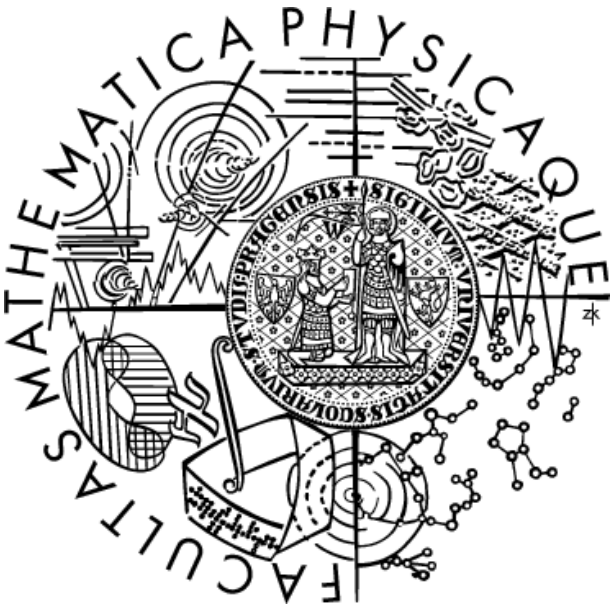
Charles University in Prague
Faculty of Mathematics and Physics

Master Thesis



David Majda

Translating Ruby to PHP

Department of Software Engineering
Supervisor: RNDr. David Bednárek
Study Program: Computer Science, Software Systems

I would like to thank my supervisor RNDr. David Bednárek for his advice, my parents for their support and Markéta for her patience.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with lending of this master thesis. This thesis may be reproduced for academic purposes.

In Prague, August 5, 2008 David Majda

Title: Translating Ruby to PHP
Author: David Majda
Department: Department of Software Engineering
Supervisor: RNDr. David Bednárek
Supervisor's e-mail address: david.bednarek@mff.cuni.cz
Abstract:

The goal of the work is to design and implement a compiler translating a significant subset of the Ruby language into PHP, with emphasis on the correct translation of dynamic and functional language elements and compatibility with the original Ruby language implementation. The work begins with an introduction of the Ruby language and an overview of its existing implementations, highlighting their interesting properties. The work then focuses on analysis of the individual language elements' properties and a description of their translation to PHP. Detailed overview of the implemented constructs and standard library elements is attached. The result of the work is practically usable compiler that can be further extended and used in the production environment after implementing remaining Ruby language elements.

Keywords: Ruby, PHP, compiler, Ruby implementation

Název práce: Překladač Ruby do PHP
Autor: David Majda
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. David Bednárek
E-mail vedoucího: david.bednarek@mff.cuni.cz
Abstrakt:

Cílem práce je návrh a implementace překladače podmnožiny jazyka Ruby do PHP. Důraz je přitom kladen na korektní překlad dynamických a funkcionálních prvků jazyka a kompatibilitu s originální implementací jazyka Ruby. Práce začíná představením jazyka Ruby a přehledem jeho existujících implementací se zdůrazněním jejich zajímavých vlastností. Těžiště práce spočívá v analýze vlastností jednotlivých jazykových elementů jazyka Ruby a popisu jejich překladu do PHP. Přiložen je podrobný přehled implementovaných konstrukcí a součástí standardní knihovny jazyka. Výsledkem práce je prakticky použitelný překladač, který může být dále rozšiřován a po případné implementaci zbývajících prvků jazyka Ruby nasazen v produkčním prostředí.

Klíčová slova: Ruby, PHP, překladač, implementace Ruby

Contents

Chapter 1

Introduction

The primary goal of this work is to design and implement a compiler translating a significant subset of the Ruby language [1] into PHP [2], with emphasis on the correct translation of dynamic and functional language elements and compatibility with the original Ruby language implementation.

The result is a practically usable compiler (available on the accompanying CD), which is able to translate simple one-file Ruby programs, supports majority of Ruby language elements and a significant part of the core class library[1] and that can be extended to support the rest. After implementing remaining parts of the Ruby language, it could potentially be used in production environment as an alternative Ruby language implementation.

To achieve the goal of writing the compiler, it was necessary to study the documentation and internals of existing Ruby implementations. The overview highlighting their interesting properties is presented in this thesis.

