UNIVERSITY OF CINCINNATI

22 January, 2001

I, Bradley M. Kuhn,
hereby submit this as part of the requirements for the degree of:

Master of Science

in:

Dept. of Elec. and Comp. Eng. and Comp. Science

It is entitled:

Considerations on Porting Perl to the Java Virtual Machine

Approved by:
Fred Annexstein
John Franco
Larry Wall
Hongwei Xi

Considerations on Porting Perl to the Java Virtual Machine

A thesis submitted to the

Division of Research and Advanced Studies
of the University of Cincinnati
in partial fulfillment of the requirements for the degree of

Master of Science

in the Department of Electrical and Computer Engineering and Computer Science

of the College of Engineering

2001

by

Bradley M. Kuhn

B.S., Loyola College In Maryland, 1995

Committee Chair: John Franco, PhD.

Abstract

The Java Virtual Machine (JVM) is perhaps the most interesting aspect of the Java programming environment. Much attention has been placed on porting non-Java languages to the JVM. Such ports are useful since JVMs are now embedded in hardware devices, as well as in software applications such as web browsers. In addition, well designed JVM ports can utilize the JVM as a common object model for multiple languages, allowing larger applications to easily be written in and scripted with multiple programming languages.

This thesis presents a survey of possible approaches for porting non-Java languages to the JVM. The advantages and disadvantages of each approach are considered. Examples of JVM ports of other programming languages, such as Python, Scheme, and Tcl are presented and considered.

The focus, however, is a port of Perl to the JVM. The internals of the existing Perl implementation are discussed at length with examples. The `perl` front-end parser, lexer and intermediate representation (IR) are described in detail. The default Perl compiler back-end, called the Perl Virtual Machine (PVM), is considered and described.

Two approaches for porting Perl to the JVM are presented. The first approach reuses the existing `perl` front-end via Perl's `B` module to compile directly to JVM assembler (using Jasmin syntax). This approach is described and examples are given. The problems of mapping the PVM onto the JVM, the lack of generalization of the existing `perl` IR, and complications caused by the JVM bytecode verifier are introduced and explained.

The second approach massages the existing `perl` IR into the Kawa system's more generalized IR. This approach is much more successful than direct compilation, and reasons are given to make that case. Kawa's IR is presented, and an example of a Perl program compiled to Kawa's IR is given.

Finally, conclusions and lessons learned from this work are presented. A framework for the future work required to complete a Perl port to the JVM (via Kawa) is given. A brief comparison between the Kawa/JVM infrastructure and Microsoft's .NET/C# system is presented.

Dedication

I dedicate this work to my fiancée, Elizabeth A. McKeever. She indeed knows how to love a hacker—which is certainly not an easy task. Thanks, keever, for always reminding me there is a world outside of computer science.

Acknowledgements

First, I thank the Usenix Association for a student scholarship and stipend during the early research of this work. In addition, the Department of Electrical and Computer Engineering and Computer Science provided me with a University Graduate Scholarship for much longer than I would have thought possible, and for that I am very grateful.

I would like to thank my committee for taking the time to read my thesis and listen to my defense. I give special thanks to Larry Wall for traveling so far to serve on my committee.

Linda Gruber, the ECECS graduate program coordinator, deserves special mention. Her relentless work to ensure that graduate students have what they need is an asset to the department.

I would also like to thank the other Canonical Hackers for their continuous support of my work and my ideas, even when I doubted them myself.

I thank also the Perl community in particular and the free software community in general. Without the plethora of free software that is available for Perl and Java, this work would not have been possible.

Finally, I am grateful for the Cosource system, and those who helped fund my software development through that system.

Contents

List of Figures

Chapter 1

Introduction

Since its release in the late 1980s, the Perl programming language has evolved from a relatively simple scripting language to an advanced, portable, garbage collected programming language. The current version of Perl supports sophisticated features such as object oriented programming, functional programming, advanced pattern matching and complex data structures. Separate to the evolution of Perl, the Java language environment (which includes the Java Virtual Machine) has become a popular choice in its own right, because of Java's comparable portability, its threading model, and its garbage collected object model. The Java and Perl communities remain largely separate, partly due to a lack of tools to integrate the two languages.

This thesis addresses the problem of porting Perl to Java environment. To begin, in this chapter, the Java Virtual Machine (JVM) is introduced and briefly described. The usefulness of integrating languages, such as Perl, with Java through the use of the JVM itself is addressed. Further, the particular importance of direct JVM ports is explained. Following that, some specific challenges on porting languages such as Perl to the JVM are introduced.

In Chapter 2, the various possible approaches for porting non-Java languages to the JVM are introduced and discussed. In each case, the possibility of using that approach for Perl is briefly addressed. Then, in Chapter 3, the internals of `perl`[1] are discussed and explained. Following that, in Chapter 4, the first approach attempted for the Perl/JVM port is presented, and its drawbacks are explained. A second approach is presented in Chapter 5, and its inherent advantages are explained. Finally, Chapter 6 draws some conclusions based on this work, and addresses how and why this work should continue.

[1]In the Perl community, the lowercase word, "`perl`", refers to the canonical (and currently only) implementation (which is written in C) of Perl. The uppercase word, "Perl", refers to the language itself. This convention is also used throughout this document.

1.1 The Java Virtual Machine

The Java Virtual Machine (JVM) is described in detail in [19, 17]. In this section, a summary of the JVM is presented. The information in this chapter is the prerequisite knowledge about the JVM required for understanding this thesis.

1.1.1 Purpose of the JVM

The JVM was originally designed as the "cornerstone of the Java programming language" [17, page 3]. The developers of Java wanted a virtual platform that would allow Java to be as portable as possible. With that goal in mind, they developed the JVM as a generalized, assembler-like instruction set for object oriented programming.

The JVM is a well defined specification for a virtual architecture. Each implementation of the JVM must adhere to this specification. This well defined specification is key to the success of the JVM—without it, incompatible JVM environments would be common. By contrast, with a clear specification available, incompatibilities in particular JVMs can easily be established as violations of the specification. A number of both proprietary software and Free Software JVM implementations that adhere to the specification now exist.

1.1.2 The JVM Class File

The user interface to the JVM is through the `class` file. This `class` file has a strict format. The strictness of the format is required in part so that the JVM specification remains well defined. However, the strict format is also in place for security reasons. "Security" in this context refers to the ability for the JVM `class` file loader to "verify" the `class` file. Verification allows the `class` loader to make some assurances to the system running the JVM that the given class will not attempt to over-step its bounds. Details about security and the verification process are discussed in Section 1.1.4.

Each JVM `class` file contains:

- a constant pool for constants and literals used by the class.
- a list of static and non-static fields in the class (and associated flags).
- methods in the class (with flags and function signatures).
- the superclass of the current class and any interfaces this class implements.
- code segments that implement this class' methods.

The JVM `class` file is finely tuned to the Java language. However, most object oriented languages (and even most functional and imperative languages) can be modeled to fit into

Instruction	Purpose
`invokevirtual`	Call a virtual method
`invokespecial`	Call a class or object initializer method
`invokestatic`	Call a static method
`new`	Create new object instances from a class
`putfield`	Set a field in an object

Figure 1.1: Object Oriented JVM Instructions

this relatively simple object oriented model. Thus, the close relationship of Java and the JVM does not pose a particular impediment to porting non-Java languages to the JVM.

It should be duly noted, however, that strong typing is enforced for the methods and fields of a `class` file. This can cause problems when porting more weakly typed languages to the JVM. Section 1.1.4 discusses this typing enforcement. Section 1.3 discusses how this issue impacts porting non-Java languages to the JVM.

1.1.3 Code Segments in the JVM

The code segments in the `class` file use the JVM instruction set. This instruction set includes commands typically found in most stack-based assemblers. Such commands including floating point, integer, and pointer operations, as well as common control instructions. However, the key difference between the JVM and a typical assembler is that additional instructions, specific to object oriented interfaces, are included. Figure 1.1 lists a few such instructions.

To pass arguments to method calls, an operand stack is used. Items are pushed onto the operand stack before a method is `invoke`d. Upon completion, the return value of the method is left on the operand stack. The JVM verifier, discussed in the next section, prohibits using the operand stack for any other purpose.

1.1.4 Bytecode Verification and Security

In most cases, all classes loaded by a JVM are subject to a process called "bytecode verification". This verification is done to ensure that classes do not perform operations that may cause the JVM to crash. The verification process checks that all arguments on the operand stack are legal. In addition, the verifier ensures that all types of all variables passed to methods are correct, and that all load and store operations have correct types.

To accomplish this task, the verifier must do extensive analysis on the JVM code. This analysis is briefly summarized in [17, pages 128–130]. The analysis is discussed in detail in [11, 12, 13]. The details of this verification analysis are somewhat beyond the scope of this thesis; however, details are mentioned later in those cases that impact this work directly.

Of course, this verification analysis introduces overhead. The argument is made, though, that this overhead is worthwhile, since JVM code execution can afterwards proceed more quickly without checking types and stack limits. For such quick execution to take place, the JVM must assume that the verifier has checked all these parameters. In addition, the execution process must assume that verification rejects invalid code constructed with malicious intent.

However, bytecode verification is not without disadvantages. The verifier can limit some otherwise valid uses of the JVM. In fact, the first approach used in this project was impeded (and eventually rendered futile) because code was generated that could not pass the verification. Section 4.5 discusses this problem in detail.

