USA. July 2000.

[9]
Stallman, Richard M. *Using and Porting the GNU Compiler Collection (GCC)*. Free Software Foundation, Boston, MA, USA, Edition 2.95. August 2000.