1.1 Motivation

Motivation for the thesis topic comes from real-world usage of the Ruby language in the Ruby on Rails framework [3]. Ruby on Rails is a tool for rapid development of the web applications. It is based on MVC pattern [4, 5], which dictates how the application should be structured, and ActiveRecord pattern [6], which abstracts the low-level database operations, resulting in simpler and faster web application development when compared to e.g. PHP. However, to run an application created in Ruby on Rails on a server, it is necessary to install support for Ruby, various libraries and tools.

Even when supported by the server, running Ruby on Rails application is not easy—typically it requires setting up a cluster of Ruby Mongrel web servers with some leading proxy server and a load balancer.[2] The cluster is required because Rails are not multi-threaded and the Ruby interpreter does not allow more Ruby contexts in one process.

[1] Detailed list of supported features is included in Appendix B.

[2] New deployment option emerged recently: Phusion Passenger [7], also known as mod_rails. This is a module for the Apache web server that simplifies running and deploying Rails applications. However at the time of writing, this option is not in widespread use yet and it does not solve all Rails deployment problems.

Thus, to serve multiple concurrent requests, multiple server processes running the same application are needed.

The requirement to run multiple processes containing the Ruby interpreter with fully loaded Rails environment has negative consequences to memory consumption (typical Mongrel process requires 20–40 MB RAM). This severely limits the number of applications that can be run concurrently on a shared server.

Complicated setup and relatively big memory consumption of Rails applications are probably the main reasons why companies providing Rails hosting are relatively rare and they have to ask for a price covering a relatively high expenses. Compare this situation with PHP, where hosting is a commodity and there are even many companies offering PHP hosting for free.

The described situation with Rails hosting means that beginners who want to try web application development will probably choose PHP over Ruby on Rails. From their point of view it is a cheaper and simpler option. They would not want to mess with complicated setup and pay for publishing of their application, when they can just write it in PHP and put it on any free-hosting simply by copying the files via FTP. Moreover, existing PHP developers who have paid for their hosting cannot use that hosting for their Rails applications.

This is the situation in which the idea of Ruby to PHP compiler was born. It would be great if the developer could write his or her application in Ruby on Rails, use some kind of tool which would translate the whole application to PHP and then upload the resulting code to any PHP web server. The application would run like any other PHP application.

The idea was indeed challenging and seemed useful enough to pursue. However it was soon realized that the implementation of the whole concept would require huge amount of work, certainly exceeding the scope of a single master thesis. But the core of the project—Ruby to PHP translator—looked like a project of suitable size, when reduced to a reasonable subset of Ruby.

1.2 Translator Architecture

The translator is divided into two main parts—the *compiler* and the *PHP runtime*.

The compiler accepts a Ruby source code on its input and produces equivalent PHP code on its output. Internally, it uses a lexer and parser to convert the Ruby source code into an abstract syntax tree. The actual conversion routines then walk around this tree and convert it into a similar tree containing nodes representing a program in PHP, which is serialized to the output.

The resulting PHP code is not standalone, but depends on supporting functionality provided by the PHP runtime. The runtime is necessary for several reasons:

1. **Ruby language complexity.** Many seemingly simple operations (such as class definition or a method call) have quite complex behavior and many side effects, unmatched by any PHP constructs. It would be inefficient and just ugly to emit PHP code implementing such operations again and again for every instance of given language element. Instead, in many places it was chosen to encapsulate particular

behavior into a helper function, which is contained in the PHP runtime. The emitted code simply calls this helper function. As a result, the translated Ruby code mostly consists of invocations of the PHP runtime functions and does not resemble the original code too much.

2. **Ruby language dynamism.** Ruby allows performing many operations at runtime (such as adding or removing methods in classes) which many languages allow only in compile time. The code implementing these operations must be available to the translated PHP program when running.

3. **Core class library.** Ruby heavily depends on its class library, even for such basic operations like raising[3] an exception or including a module in a class. Large part of this library has been implemented and its code is contained in the PHP runtime.

The runtime therefore consists mainly of the core class library implementation and the code supporting the compiler-generated code and runtime operations.