Note that JVM bytecode verification is only a subset of those checks performed when an "applet" is loaded from the network. In those cases, more extensive checks are done to ensure that access to the local operating system is severely limited.

1.2 Why Port Non-Java Languages to the JVM?

The JVM was designed primarily with the Java language in mind. Why, then, is it useful to port languages other than Java to the JVM? In this section, a number of arguments are presented to answer that question.

1.2.1 Hardware JVMs

Traditionally, JVM implementations have been done in software. However, implementation of JVM environments in hardware devices has become both a focus of intense research and a physical reality.

Many researchers have studied the problem of implementing JVMs in hardware [21, 8] with promising results. Supporting the JVM directly in hardware is an inevitability and will likely become quite common. One researcher even built a prototype hardware JVM implementation [9].

Meanwhile, in the commercial world, SUN has already licensed picoJava [22], a central component core for a JVM microprocessor, to a number of companies. According to SUN [25], these companies plan to incorporate this hardware technology into future products.

Thus, it is very likely that hardware devices with native JVM implementations will be common in the years to come. Developers should not be limited to only the Java programming language when writing software for these hardware devices. Porting non-Java languages to the JVM will provide choice to these developers, so they can choose the best language for the job. Such choice will remove the common limitations imposed by engineering or marketing requirements that demand a particular hardware device.

1.2.2 Embedded Software JVMs

While hardware JVMs are a look to the future, embedded software JVMs are already common. Since the early days of the Java environment, various web browsers have included embedded JVMs. These embedded JVMs allow web site designers to develop "applets" that will run on the client computer. Web browsers' embedded software JVMs provide a simple, cross-platform application delivery system that has become standard in many communities.

However, in an effort to provide some security, strict controls are placed on JVM applets. These restrictions are even more strict than those enforced by the verification process (see Section 1.1.4). These additional restrictions make it relatively impossible for one to write applets (or portions of applets) in a language that has not been natively ported to the JVM.

For example, the simplest way to interface non-Java languages to the JVM is via the Java Native Interface (JNI), which allows system level programs (in C) to access the JVM. (Section 2.1 discusses this process in detail.) However, JNI access is not permitted by applets. Thus, even though the user might have an interpreter or a compiler installed for a given non-Java language, an applet from a web site cannot take advantage of that fact. Therefore, native JVM ports of non-Java languages are needed in this case.

Web browsers are indeed an important environment for embedded software JVMs. However, web browsers are not the only place one finds JVMs embedded in software. Recently, an eight-bit software JVM was released [23]. This software will give new life to eight-bit processors, allowing a modern language (i.e., Java) to run on these smaller processors. Native ports of non-Java languages will give developers a chance to target additional languages to such systems.

Finally, some personal digital assistants (PDAs), such as PocketLinux [28] have chosen to provide the developer interface via a JVM. Since the JVM is the developer interface, most software for these PDAs is written in Java. However, native ports to the JVM will allow developers to write software for these PDAs easily in other languages. This capability will make such PDAs more useful to both software developers and users.

1.2.3 Language Integration via the JVM

While all these reasons for porting non-Java languages to the JVM are compelling, the most interesting reason is less obvious. Even though the JVM is specifically tuned for the Java programming language, it still can be used as a generalized object model. Nearly all popular object oriented features are supported on the JVM, or via its standard supporting libraries. Thus, if a JVM compiler for a non-Java language is properly aware of the object oriented features of the JVM, the JVM itself can be used as a generalized object model.

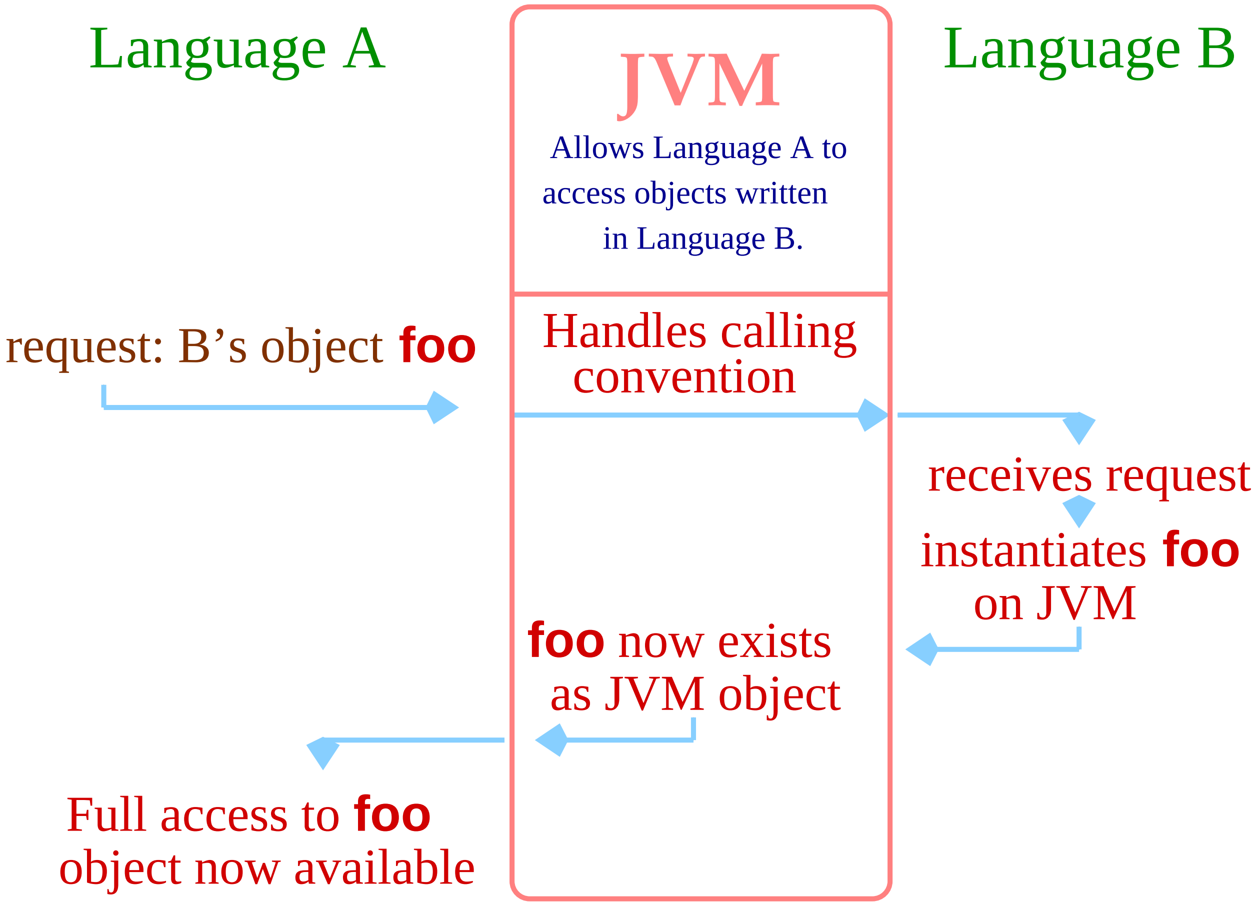

Figure 1.2: Diagram of Language Integration via the JVM

Such a generalized object model can provide a tightly coupled way for object sharing between different languages. Figure 1.2 shows generally how this might be accomplished. If both Language A's and Language B's JVM ports are aware of the JVM object model, both will access JVM objects as if they were native objects. The JVM would serve objects to the languages, and each language would use its own natural way to access objects—possibly not even aware that the objects were implemented in another language.

Some projects already exist that are beginning to successfully use the JVM in this way. For example, the JEmacs project [7] uses the JVM as a central component of an Emacs implementation. Emacs can be configured and scripted in both Scheme and Java. Work to support Emacs Lisp and Common Lisp is already underway.

Thus, the JVM can be used as the central engine for an application. If many languages have been ported to the JVM, in a manner that makes the language aware of the JVM's object model, one can design an application that allows users to script and configure the application with all supported languages. This approach is likely one of the most compelling reasons to port non-Java languages to the JVM.

1.2.4 The .NET Factor

It has been argued [27] that Microsoft's new .NET architecture [20] is better designed to be a generalized, cross-language object model. Such arguments often claim the death of the JVM as a cross-platform architecture, and look to Microsoft's technology as the future.

The .NET system indeed differs from the JVM. The .NET system was designed with language integration in mind. However, the places where .NET has particular advantages over the JVM mostly amount to mere conveniences [29]. While the designers of the JVM did not make cross-language features a priority, it was clearly on their mind [17, page 3]. To dismiss the JVM merely because it is not a perfect solution seems short-sighted.

In addition, the JVM and its related libraries have had over five years to mature. Indeed, in many circles, the JVM has just finally reached the point where it is considered a stable platform. Giving up and jumping to the latest and greatest technology will likely lead to more waiting for stability.

There is also little reason to wait. Already, dozens of languages have been ported to the JVM [26], and many of them do use the JVM in a way that is aware of its object model. Some work remains to integrate these ports, but that is not the hardest part of the job. The hard part of porting these languages to the JVM is done.

By comparison, only two or three programming languages have been ported to .NET, and all ports were funded by Microsoft itself. Not only that, none of the .NET system is available as free software, while there are a number of free software implementations of the JVM. Also, many of the programming language ports to the JVM are also free software. The JVM and ports to it are an open, established architecture. Thus the JVM, while not perfect, is currently preferable to the .NET system.

1.3 Porting Challenges

While the JVM definitely works well as a language-neutral environment, the task of porting new languages to the JVM is not without challenges. In this section, general challenges as well as those specific to Perl are discussed.

1.3.1 General Challenges

Porting a new language to the JVM is a bit more complicated than the typical problem of targeting to a new chip-set. The JVM bytecode is at a higher level than most assemblers, and thus more care must be taken when generating code.

For example, for a JVM port to be effective, it must emit code that can pass the bytecode verification process. Proper bytecode verification is more than just a rubber-stamp; some JVMs simply reject code that cannot be verified. Other JVMs might support code without verification, but JVMs are rarely regression tested with unverified code. Bugs are commonly found in JVMs when unverifiable code is run. Finally, there is the problem that unverifiable code can never run as applets.

Therefore, any JVM code emitted by a compiler must be verifiable. To emit verifiable code, it usually means that some operations that *seem* completely legal given the JVM specification are not possible. Therefore, a compiler writer must not only be intimately familiar with the JVM specification, but must also be aware of the specific expectations of the verification process. (Section 4.5 describes trouble encountered in this work with the JVM bytecode verifier.)

Another challenge faced when porting to the JVM is its object oriented nature. First, the compiler writer must find a way to reasonably map a particular language structure onto the JVM concept of a `class`. Most languages have a comparable structure to a `class` (e.g., `lambda` in Scheme, or `package`s in Perl). However, the details are sometimes tricky. For example, in Scheme, generating a class for every single `lambda` in a program can be a performance hit, if not done with great care.

Finally, for a JVM port to be as useful as possible, the compiler must generate code that is aware of the JVM object model. If this is not done, the port will likely be functional, but will not reap the advantages discussed in Section 1.2.3. This issue represents the biggest challenge of a JVM port, particularly when integration with other non-Java languages is desired. Clear object interfaces must be defined for inter-language access to be successful.

1.3.2 Perl-Specific Challenges

In addition to the general challenges that the JVM presents inherent in its nature, there are also some challenges that are specific to porting Perl to the JVM. Perl is a unique language;

This program:

```
package AlwaysOne;
sub FETCH { return 1;}
sub STORE { print "warning:  cannot change this variable\n"; }
sub TIESCALAR { my($c, $r) = @_; return bless \$r, $c; }
package main;
tie $x, AlwaysOne;
print "$x\n"; $x = "Hello World\n"; print "$x\n";
```

generates the output:

```
1
warning:  cannot change this variable
1
```

Figure 1.3: A Simple Example of a Tied Perl Scalar

Perl has both cultural and technical aspects that make it different from other programming languages. These issues create some specific challenges in porting Perl to the JVM.

First, parsing Perl is a particularly difficult problem. The Perl language is designed with lots of syntactic sugar and special cases. There are reasonable and laudable design goals (which are beyond the scope of this thesis) for this design. Regardless, though, this design makes Perl particularly difficult to parse. Currently, the only known full parser for Perl is the one that ships with the `perl` distribution, and it is infamously very complex.

There is also no formal specification for Perl. While this will likely change for the next major version of Perl, the current mantra of the Perl community is "the implementation is the specification"—meaning that whatever `perl` does defines what the language Perl itself should always do. Implementing a new compiler, given that no specification exists, is particularly difficult.

Finally, Perl's native data types are particularly uncommon. While on the surface, the native data types (array, hash, and scalar) seem common, there are some frequently used special cases that introduce complexity for a compiler writer. For example, Perl's `tie` feature allows the programmer to arbitrarily make any variable in the language a full-fledged object, where arbitrary code is executed for any variable access. Figure 1.3 shows a simple example of a tied Perl scalar.

In this example, the scalar variable, `$x`, is tied to the package `AlwaysOne`. The example shows how the semantics of variable assignment and variable use are changed utterly by `tie`. The `FETCH` and `STORE` methods replace assignment and use of `$x`.

As can be seen through the example of `tie`, Perl's data types, while seemingly straightforward at first glance, are actually quite advanced and complex. A JVM port of Perl must handle these special cases. Care must be taken so that the underlying data structures used to implement Perl's data types can support these advanced features.

Chapter 2

Possible Approaches

This chapter discusses possible approaches for porting Perl to the JVM. These approaches are not necessarily specific to Perl, and when appropriate, examples are given of other languages for which JVM ports of a given nature have been attempted. In addition, advantages and disadvantages to each approach are presented.

2.1 JNI

Even though it does not actually accomplish a JVM port, the simplest method for integrating a non-Java language with the JVM is worth noting here. This simple method is accomplished via the Java Native Interface (JNI), which provides native operating system access to the JVM.

JNI is primarily an interface between the C and Java languages. JNI provides a C API, so that C programmers can access objects and methods that exist on the JVM. In turn, Java methods can be implemented in C.

The JNI is not even close to a C port to the JVM. It merely provides, in those specific circumstances where a JVM is run as a process on a full-fledged operating system, the ability to execute calls between C and Java. Thus, the JNI has the serious downside that it cannot be used directly to gain the advantages discussed in Sections 1.2.1 and 1.2.2. Namely, JNI-based systems cannot be used in embedded software and hardware JVMs. In addition, while the JNI gives relatively complete access to Java from C, the access from Java to C is limited mostly to writing Java methods in C.

Thus, the JNI does not bring us particularly close to a native JVM port. However, the JNI is worth mentioning here because it shows the first step in language integration between Java and non-Java languages. In fact, when a particular language is already implemented in C, the JNI can be used to provide some rudimentary integration between that particular language and Java.

For example, the Java-Perl Lingo (JPL) used the JNI to provide such access between Perl and Java. The JPL is part of the core `perl` distribution and eases the integration of Java and Perl. As Larry Wall frequently points out, the JPL at least shows that Java's and Perl's semantics are compatible [31]. The JPL is indeed useful to programmers who have access to both `perl` and a Java environment on a single operating system. The JPL can be used to write some Java methods in Perl, instantiate Perl objects in Java, and/or instantiate Java objects in Perl. To provide these features, the developers of the JPL used the JNI to interface `perl` (which is written in C) with the Java environment.

However, such a solution can never be developed into a full Perl port to the JVM. The JNI is usually not available on JVMs that are embedded in hardware or other software programs. Thus, while the JPL is a powerful tool for those who use Java and Perl on a system where `perl` already runs, it will never allow Perl to run on embedded JVMs.

In addition, the JPL has some overhead. The running process of a JPL program must have an active instance of `libperl.so` (the `perl` shared library) to run the Perl code. In addition, the process must have an active instance of a JVM to run the Java code. A native Perl port to the JVM would run faster, since the Perl code would run completely independently of `perl`. Also, a full JVM port would enable Perl to take advantage of advances in the state of the art of JVM optimization technology.

Finally, the JPL really serves a different problem space than a Perl port to the JVM. The JPL seeks to maximize flexibility of transition between Java code and Perl code. Even though a full Perl port to the JVM would seek to eventually provide such flexibility, that is only a small part of the picture. The goal instead is to run nearly any arbitrary Perl program natively in the JVM environment by generating a valid JVM `class` file that is the equivalent to the given Perl program.

This is not to say that the JPL is not useful as we pursue a complete port of Perl to the JVM. Indeed, the solution discussed in Chapter 5 requires the JPL to run, since Java and Perl code must call each other.

2.2 Survey of Approaches

Since the JNI is not a clear path to a full JVM port, other methods must be considered. Traditionally, there have been four ways in which languages are ported to the JVM [1].

The traditional approaches are as follows:

- Implementation of a language interpreter in Java.
- Compilation from the source language to Java source.
- Direct compilation from the source language to JVM bytecode.

[1] These categories were originally classified in [4], but are extended a bit here.

- Mapping of language structures and idioms onto the JVM.

In the subsequent subsections, each of these approaches is considered in detail.

2.2.1 Implementation of a Language Interpreter in Java

For most languages, implementation of a language interpreter in Java is perhaps the most straight-forward method of porting that language to the JVM. This approach was used by the Tcl [16] and Python [14] ports. Since there are number of different compilers that can convert Java source into JVM bytecode, an interpreter written in Java can run easily on the JVM.

When a program from the source language needs to be run on the JVM, this new interpreter must take as input the source program, as well as the input that the source program is expecting. An `eval` construct using these two input sets is then invoked, and in that manner, the source program is run.

This approach has a number of advantages. First, if the source language has a well-written specification, or is a language with few constructs, based around a single paradigm (e.g., a relatively pure object oriented or functional paradigm), then implementing an interpreter for the language is often a simple matter of implementing the specification. Design issues are often already decided by the specification or by the paradigm, greatly easing the burden on the implementor.

A second advantage is that real-time, on-the-fly code evaluation (i.e., `eval($string)`) is always available. The Java program that implements the interpreter simply needs to instantiate a new instance of the interpreter, and feed it `$string` as input.

However, this approach has two disadvantages, one of which is particularly problematic for a Perl port. The first disadvantage is speed. Since hardware devices that have JVMs on a chip are only a subset of the useful deployments of the JVM, considerations for JVM implementations in software are important. When a JVM is implemented in software, JVM bytecodes are typically interpreted by this software. Thus, as Per Bothner notes, "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" [4]. Such a situation is unacceptable for Perl, which has always prided itself on speed.

Another disadvantage that might be acceptable for some languages, but is completely unacceptable for Perl is code divergence. If a language has a well-defined specification that describes precisely the syntax and semantics of the language, code divergence is not an issue. An implementation must adhere to the specification. However, it has often been noted in the Perl community that "the specification is the implementation". The community cannot tolerate divergent implementations. Indeed, much work in the mid-1990s was done to stop the divergence of the Microsoft and Unix-like Perl implementations.