Both parts of the translator (the compiler and the PHP runtime) are described in detail in separate chapters.

1.3 Overview

The work is divided into seven chapters, beginning with the introductory Chapter 1. Chapter 2 briefly introduces the Ruby language and highlights its distinctive features, along with a summary of existing Ruby implementations and their approaches. The chosen translation approach and key decisions of the translator design are described in Chapter 3, followed by a description of the implementation specifics in Chapter 4 and Chapter 5. Related work is summarized in Chapter 6. In Chapter 7 the work is concluded and areas where the implementation could be improved and further extended are outlined.

Two appendices are included: Appendix A describes the requirements, installation and usage of the translator. Appendix B contains a detailed list of implemented features.

[3] What other languages call "throwing an exception", Ruby calls "raising an exception". The thesis sticks with this terminology.

Chapter 2

Ruby Language

Ruby is a high-level object-oriented general-purpose language with functional programming elements, dynamic and reflexive features and a rich but clean syntax. It belongs to the same language family as Python, PHP or JavaScript, but it borrows most features and syntax from Lisp, Smalltalk and Perl.

The first part of this chapter briefly describes Ruby history and features. This sets up a context for later chapters of the work so that readers unfamiliar with Ruby are able to follow.

The second part of this chapter describes Ruby implementations, namely MRI, JRuby, IronRuby, Rubinius, MacRuby, MagLev, XRuby, Cardinal, HotRuby and IronMonkey, and highlights their most interesting features.

2.1 History

The Ruby Language was born in Japan, when its creator Yukihiro Matsumoto (often called "Matz" in Ruby circles) perceived a need for a high level language, optimized for ease of use and programmer's convenience as opposed to machine performance. He began the implementation in February 1993 and first version was released in 1995 as open source software. More versions followed in next years.

The language initially took off mainly in Japan, but received a significant boost in western countries after Ruby on Rails—a framework for rapid development of web applications written in Ruby—became popular in 2006.

The original Ruby implementation was written in C and had various problems (see Section 2.5.1). Also, the interaction with other mainstream languages such as Java or C# was hard to do from Ruby. To remedy this situation, several other Ruby implementations were created, including JRuby based on JVM and IronRuby based on .NET. Gradually, some of those implementations went from toy projects to serious competitors to the official implementation. Most of them are open source software.

Evolution of Ruby still continues under the direction of Yukihiro Matsumoto, with feedback from the user community and alternative implementation creators. The development is divided between the stable 1.8 branch (version 1.8.7-p22 is the most current at the time

of writing) and experimental 1.9 branch (version 1.9.0-3 is the most current at the time of writing). The development of new and sometimes incompatible features happens on the 1.9 branch, the 1.8 versions are constrained to minor updates and backporting of compatible features from 1.9. Because of the incompatibilities between Ruby 1.8 and 1.9 and instability of the 1.9 branch, most of the world uses Ruby 1.8, with version 1.8.6-p114 often recognized as the "gold standard".

2.2 Specification

As with many languages developed as open source software, there is no specification of the Ruby grammar, behavior and functionality of its core class library. Ruby is simply a language induced by its implementation.

This poses serious problems for other implementations, which often have to resort to reverse-engineering Ruby behavior from its source code. While there exists a project [8] that aims to specify expected behavior in form of executable tests, it is quite young and the tests have not been completed yet.

Indeed, the absence of formal specification (or even a language grammar) posed significant problems in development of the translator.

2.3 Features

Ruby's most distinctive features include:

- Dynamic typing
- Smalltalk-like object system
- Blocks and functional programming support
- Introspection and metaprogramming facilities
- Text-processing features similar to Perl
- Error handling using exceptions
- Automatic memory management with garbage collection
- Built-in threading and continuation support

All these features are described in more detail in the following sections.

2.3.1 Dynamic Typing

Ruby utilizes dynamic typing, which means that the type of the variables is not determined at their declaration, but depends on the kind of values assigned to them. Values of different types can be assigned to the variable during its lifetime. Ruby variables are declared implicitly by assignment.

2.3.2 Object System

Ruby is a purely object-oriented language, which means that every value in the language is an object.