While the Perl community has plans to change this approach, by developing a language specification for newer versions of Perl, this work is still speculative. In addition, a good port of Perl to the JVM should support older versions as well as newer ones.

Therefore, if this interpreter approach were to be taken for current versions of Perl, it would require compiling `perl`, the existing C implementation of Perl, with a C compiler targeted to JVM. Experimental compilers of this nature do exist [30], but they are far from ready for production. In addition, such a port of Perl would undoubtedly be slower than any of the other approaches. Indeed, given the relatively large size of `perl`, such a port would most likely be completely inappropriate for JVM implementations embedded in small hardware devices or those embedded in larger software programs.

Therefore, simply waiting for a C compiler to be targeted to the JVM is not a reasonable approach for porting Perl to the JVM. Other methods must be investigated and attempted.

2.2.2 Compilation from Source Language to Java

Compilation of the source language into Java source code is a possible approach for porting to the JVM. As was mentioned above, compilers that target Java source to the JVM are widely available. If, for every program in the source language, an equivalent program in Java were constructed automatically, then the source language would be effectively ported to the JVM.

The only real advantage to this approach is that the porter need not be concerned with the inner workings of the JVM. This minor advantage does not outweigh the two grave disadvantages. First, the port becomes immediately susceptible to changes in the Java language and its accompanying class libraries, which are more subject to change than the JVM specification. Second, the Java source language is not as expressive as JVM bytecodes. Although Java source is very close to JVM bytecodes, there are constructs (such as `goto`) that exist on the JVM but do not exist in Java [4].

With these disadvantages and only one minor advantage, it is not surprising that there has yet to be any language successfully ported to the JVM using this method. It should be noted, however, that one attempt was made to port Perl to the JVM in this manner. This port worked only for a very small subset of Perl. The designers of this system even note themselves that "to provide a complete translation would require many Java classes to be written, possibly making a bytecode-to-bytecode translation more effective" [18]. Despite that this system does support a subset of Perl, the conventional wisdom in both the Perl and JVM porting communities is that source language to Java source translation is not feasible to port more than small subsets of given languages.

2.2.3 Direct Compilation to JVM Bytecode

The next approach is perhaps the most traditional one: to provide a compiler that targets the source language directly to the JVM. This can be done either by writing a compiler from scratch (as was done for Scheme [5] and (of course) Java), or by retargeting an existing compiler to the JVM (as was done for Eiffel).

The former method will cause code divergence, which is appropriate for Scheme and Java, since these languages have detailed written specifications. As has been established, such code divergence is not reasonable for Perl, at least not at the current time. The later method of retargeting a compiler to the JVM is reasonable for Perl, yet there is a risk.

As was discussed in Section 1.2.3, a useful feature of a proper JVM port is to permit the source language to use the JVM to communicate smoothly with other languages. When the Eiffel compiler was retargeted to the JVM, the port was not made "aware" of the JVM's object model. Thus, it has been a difficult road to use Eiffel's JVM port to integrate Eiffel and Java, since the compiler treats JVM bytecode as just another assembler syntax—not as a rich object architecture. Plans to modify the Eiffel port to support integration with Java exist; however, the design of Perl's port to the JVM must not inherently contain this limitation.

2.2.4 Mapping Language Idioms onto the JVM

The final approach is perhaps the most uncommon, and the most limited. This approach maps each language idiom onto the Java architecture. If all the language features have semantic equivalents in Java or directly on the JVM, a mapping can be done to allow the language to run on the JVM. ADA's port to the JVM relied heavily on this approach [10].

This method, of course, must usually be combined with some Java or JVM code generation to be completely successful. However, it is worth categorizing this approach separately, since when used with a retargeted compiler, the problem that the initial Eiffel port encountered is avoided.

The downside to this approach is that each language idiom *must* have an equivalent in Java or on the JVM. If more than a few such idioms do not have equivalents, then a programmer must construct such idioms, usually by implementing them from scratch in Java. This is not unreasonable, but it does take time and effort. Therefore, this approach (when used exclusively) will likely only be successful with languages that are very much like Java—relatively strongly typed, and primarily object oriented.

2.3 Which Approach for Perl?

Given the various categories of JVM ports, the key question is which method should be used for Perl. In this work, we describe two approaches that were attempted. Both approaches attempt to leverage the existing `perl` compiler (which is discussed in detail in Chapter 3).

The first approach was roughly a direct compilation method—retargeting `perl` to the JVM via the Jasmin assembler. Chapter 4 discusses this effort, and why it ultimately failed.

The second approach is a somewhat novel approach. A "middle layer" intermediate representation provided by newer versions of Kawa [5] is used to mitigate the problems found in the first approach. Chapter 5 discusses this effort.

Chapter 3

Internals of `perl`

Before the specific approaches for porting Perl to the JVM can be discussed, a brief digression is necessary. To understand the approaches taken in this work, some understanding of the internals of the canonical Perl implementation, `perl`, must be understood. This chapter presents these `perl` internals and compares them to the JVM.

3.1 `perl` As a Compiler and Virtual Machine

It is a common misconception that `perl` is an interpreter for Perl. The misconception arises from the fact that `perl` has two components within the same actual binary. First, `perl` has a front-end compiler which includes a lexer and parser that analyzes a Perl program and produces an intermediate representation (IR) of the program, in the form of a syntax tree. Such an environment is typical of a compiler front-end as discussed in [2].

Second, `perl` has a back-end, which includes an implementation of the native Perl data types (such as scalar, array, and hash), as well as the Perl Virtual Machine (PVM). The PVM can take the IR generated by `perl`'s front-end, along with the data type implementations, and evaluate the IR (thus executing the code given by the Perl programmer). Thus, `perl` is not an interpreter. Instead, `perl` is actually a combination of a compiler and a virtual machine for Perl.

When seen in this fashion, the similarities between the `perl` environment and the Java environment are striking. However, there are some key differences.

Those differences are as follows:

- The JVM has a detailed written specification. The PVM is documented primarily only in the source code for `perl` itself.

- The JVM has fewer operation codes (OP-codes) than the PVM. Indeed, the PVM has a separate OP-code for nearly all of the over 200 Perl builtin functions. Overall, there are a 346 different OP-codes in `perl` [1].

- The JVM has very simple native data types, and relies on standard class libraries to provide more complex types. The PVM has a number of complex native data types (e.g., hash, scalar and array).

- Java compilers and JVMs are usually implemented separately. The PVM and the front-end compiler are tightly coupled inside `perl`.

These differences are really drawbacks of using `perl` as a compiler for a new virtual machine environment. First, the lack of a written specification for the PVM leads to a tendency for changes to occur in the PVM that are only made aware to core `perl` developers. The development model is open, of course, but keeping up with the details of the development is a big job, regardless.

The sheer size of the PVM makes it somewhat unwieldy. Not only are there 346 OP-codes but most OP-codes have a number of flags and options. These options and flags change the semantics of how the OP-code is evaluated. Many times, the only way to truly understand how a given OP-code works requires tracing through the source code of `perl`.

Since the source code *is* available and unencumbered, these problems can be mitigated simply by reading the source and becoming familiar with the system. However, the tightly coupled nature of the front-end compiler, its IR and the PVM creates additional problems that are more difficult to overcome when porting to new virtual machines.

The IR generated by `perl`'s front-end compiler relies on a number of complex data structures to represent Perl's native data types. Since `perl` has always lived in the same binary as the PVM, the PVM assumes those data structures are available[1]. To perform a direct translation to a new virtual machine using `perl`'s IR, equivalent data structures must be developed on that new virtual machine. As it turns out, these data structures constitute much of the semantics of Perl. Namely, nearly all variable accesses in Perl go through these data structures. Thus, using the IR to implement a port to the JVM is feasible, but a challenge.

3.2 The `perl` Intermediate Representation

While the tight coupling of the PVM and `perl`'s front-end does cause some problems, there is still a clearly defined intermediate representation (IR) that is generated by `perl`'s front-end. This section discusses that IR.

[1] There is no special reason that two systems living in the same binary must be tightly coupled. However, there is a tendency by designers to use what is available in code that is linked within the whole system. The Java environment, by designing the compiler and the virtual machine as separate entities, never fell victim to this psychological factor.

```
defined $foo;
```

Figure 3.1: "Defined" Perl Program Example

The `perl` IR consists of a parse tree. In the context of `perl`, this IR is commonly referred as the "OP-tree". This OP-tree is an acyclic directed graph representing the flow of the Perl program. There are twelve different OP classes, and a total of 346 different OPs.

Each OP has a number of flags and options. These flags and options control the behavior of the OP. Care must be taken when using the IR, as these options and flags can often change the semantics of the OP-tree evaluation.

Certain types of OPs also have additional fields that might refer to other OPs, or to internal `perl` data structures. For example, the LISTOP, which is used to group other OPs into a list, contains a field for its child OPs. The SVOP, which is used to refer to a scalar variable, contains a field that points to `perl`'s internal representation of that scalar variable.

The next two subsections contain examples of Perl programs, and their equivalent OP-trees.

3.2.1 The "Defined" Example

In this example, we consider a very simple Perl program that tests whether or not a variable is defined. This simple program can be found in Figure 3.1.

This program tests the scalar variable `$foo` to see if it is defined. In this example, the test would evaluate to false, since the `defined` test occurs before anything has been assigned to `$foo`.

Figure 3.2 contains the OP-tree of this program. The top-level OP is the "leave" LISTOP. This OP is always the final OP evaluated by the PVM, and is the parent of the rest of the program.

The "enter" OP simply notes that a program has been entered. The "nextstate" OP indicates that the evaluation will continue on to another statement in the program.

The "defined" OP is UNOP, or unary operator. This means that "defined" takes one argument. That argument is a child OP of "defined". That child OP is an SVOP, or scalar operator. The type of the SVOP is "gvsv", which means it refers to a global scalar variable. The data attached to the SVOP is what is called a "typeglob", an internal `perl` data structure that is used to refer to a symbol table entry for `foo`. That symbol table entry can in turn access the scalar entity, `$foo`.

This example is indeed quite trivial. The next section discusses a slightly more complex example.

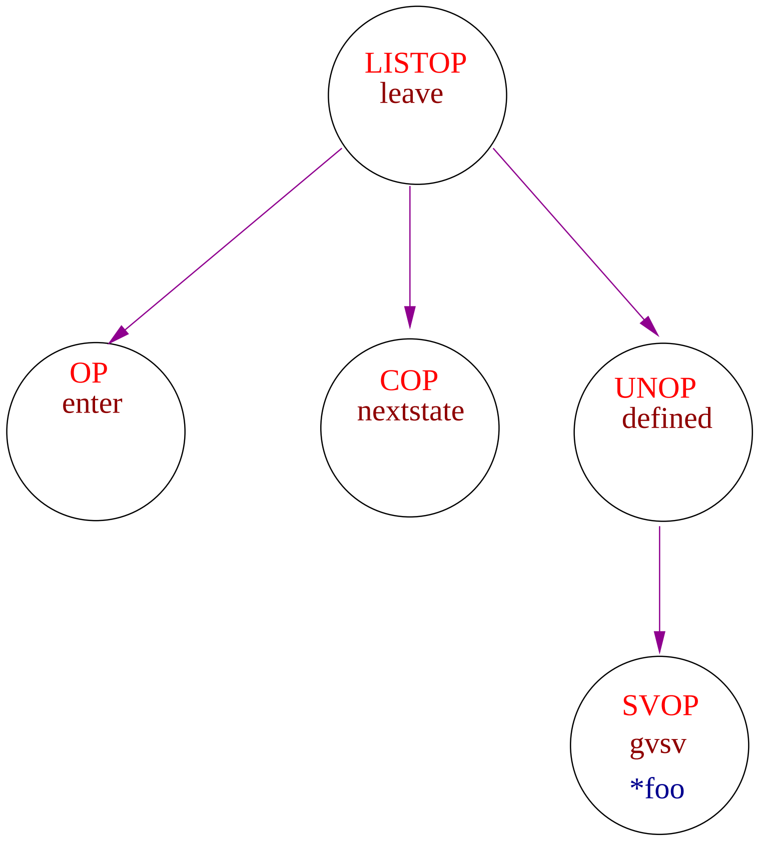

Figure 3.2: OP-Tree of “Defined” Example

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

Figure 3.3: "Add and Print" Perl Program Example

3.2.2 The "Add and Print" Example

Figure 3.3 shows a more complex example. This example has variable use and assignment, as well as a `print` statement. The OP-tree diagram of this program is in Figure 3.4.

This example introduces the two BINOPs (or binary OPs)—"sassign" and "concat". The "sassign" BINOP is used for variable assignment. Note that there is a dummy "null" UNOP[2] above the SVOP that is used to pass extra options to the "gvsv" SVOP. In this case, these options denote that the results of the "gvsv" OP will be used as an l-value.

Another OP that is introduced in this example is the "pushmark" OP. This OP refers directly to an operation on the PVM run-time stack. When the PVM evaluates the OP-tree, the PVM run-time stack is used to hold OP evaluation results as they are built during evaluation. In this example, the "print" LISTOP expects to use the PVM run-time stack to keep track of the arguments to be printed.

Thus, the "print" LISTOP first performs a "pushmark" to note where its arguments begin on the stack. After the PVM has evaluated the other child OPs of "print", "print" finds its mark on the stack, and prints all arguments after the mark.

This behavior can be demonstrated by using `perl`'s debugging command-line options, `-Dts`. Figure 3.5 shows this debugging output for the "print" operation. (The mark is represented in this output by an `*`. Note, too, that PV and NV refer to the `perl` internal string and number types, respectively.)

The figure shows that after the evaluation of "print", all arguments back to and including the mark have been consumed (and the side effect of printing those items to the standard output occurs, of course). "print" leaves a single item on the stack, a scalar value of "true" (called `SV_YES` internally by `perl`).

As that `SV_YES` shows, it is not only those OPs that perform a "pushmark" that use the run-time stack. Most OPs, when evaluated by the PVM, leave some items on the stack. In fact, one of the purposes of the "nextstate" OP is to clear the stack of unconsumed items.

It should be noted as well that some other OPs, such as "concat", use the stack for multiple items, even though they do not perform a "pushmark". In those cases, the OP defines a constant number of stack items it will consume. For example, BINOPs like "concat" always consume exactly two items.

[2] In other cases, "null" OPs are also inserted, but there presence is not particularly interesting in those cases, and they are omitted from Figure 3.4 for readability.

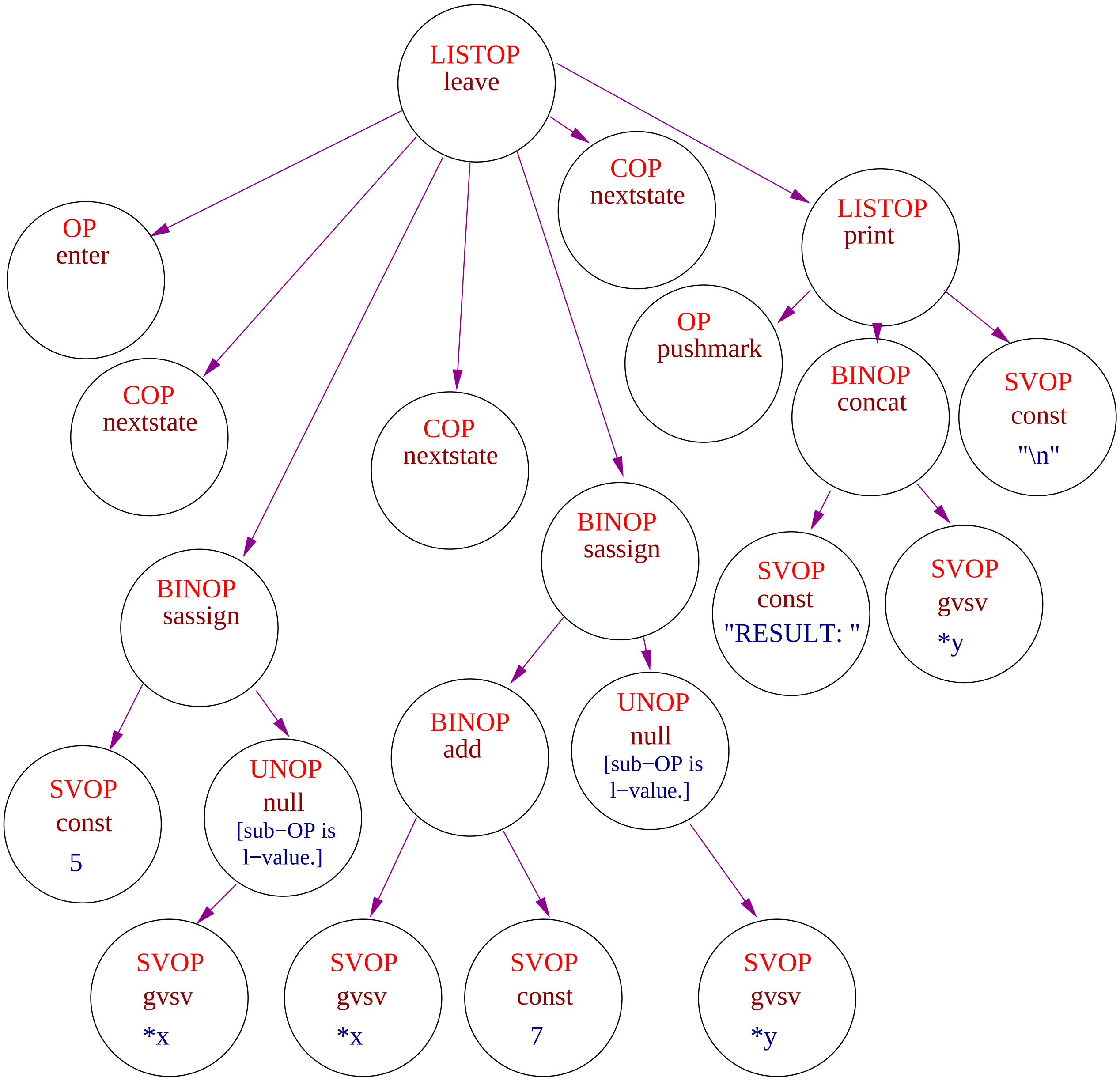

Figure 3.4: OP-Tree of “Add and Print” Example

OP	pushmark			
STACK	⟹	*		
OP	const(PV("RESULT: "))			
STACK	⟹	*	PV("RESULT: ")	
OP	gvsv(main::y)			
STACK	⟹	*	PV("RESULT: ")	PVNV(12)
OP	concat			
STACK	⟹	*	PV("RESULT: 12")	
OP	const(PV("\n"))			
STACK	⟹	*	PV("RESULT: 12")	PV("\n")
OP	print			
STACK	⟹	SV_YES		

Figure 3.5: Partial Stack Evaluation of "Add and Print" Example

3.3 Accessing the IR via `B`

The previous example demonstrates how closely tied the `perl` IR is to the PVM—many OPs are specifically tuned for operations on the PVM. And, given the other problems discussed in Section 3.1, it is not possible to simply "map" the PVM onto the JVM in any simple way. However, thanks to the `B` module, `perl`'s existing front-end can be used to leverage some parts of the IR for a JVM port, even if the PVM and the JVM are not really analogous.