Ruby's object system is derived from the one used in Smalltalk. It is based on the idea of *message passing*, where objects receive named messages with parameters and it is up to them to respond. Classes usually define methods, which cause the instances to handle particular message, but they may use the `method_missing` catch-all method to capture all messages not handled by any method. Most language operators are implemented as methods.

The object system supports single inheritance, but allows injection of functionality into defined classes by using modules (this mechanism is often called *mixins*). This feature may be used to partially simulate multiple inheritance.

Notable feature of class and module definitions is that there can be Ruby code present inside them—the definitions are more similar to namespaces than to class definitions in common languages. This code is executed once (when the file with a class or module definition is loaded).

Ruby classes and modules are *open*: methods, constants or class variables contained in a class or a module can be added, removed or redefined any time. This includes all built-in classes, such as strings or integers. Large parts of the language semantics can be redefined any time.

2.3.3 Blocks and Functional Programming Support

Although Ruby is not a functional programming language in the strict sense, it supports some functional programming constructs.

For example, when calling a method, a programmer can pass it a parametrized anonymous block of code. The method can yield control to that block and get its return value or convert it to an object and store it to be invoked later. Blocks have closure properties.

The mechanism is similar to anonymous functions (lambda functions) in other languages and can be used e.g. to implement callbacks or comparators for sorting routines. Most often, blocks are used as internal iterators when manipulating collections, which provide support for traditional functional programming primitives such as `map`, `reduce` (called `inject` in Ruby) and `filter` (called `find_all`).

Listing 2.1 shows a simple example of blocks, where a list of squares is created, even numbers are filtered out of it, the list is then sorted and each item is printed.

Listing 2.1: Ruby program using blocks.

```
# Prints 25, 9 and 1.
(1..5)                        \
  .map    { |x| x * x }       \
  .reject { |x| x % 2 == 0 } \
  .reverse                    \
  .each   { |x| puts x }
```

2.3.4 Introspection and Metaprogramming Facilities

Ruby offers rich facilities for introspection and metaprogramming. For example, objects can be asked what class they belong to, whether they respond to a specific method or what instance variables they contain. Similarly, classes can be asked what methods or constants they define. Almost all the information carried by objects is available for inspection and change.

In the example code in Listing 2.2, a new anonymous class is created at runtime and a method is defined on it. Afterwards, a new instance of this class is created, it is queried for presence of the defined method and this method is called.

Listing 2.2: Ruby program using reflection and metaprogramming.

```
klass = Class.new
klass.send(:define_method, :greet) do |name|
  puts "Hello, #{name}."
end
instance = klass.new
puts instance.respond_to?(:greet) # Prints "true".
puts instance.greet("reader")     # Prints "Hello, reader".
```

2.3.5 Text-processing Features

Ruby contains built-in regular expression support and many convenience methods that enable it to process text easily. These features are often used in combination with Ruby functional elements.

Listing 2.3 shows an example program that finds users in `/etc/passwd` file who use bash as their shell and prints their logins, sorted alphabetically. Note how succinct and self-explaining the program is.

Listing 2.3: Ruby program using text processing features.

```
puts open("/etc/passwd").readlines.grep(/bash$/).sort.map do |line|
  line.split(":")[0]
end
```

2.3.6 Error Handling

Ruby handles errors using exceptions. Exceptions are objects of class `Exception` or some of its subclasses. Raising and catching exceptions is similar to most other common languages that use exceptions. All exceptions are unchecked (i.e. raised exceptions are not declared in the method headers and there is no obligation to catch exceptions).

2.3.7 Automatic Memory Management with Garbage Collection

Programmer in Ruby does not need to worry about memory allocation issues—all memory is allocated and freed automatically. This is typical for all dynamic languages.

In the original implementation, a conservative mark and sweep garbage collector is used, other implementations often depend on the garbage collector of the underlying platform (e.g. JVM or .NET).

2.3.8 Built-in Threading and Continuation Support

Using built-in classes `Thread` and `ThreadGroup`, Ruby offers threading support directly in the language. Creating a new thread is as easy as calling a method with a block—see Listing 2.4 for an example.