The `B` module allows programmers to implement their own back-ends separate from the PVM. The `B` module provides facilities to examine and manipulate the IR generated by `perl`'s front-end. In addition, the `B` module can be used to examine the internal data structures used by the both the PVM and the front-end.

On the `perl` command line, the user interface to the `B` module is through the `O` module. The `O` module acts primarily as a wrapper, allowing the corresponding `B` module to be invoked. Instead of running the "default" `perl` back-end (i.e., the PVM), the user can use the `O` module to choose a completely different back-end. That back-end is implemented by the corresponding `B` module, which can be written in Perl or C. ([3] and [15] contain a more complete discussion of the `B` and `O` modules.)

The next chapter discusses the first approach to a JVM port, which made use of the `B` module and `perl`'s IR to compile to the JVM.

Chapter 4

First Approach—Direct Compilation

Thanks to `perl`'s IR and the `B` module, there is no need to parse Perl, find syntax errors, nor do any typical front-end compiler work to port Perl to the JVM. The current `perl` implementation can basically be retargeted in some way to the JVM.

The first approach sought to use these tools to directly compile from `perl`'s IR to JVM assembler. This chapter discusses that first approach in detail, and addresses the reasons that it ultimately failed.

4.1 Using Jasmin Assembler

The JVM `class` file format is somewhat complex. Directly generating such a file from a `B` module would be tricky. Unfortunately, there is no standard assembler syntax for the JVM, so there are no tools in the standard Java environment to easily generate JVM `class` files directly. Attempting to generate a JVM `class` from `B` was a focus of much attention early in this project. Generating `class` files from `B` proved very difficult, given the strict format of a JVM `class` file.

However, Brian Jepson proposed that instead of generating the JVM `class` file directly, output instead could be generated using the Jasmin assembler [15]. Jasmin assembler [19, pages 399–409] is a syntax for writing JVM `class` files that is similar to assembler formats used for physical architectures. This solution greatly reduced the problem scope, since there was now an easy way to generate JVM `class` files from `B`.

However, there was the concern that the Jasmin assembler format is not standardized; it is simply one possible format for JVM bytecode assembler. Indeed, other formats do exist and are in use. Therefore, it was imperative that a JVM port not rely on one particular assembler syntax.

To alleviate this problem, the concept of JVM "bytecode emitters" was introduced. First, a virtual base class called `B::JVM::Emit` was created. All code that must emit Java bytecode

```
my $emit = new B::JVM::Jasmin::Emit("Foo");
$emit->methodStart("main([Ljava/lang/String;)V", "static public");
$emit->dup("main([Ljava/lang/String;)V");
$emit->methodEnd("main([Ljava/lang/String;)V");
```

Figure 4.1: Example of `B::JVM::Emit` Interface

uses the interface provided by `B::JVM::Emit`, and all subclasses of `B::JVM::Emit` must provide implementations of `B::JVM::Emit`'s interface specific to that given assembler syntax.

As an example, consider the code in Figure 4.1. It creates a JVM class called `Foo`, with one static public method, `main`, whose body has a single JVM `dup` instruction.

If a standard assembler format for the JVM is ever created, one needs only implement `B::JVM::StandardAssembler::Emit` as a subclass of `B::JVM::Emit`, and change the first line in Figure 4.1 to:

```
my $emit = new B::JVM::StandardAssembler::Emit("Foo");
```

Assuming that `B::JVM::StandardAssembler::Emit` is implemented properly, the rest of the code will function properly, generating the `Foo` class as described.

Thus, creation of this abstract base class mitigated the concern that Jasmin assembler syntax is not standard. Jasmin could be used, via the abstract interface, without worry that it might be outdated and replaced eventually[1].

4.2 Data Type Support

With full access to the `perl` front-end, the `B` modules to manipulate the IR, and a code emitter object (as described in the last section) for JVM bytecode, most of the components for a Perl to JVM compiler are in place. However, recall that the IR generated by `perl` assumes that both a PVM and implementations of Perl's native data types are available. To successfully port Perl to the JVM, the data types that Perl considers native must be available on the JVM.

One approach would be to "map" all of Perl's data types onto equivalent data types already available for the JVM. Unfortunately, in most cases, this approach is not possible, since Perl's native data types are so unique. For example, at first glance, it might seem feasible to map Perl's hash onto an object of type `java.util.Hashtable`. However, Java's hash tables do not understand the concept of `tie`. Similarly, scalars cannot be mapped onto `java.lang.String`, since scalars act like numbers when they are supposed to, and Java strings do not. The uniqueness and flexibility of Perl's data types become the headache of

[1] In the end, this turned out to be a non-issue, as the solution described in Chapter 5 rendered an assembler-based bytecode emitter unnecessary.

```
class  SvBase implements Cloneable {
       boolean defined;
       SvBase() { undef(); }
       boolean isDefined() { return defined; }
       void undef() { defined = false; }
       // [...]
}
```

Figure 4.2: Portions of the `SvBase` Class

the compiler writer who wants to port Perl to an architecture where the native data types are not so unique and flexible.

Thus, for each data type that the PVM considers "native", an equivalent class for it must exist on the JVM. Since the Java language easily compiles to the JVM in an idiomatic way, these classes are implemented in Java. Each class provides the interface that Perl expects for the data type, and since the implementation is in Java, the new data type can run on the JVM.

As an example, consider the portion of the class `SvBase` seen in Figure 4.2. `SvBase` is analogous to the `perl`'s `SvNULL` object. `SvBase` provides a simple interface to a scalar variable. For example, it provides a function to test if a scalar is defined, and a function to undefine a scalar. (More complex scalar operations are implemented by subclasses of `SvBase`.)

Perl's data types are thus handled by providing a library of Java classes that are analogous to those expected by the PVM. These Java classes allow PVM operations, expected by the `perl` front-end, to be more easily mapped onto the JVM.

4.3 Putting It Together with `B`

The final step to achieve the JVM port is to support the OP-codes in the IR. In this area, the `B` module is most useful. The `B::JVM::Jasmin` module that generates the Jasmin assembler from `perl`'s IR is written as a subclass of `B`. It descends the syntax tree provided by the IR, in a depth-first fashion. For each node in the OP-tree, the module processes that node using the emitter to generate Jasmin code. The emitted Jasmin code utilizes the various data type classes to perform the task the OP-code would have performed had it been run on the PVM.

As an example, Figure 4.3 presents the code from `B::JVM::Jasmin` that handles the "gvsv" SVOP. In this code segment, we see part of the subroutine, `B::SVOP::JVMJasmin`. The name indicates that it is the code for handling SVOPs for the Jasmin-based JVM port.

```
sub  B::SVOP::JVMJasmin {
     my  $op = shift;
     my  $name = $op->name();
     #   ...
     my  $curMethod = # ...
     #   ...
     if  ($name eq "gvsv") {
         my $stashName = $op->gv->STASH->NAME();
         my $gvName = $op->gv->NAME();
         $emit->getstatic($curMethod, "Stash/DEF_STASH", "LStash;");
         $emit->ldc($curMethod, cstring $stashName);
         $emit->invokevirtual($curMethod,
                "findNamespace(Ljava/lang/String;)LStash;");
         $emit->ldc($curMethod, cstring $gvName);
         $emit->invokevirtual($curMethod,
                "Stash/findGV(Ljava/lang/String;)Linternals/GV;");
         $emit->invokevirtual($curMethod, "GV/getScalar()LScalar;");
     }
     #   ...
}
```

Figure 4.3: `B::JVM::Jasmin` Code for handling “gvsv” SVOP

The B module identifies that a given OP is an SVOP, and calls the routine. As a user of B, B::JVM::Jasmin provides the JVMJasmin portion of the name. This name is given on the command-line via the O module, so the user can indicate that the Jasmin-based JVM back-end is desired.

The first argument when OP-code subroutines, such as B::SVOP::JVMJasmin, are invoked is the object referring to the current OP-code. Usually, the name method is called to find the exact type of the OP-code, and Figure 4.3 reflects this.

However, in Figure 4.3, only the code for handling the "gvsv" is shown. The "gvsv" OP-code is used when a global scalar variable is mentioned. This OP-code must find the actual data of the variable by searching for it in the name space. To generate the equivalent Jasmin code for this OP-code, the three Java classes, Stash, GV, and Scalar, must be used. These are equivalents to stashes, GVs and SVs on the PVM [1].

If the name of the variable for the given "gvsv" SVOP is, for example, $foo, then the code in Figure 4.3 generates Jasmin assembler that looks something like this:

```
getstatic      Stash/DEF_STASH LStash;
ldc            "main"
invokevirtual  findNamespace(Ljava/lang/String;)LStash;
ldc            "foo"
invokevirtual  Stash/findGV(Ljava/lang/String;)Linternals/GV;
invokevirtual  GV/getScalar()LScalar;
```

or, as its (easier to read) Java equivalent[2]:

```
Stash.DEF_STASH.findNamespace("main").findGV("foo").getScalar();
```

If you compare this to the process described in [1] of how a stashes work inside perl, it is easy to see that this is equivalent code for a "gvsv" OP-code (given that the Stash and GV Java classes do their jobs correctly!).

4.4 The "Defined" Example with B::JVM::Jasmin

This section shows how B::JVM::Jasmin compiles the "defined" example from Figure 3.1. The Jasmin code emitted by B::JVM::Jasmin for the simple program from Figure 3.1 is as follows:

[2] B::JVM::Jasmin does *not* actually translate to Java. The Java code is provided for didactic purposes only.

```
.class          public main
.super          java/lang/Object
.method         static public main([Ljava/lang/String;)V
.var 0          is foo LSvBase
new             SvBase
dup
astore_0
dup
invokespecial SvBase/<init>()V
invokevirtual SvBase/defined()Z
```

or, as its (easier to read) Java equivalent[3]:

```java
class main {
        static public void main(String argv[]) {
                SvBase foo = new SvBase();
                bar.defined();
        }
}
```

Figure 4.4 is a graphical representation of how `B::JVM::Jasmin` emits the Jasmin code for the "defined" example. Figure 4.4 is similar to Figure 3.2, but, in addition to the OP-tree, the new figure shows when code is emitted. The down arrows denote that code is emitted as the node is entered, and the up arrows denote code is emitted when the node is exited.

4.5 Failure of this Method

This method of Jasmin code generation direct from a `B` module works well for simple examples. However, it quickly became unwieldy. Eventually, it was clear that this approach was not the best possible.

While it is basically straight-forward to add new support for additional OP-codes, the amount of work can still be large. For example, to add support for basic Perl tied variables, special Java classes would need to be developed that act much like the data structures internal to `perl`. While this can be done by reimplementing in Java what `perl` already does, such a task is quite complex, tedious and time-consuming.

In addition, emitting Jasmin assembler for even the most basic operations is tedious. For example, operations like simple conditionals needed to be carefully hand-coded and

[3] `B::JVM::Jasmin` does *not* actually translate to Java. The Java code is provided for didactic purposes only.

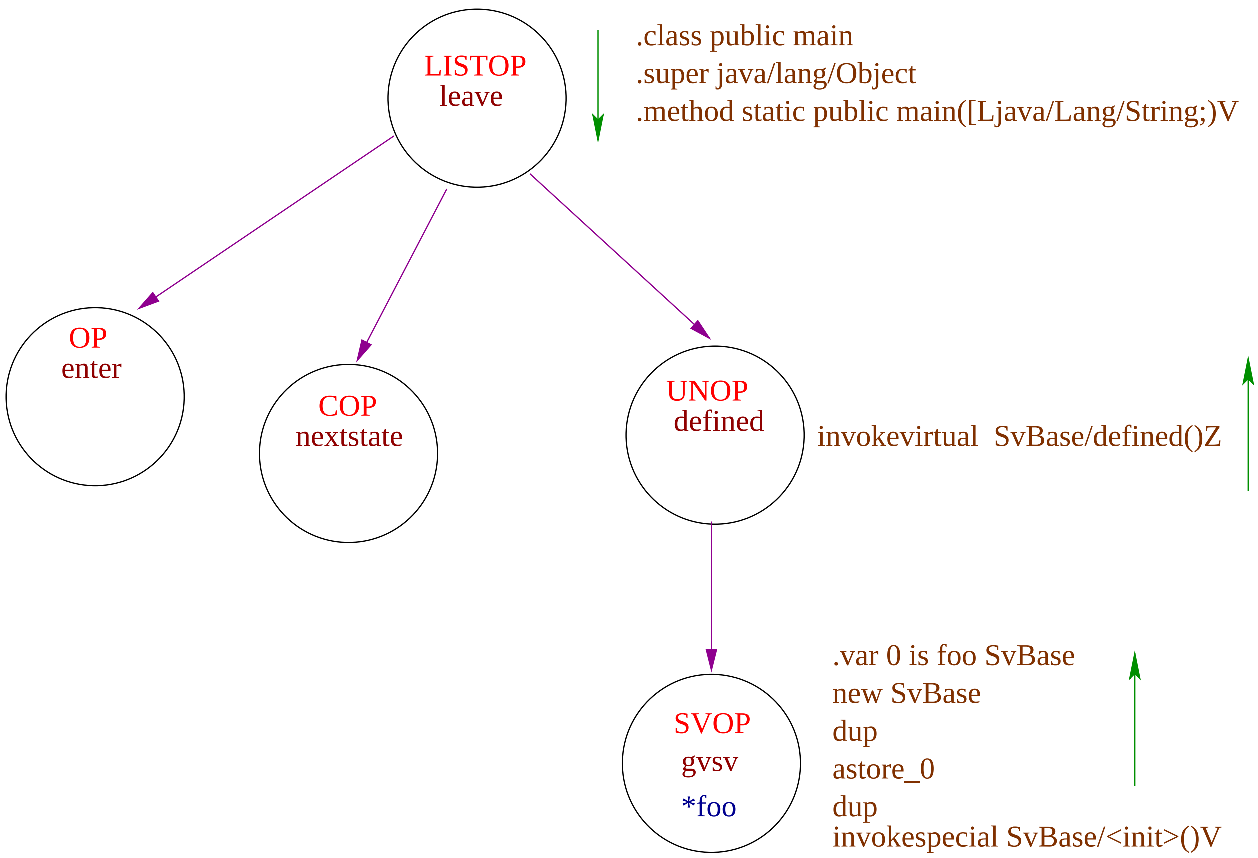

Figure 4.4: `B::JVM::Jasmin` Evaluation of "Defined" Example

debugged. In the case of conditional statements, this work took days, and most of the work was simply finding bugs in the way Jasmin code was emitted. Compiler technology has really evolved to the point where such tedious work for code generation should not be done by hand. This fact hinted that the approach described here was flawed.

Yet, the worst problem encountered with this approach, and the one that finally made it completely clear that `B::JVM::Jasmin` was on the wrong path, was dealing with bytecode verification. This problem is exemplified best by the "novel" way `B::JVM::Jasmin` handled the PVM run-time stack. It seemed that the JVM operand stack could be used to emulate the operations of PVM run-time stack—using the JVM stack to hold the "mark" and those values to be later fed to LISTOPs like "print". However, when subject to bytecode verification, JVM code of this nature was rejected, since the verifier could not determine a constant limit on the stack (i.e., the verifier cannot solve the halting problem).

Quickly, it became clear that this direct compilation was not the best approach. Either much more of the system would need to be hand-coded in Java, or excessive care would need to be taken to ensure that no generated code could possibly upset the bytecode verification process. This issue, coupled with the sheer difficulty in doing such low-level code generation, led to a search for a new approach. That new approach was quickly discovered, and is discussed in the next chapter.

Chapter 5

Second Approach—Kawa Integration

Parallel to the work described in Chapter 4, Bothner had continued to develop his Kawa system for Scheme [4, 5]. As he worked, he began to abstract some of the components of Kawa that were not Scheme-specific, and developed them into an interesting layer of abstraction. The end result was a high-level, 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 chapter discusses how that IR can be used to facilitate a better, easier and more robust port of Perl to the JVM than those methods discussed in Chapter 4.

5.1 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 [24].

As discussed in Chapter 3, `perl` does have its own IR. Chapter 4 discussed how this IR was used to generate Jasmin 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 would be 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 facilitate a port to the JVM. The solution proposed here considers using a *second* IR that is specifically designed to function with the JVM. A translator can then be 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 (i.e., `lambda`) 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 would provide facilities (better than those provided on the bare JVM) to implement those additional features.

Kawa's IR can definitely serve as this new IR. Originally designed for Scheme, Kawa's IR has been generalized to support basic generic features that are common in many dynamically typed, very high level languages. In addition, it 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.

5.2 The Kawa IR

The Kawa IR is provided by the Java package, `gnu.expr.*`. This package, described in [4, 5], 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 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, including handling of parameters, localized 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, and static and instance methods. When translating `perl`'s IR to Kawa's IR, a `ModuleExp` is used to represent each Perl `package`.

5.3 The "Add and Print" Example with Kawa

This section shows how the Perl program from Figure 3.3 (the "add and print" example) is compiled to Kawa's IR. Figure 5.1 shows a diagram representing how the "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 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 case, 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. It should be noted that while in this example, we are dealing only with simple scalars, these expressions can 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`.