Listing 2.4: Ruby program creating a thread.

```
t = Thread.new do |thread|
  print "Hello from a new thread"
end
```

Ruby implementations differ in the implementation of threads—some use their own implementation ("green threads"), some use threading support of the underlying platform (see Section 2.5).

Ruby also supports an unusual construct—continuations. They are used very rarely in real world Ruby programs and are not included in some Ruby implementations for technical reasons.

2.4 Typical Usage

Since its creation, Ruby was most commonly used as a system scripting language in Unix environment. Its powerful text processing capabilities are useful when processing outputs of various programs or reading text files, its dynamic nature makes scripts quick to write and object-oriented programming support allows to achieve clear code organization and reuse.

In many places (especially in Japan), Ruby replaced Perl that was traditionally used in the role of a system scripting language. The advantages of Ruby over Perl are mainly much cleaner syntax and object-oriented programming support.

Ruby gained significant popularity after appearance of the *Ruby on Rails* web application framework [3]. This open source framework aims to simplify development of the web applications. It is based on MVC pattern [4, 5], which dictates how the application should be structured, and ActiveRecord pattern [6], which abstracts the low-level database operations. It establishes certain conventions to which the developer must adhere, resulting in similar structure of all Rails applications and possibility to automate many tasks by the framework (this is called the *Convention over Configuration* principle). The framework also respects the *Don't Repeat Yourself* principle, avoiding duplicated information scattered in the application code.

Ruby on Rails significantly uses Ruby metaprogramming facilities and takes advantage of its dynamic nature. The simple syntax allows creating little internal domain specific languages, such as the one of ActiveRecord validation specification. It would be harder to create similar framework in other, more conventional languages.[1]

[1] People try nevertheless—see e.g. CakePHP [9].

It is not an exaggeration to say that Ruby on Rails is the "killer app" for Ruby and most programmers who know Ruby know it because they work with the Ruby on Rails framework.

2.5 Implementations

Currently, several implementations of the Ruby language exist in various stages of development. The following sections introduce all implementations known to the thesis author and highlight their specific features or language extensions.

2.5.1 MRI

MRI [1] is the original implementation of the Ruby language. Its name is an abbreviation of "Matz Ruby Interpreter". It is by far the most widely used implementation and also the most mature one. As there is no formal specification of the Ruby language, it also serves as a reference implementation—Ruby is essentially the language induced by this implementation. All other Ruby implementations are generally compared to MRI in terms of completeness, performance and bugs. The implementation is open source (dual licensed under GPL and the Ruby License).

During the development, Matz focused mostly on the language and its features, and not on the quality of the implementation. The result is that the implementation is quite naive and (according to the thesis author's judgment) the overall code quality is not high. Matz himself admits that he is more interested in designing the language than in implementing it. [10]

There are several areas of the implementation that are often criticized:

- **Execution speed.** In MRI, the source code is translated into an abstract syntax tree by the parser and executed directly by a naive tree walker. There is no translation into a bytecode and virtual machine execution. This results in very slow performance in comparison with other dynamic scripting languages.[2]
- **Bad Unicode support.** All strings in Ruby are just sequences of 8-bit characters, not Unicode code points as in other contemporary languages. The source code itself can be either in 7-bit ASCII, Kanji (using EUC or SJIS encoding), or UTF-8. The Unicode support can be added using libraries, but this is a clumsy and limited solution.
- **Threading support.** MRI implements Ruby threading support using green threads (i.e. it does not use threading support provided by the underlying operating system, but implements its own threads in the user space). This threading model has several well-known drawbacks[3] and also complicates compatibility with other implementations such as JRuby or IronRuby, where green threads cannot be used because of

[2] See e.g. language shootout results at http://shootout.alioth.debian.org/gp4/benchmark.php?test=all&lang=all.

[3] For example, a blocking system call invoked from one thread blocks all other threads. Also, threads cannot be spread over multiple cores on multi-core or multi-processor machines.

limitations of the underlying platforms (JVM or .NET, respectively). The reason why Matz chose green threads is simple—portability. In mid-90's many operating systems did not offer reliable threading support, so the only portable option was to implement threads in the user space.

- **Garbage collector.** Ruby uses a conservative mark-and-sweep garbage collector. Due to its implementation, this garbage collector is not fork-friendly and forked Ruby processes often will not share almost any memory after one of them executes the garbage collection. This is very undesirable behavior.

Remedy to some of the often-criticized aspects of Ruby can be found in current development version—the 1.9 branch.

Ruby 1.9 contains *YARV*—a new bytecode interpreter of Ruby replacing the original naive tree walker. Its main goal is to speed up execution of Ruby programs. Initial benchmarks indicate that the execution is generally 2× to 10× faster than Ruby 1.8.6. [11]

YARV compiles the Ruby source code into *YARV instruction sequences*, containing the bytecode. This bytecode is quite high-level and is interpreted by the *YARV virtual machine*, which is a conceptually simple, but heavily optimized stack-based virtual machine. YARV does not attempt to compile the bytecode into native processor instructions.

Ruby 1.9 also contains support for Unicode, implemented in quite general manner—every string in Ruby program specifies its encoding (explicitly or implicitly) and the language contains support for handling strings in all encodings in an uniform way. Many common encodings are supported and strings can be converted between them.

Another problem remedied by Ruby 1.9 is threading—threads are implemented as native threads instead of green threads used in previous versions of Ruby. However to simplify the implementation, Ruby 1.9 uses a global interpreter lock, preventing simultaneous execution of Ruby code.

Although Ruby 1.9 solves many of the problems of previous versions of Ruby, it is currently considered as an unstable branch and released versions are not meant to be used in production environment. The production-level version is expected in December 2008.

2.5.2 JRuby

The JRuby project [12] was started in 2001 with a goal to port Ruby language to the Java platform, mainly to allow using code and libraries written in Java from Ruby and vice versa.

At the beginning, JRuby was a straight port of the C implementation of Ruby 1.6 to Java. It used the same method of executing Ruby code—traversing the abstract syntax tree. Later, the implementation was upgraded to Ruby 1.8 and significantly enhanced. By now, JRuby supports three modes of execution:

1. Interpretation
2. Ahead-of-time compilation of Ruby to Java bytecode
3. Just-in-time compilation of Ruby to Java bytecode

The result of the ahead-of-time compilation can be a standard Java `.class` file. The just-in-time compiler implements advanced features such as decompilation and reoptimization of the generated bytecode. According to one of the lead developers, JRuby is faster than MRI 1.8 and sometimes even than MRI 1.9. [13]

JRuby supports full, two-way integration with underlying Java platform. The programmer can call Java code and use Java libraries transparently from Ruby code executed by JRuby, or execute Ruby programs from the Java code and interact with them. Beyond obvious advantages in reusing existing code, this tight integration makes Ruby suitable as embedded scripting language in Java applications.

JRuby is the most mature of all implementations beyond MRI. It is almost 100% compatible with Ruby 1.8.6 and it is capable of running Ruby on Rails applications. Most obvious differences from MRI—a different threading model and absence of continuation support—are forced by the limitation of JVM, which cannot emulate MRI's green threads and does not allow explicit call stack manipulation, required for implementing continuations. In practice, these limitations are not significant and compatibility level is sufficient for most real-world applications.

JRuby is open source and its development is currently backed by Sun Microsystems, who hired two JRuby lead developers to work on JRuby full-time. The current development aims mainly to enhance performance, real-world compatibility and to make the integration with Java platform easier than it currently is.

2.5.3 IronRuby

IronRuby [14] is a port of Ruby onto Microsoft .NET Framework. It is developed by Microsoft itself and it is implemented on top of the Dynamic Language Runtime (DLR, a library on top of the Common Language Runtime, providing services useful for implementing dynamic languages, such as dynamic type system, dynamic method dispatch and dynamic code generation). The project is quite young, it was announced by Microsoft at the MIX 2007 conference in April 2007. It is developed in a very unconventional way (for Microsoft): as an open source project hosted on RubyForge (the server where most Ruby-related projects are hosted), licensed under Microsoft Permissive License. Non-Microsoft developers are allowed to contribute to the source code.

The primary motivation behind IronRuby is similar to the JRuby project—to enable code sharing between Ruby and .NET platforms. It is possible to call libraries in the .NET framework from Ruby code and also to interact with Ruby programs from the code written in any other language supported by .NET/CLI infrastructure.