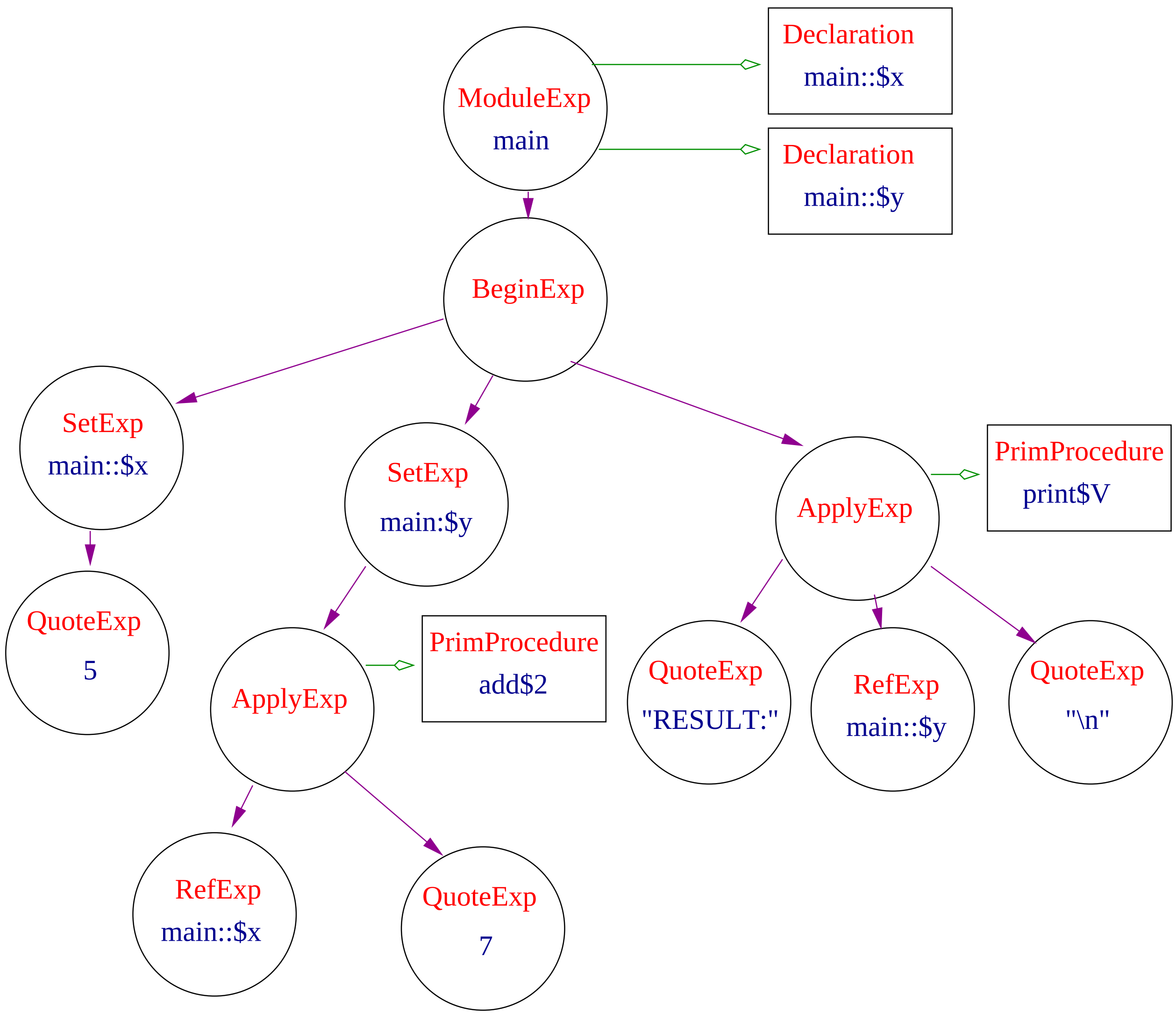

Figure 5.1: “Add and Print” Example in Kawa’s IR

Through this example, Kawa's flexibility is clear. The direct compilation method discussed in Chapter 4 failed because it attempted to compile from `perl`'s IR to a very low level JVM interface—the Jasmin assembler. The `perl` IR can be massaged much more easily into Kawa's IR, because Kawa provides higher level semantic concepts and an infrastructure for compilation.

Chapter 6

Conclusions and Future Work

Porting Perl to the JVM represents 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 thesis presented a study of possible approaches for a Perl port to the JVM.

Clearly, it made sense to mitigate the parsing issues by using `perl`'s existing front-end. Use of that front-end leveraged the many person-years that went into the 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. The first approach tried, a direct compilation method, ultimately failed. This approach relied too much on `perl`'s IR. This approach implicitly assumed that the PVM could be easily mapped onto the JVM 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 our 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. Parts of the JVM, such as the bytecode verifier, rely on assumptions that are not applicable when the JVM is used in a more general way. 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. Such integration would bring the community closer to the goals described in Section 1.2.3.

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 key impact of this work, however, is not only the final conclusion that "Kawa is the right path for JVM compilation of non-Java languages". The process of finding that solution is the greatest contribution here. This work shows the unexpected pitfalls that are encountered when direct compilation to JVM bytecode is attempted. In addition, this work exposes the problem of tightly coupling the IR with a particular virtual machine architecture. It is hoped that the Perl6 effort (which seeks to reimplement `perl` from the ground up) can make use of the lesson learned here, so that future versions of `perl` will ease the burden of porting to architectures like the JVM.

The most open area for future work is to continue porting more of Perl to the JVM via Kawa. Currently, only a small 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. With that in mind, all software developed in conjunction with this thesis has been released as free software, and contributions from the community have already begun.

Such work does not only benefit the Perl community, either. Already, the work of porting Perl to the JVM via Kawa has inspired enhancements to Kawa itself [6]. 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.

Bibliography

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

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, USA, first edition, March 1988.

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

[4] Per Bothner. "Kawa–Compiling Dynamic Languages to the Java VM". In *Proceedings of the* USENIX *1998 Annual Technical Conference: Invited Talks and FREENIX Track*, pages 225–272, New Orleans, Louisiana, USA, June 1998. Also available at `http://www.bothner.com/~per/papers/`.

[5] Per Bothner. "Kawa: Compiling Scheme to Java". In *Proceedings of the 1998 Lisp Users Conference*, Berkeley, CA, USA, November 1998. Also available at `http://www.bothner.com/~per/papers/`.

[6] Per Bothner. Personal Communication, December 2000.

[7] Per Bothner. "JEmacs: The Java/Scheme-based Emacs". In *Proceedings of the FREENIX Track of the 2000* USENIX *Annual Technical Conference*, pages 271–277, San Diego, CA, USA, June 2000. Also available at `http://jemacs.sourceforge.net/Freenix00/Freenix00.html`.

[8] Chris Cladingboel. "Hardware Compilation and the Java Virtual Machine". [Online] Available at `http://www.wadham.ox.ac.uk/~chris/project`, July 1998.

[9] Nathan Clement. "Hardware Implementation of the Java Virtual Machine". [Online] Available at `http://murray.newcastle.edu.au/users/students/1999/c9510422`, November 1999.

[10] Cyrille Comar, Gary Dismukes, and Franco Gasperoni. "Targeting GNAT to the Java Virtual Machine". In *Proceedings of the conference on TRI-Ada '97*, pages 149–161, 1997.

[11] Stéphane Doyon. "On the Security of Java: The Java Bytecode Verifier". Master's thesis, Université Laval, Sainte-Foy, Québec, Canada, April 1999.

[12] Stéphane Doyon and Mourad Debbabi. "Verifying Object Initialization in the Java Bytecode Language". In *Proceedings of the 2000 ACM Symposium on Applied Computing*, volume 2, pages 821–830, Como, Italy, March 2000.

[13] Allen Goldberg. "A Specification of Java Loading and Bytecode Verification". In *Proceedings of the Fifth ACM Conference on Computer and Communications Security*, pages 49–58, San Francisco, CA, USA, November 1998.

[14] Jim Hugunin. JPython. [Online] Available at `http://www.jpython.org`, October 1999.

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

[16] Ray Johnson. "Tcl and Java Integration". Technical report, Sun Microsystems Laboratory, February 1998. [Online] Available at `http://www.scriptics.com/products/java/tcljava.pdf`.

[17] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Mountain View, CA, USA, first edition, 1997.

[18] Raymond Mccrae, Huw Evans, and Ray Welland. "PerlCaffeine: Compiling Perl to Java". In *Proceedings of the Perl Conference 4.0*, pages 127–135, Monterey, CA, USA, July 2000.

[19] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly and Associates, Sebastopol, CA, USA, first edition, March 1997.

[20] Microsoft, Inc. "Microsoft .NET Homepage". [Online] Available at `http://www.microsoft.com/net/default.asp`, December 2000.

[21] Vijaykrishnan Narayanan. *"Issues in the Design of a Java Processor Architecture"*. PhD thesis, University of South Florida, 1998.

[22] J. Michael O'Connor and Marc Tremblay. "picoJava-1: The Java Virtual Machine in Hardware". *IEEE Micro*, 17(2):45–53, March/April 1997.

[23] OneEighty Software, Ltd. "Breakthrough Brings Java Capabilities to Eight-Bit Platforms". [Online] Press Release Available at `http://www.180sw.com/PDF/GENEVA-8bit-20001214.pdf`, December 2000.

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

[25] Sun Microsystems, Inc. "picoJava Microprocessor Core". [Online] Available at http://www.sun.com/microelectronics/picoJava, October 2000.

[26] Robert Tolksdorf. "Languages for the Java VM". [Online] Available at http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html, December 2000.

[27] Nathan Torkington. "What Every Perl Programmer Needs to Know About .NET". [Online] Available at http://www.perl.com/pub/2000/12/net.html, December 2000.

[28] Transvirtual Technologies, Inc. "PocketLinux". [Online] Available at http://www.pocketlinux.com, December 2000.

[29] Jon Udell. "Does JVM Already Deliver What .NET's CLR Promises?". [Online] Available at http://www.byte.com/column/BYT20001214S0006, December 2000.

[30] Trent Waddington, Cristina Cifuentes, and Mike Van Emmerik. "A Resourceable and Retargetable Binary Translator". [Online] Available at http://archive.csee.uq.edu.au/~csmweb/uqbt.html#gcc-jvm, December 1999.

[31] Larry Wall. Personal Communication, August 1998.