The implementation does not support all features of the Ruby language and most standard libraries are not yet implemented. However, an announcement at the RailsConf 2008 conference in May 2008 stated that IronRuby is able to run simple Ruby on Rails applications.

2.5.4 Rubinius

In some aspects, the Ruby language is similar to Smalltalk. Common features include similar object model (based on the idea of message passing and dynamic dispatch—the

object can decide at runtime, if and how it will handle incoming messages) and dynamic nature of the language (almost anything can be inspected and redefined at runtime). Most Smalltalk implementations are almost completely written in Smalltalk itself, the only exception is usually a small virtual machine that runs the Smalltalk code. However in Ruby world, things are different—most Ruby implementations are written in some underlying language (C, Java, C#,...).

The Smalltalk implementation model inspired the Rubinius project [15], which began its life as "Ruby in Ruby". Most Rubinius code is written in Ruby, including most of the core class library. The Ruby code is compiled into a bytecode by a compiler, which is also written in Ruby. The bytecode is interpreted by a stack-based virtual machine written in C.

The virtual machine implements several interesting features such as generational garbage collection and direct threading (a technique, where the bytecode is modified in memory to contain direct pointers to code implementing the opcodes instead of opcodes themselves, avoiding a switch over the opcodes). In future, the virtual machine is intended to be rewritten in a dialect of Ruby that would be used to generate the C code. Similar technique was used in the Squeak implementation of Smalltalk.

Interesting fact is that Rubinius is compatible with Ruby 1.8 API using a special library. This allows C extensions written for MRI to run in Rubinius after simple recompiling, easing its real-world adoption.

Rubinius is currently under heavy development and supports most parts of the Ruby language, library classes and methods. At the time of writing the most recently released version was 0.9. Again, an announcement at the RailsConf 2008 conference in May 2008 stated that Rubinius is able to run simple Ruby on Rails applications. Production-level version is expected to be announced within a few months.

2.5.5 Other Implementations

Implementations described in previous sections are generally relatively advanced and they are able to run many real-world Ruby programs. All of them can run at least very simple Ruby on Rails applications. Some of them are ready for deployment in production environment (MRI, JRuby), the others are expected to be ready soon (IronRuby, Rubinius).

There are also several Ruby implementations that are not yet practically usable for various reasons, mostly because they are at the beginning of their development cycle or because they were abandoned by their creators in early stages. These implementations often contain unconventional ideas, so let's describe them shortly in the next paragraphs.

2.5.5.1 MagLev

MagLev [16] is a Ruby virtual machine based on S64, a Smalltalk virtual machine, developed by GemStone Systems. The project has two main goals: to significantly improve Ruby performance and to enable Ruby to work in the distributed environment.

The usage of modified S64 machine for executing Ruby code was possible because Ruby and Smalltalk share very similar semantics regarding the object model, a method call dispatch mechanism and dynamic language features. Because the S64 is highly optimized

(a result of more than 20 years of its development), the speed of executed Ruby code is significantly higher than the speed of the original Ruby implementation according to MagLev authors (official benchmarks are not yet available).

Multiple virtual machine instances can be running simultaneously and share the global state (e.g. global variables or class definitions), even when they are running on different hardware machines. This is implemented by using an object cache that manages synchronization of the objects between the instances and a storage mechanism that can store the objects persistently and allows manipulation of very large data sets (in order of petabytes). To implement the object sharing and synchronization, MagLev extends Ruby with transactional semantics.

Using the MagLev object storage and sharing features, deployment of large Ruby on Rails applications could be much simplified, because the only servers that need to be running are MagLev instances (in typical large-scale Rails applications today, the web servers, database servers and usually some cache server instances need to be running).

The MagLev project is fairly young (it was introduced at RailsConf 2008 in May 2008 as a three month old project) and while it already runs WEBrick (a simple web server written in Ruby), its support of Ruby is not complete. Precise current status of the project is unknown and so are the details of its implementation, because the project is not developed as open source. Despite that, being backed by a commercial entity, MagLev is expected to improve quickly and it would not be surprising if an announcement of usable production version appeared within several months.

2.5.5.2 MacRuby

MacRuby [17] is a port of Ruby to the Objective-C runtime, developed as an open source project by Apple. The Objective-C runtime is the primary development environment on Apple's Mac OS X system. The main goal of MacRuby is painless integration with the Cocoa Objective-C framework, allowing development of native Mac OS X applications in Ruby. Similar integration was previously possible only via RubyCocoa bridge library [18] for Ruby, but that library fails to overcome some of the differences between Ruby and Objective-C environments, mainly in areas of object model, threading and garbage collection. This leads to less than optimal programmer experience.

Similarly to other ports of Ruby to various platforms/frameworks, MacRuby allows two-way integration between Ruby and Objective-C objects/classes. Many Ruby core classes are implemented as descendants of standard Objective-C classes (for example, Ruby `Array` is a subclass of Objective-C `NSArray`), so they can be used in Objective-C without any conversions or wrapping. The Ruby runtime can handle casual Objective-C objects without any conversions, too. This design leads to conceptional simplicity and good performance of calls spanning between both environments. MacRuby also uses threading support and the garbage collector of the Objective-C runtime.

While most other implementations target Ruby 1.8.6, MacRuby aims to be compatible with Ruby 1.9—the current development branch. As most real-world Ruby programs are written for version 1.8, which is incompatible with 1.9 in many aspects, the practical usability of MacRuby is limited.

2.5.5.3 XRuby

XRuby [19] is a compiler of Ruby to Java bytecode, producing standard Java `.class` files. Its goal is similar to that of JRuby—to enable integration and code sharing between Ruby and Java platforms. It is developed as an open source project by volunteers mostly from China.

The obvious question is, why the XRuby project was started when similar project—JRuby—had existed for quite a long time. According to Xue Yong Zhi, lead developer of XRuby, the main reason was that JRuby was almost dead project in 2005. It supported only old Ruby 1.6 and it was an interpreter, not a compiler. Creating a brand new project made sense at that time. [20]

Instead of reusing the Ruby language parser from the C implementation like all other implementations, XRuby implements its own parser using ANTLR tool. This parser is cleaner than the original one and thus more easily maintainable.

The authors claim that in most benchmark tests, XRuby runs faster than the original Ruby implementation. [21]

The fate of the XRuby project is very uncertain, since its developers participated mostly in their free time and now they are either too busy, or employed to work on Ruby in corporate world. Unless a new maintainer steps up, XRuby will probably fade away. [20]

2.5.5.4 Ruby.NET

Ruby.NET [22] is another (chronologically the first, actually) port of Ruby onto Microsoft .NET Framework. It began its life in 2005 as a Microsoft-sponsored research project at the Queensland University of Technology to study how well such a dynamic language like Ruby maps to the .NET platform.

In 2007, when the IronRuby project was born, the Ruby.NET future began to be uncertain. IronRuby used the Dynamic Language Runtime, which Ruby.NET could not use at the time and basically needed to reimplement its features. Also, the authors were primarily researchers and felt that Microsoft is in better position to produce a production-level implementation. The developers finally decided that there is no need to have two competing .NET Ruby implementations and ended the project. [23]

Some parts of the Ruby.NET project live on, because IronRuby used Ruby.NET lexical scanner and parser in its source code.

2.5.5.5 Cardinal

Cardinal [24] is a compiler that converts Ruby code into bytecode of the Parrot virtual machine. Parrot itself is a quite ambitious project, trying to create a virtual machine that would allow to run various dynamic languages efficiently. In its current version, it supports front-ends in various stages of maturity for many languages, including PHP, Python, Perl, Lua, JavaScript, Scheme and—via the Cardinal project—Ruby.

The Parrot project is notorious for its long development time (the development started in 2001, its current versions are still very unfinished and unstable), frequent architecture changes and internal code rewrites. [25]

The Cardinal project reached only version 0.1, which was released in August 2004. This version did not support most parts of Ruby, with very important parts (such as instance variables and support for class reopening) missing. Currently the project appears to be abandoned.

2.5.5.6 HotRuby

HotRuby [26] is a Ruby virtual machine written in JavaScript. It allows to run Ruby code in the web browsers and Flash programs[4].

HotRuby uses Ruby 1.9 to compile the Ruby source code into the YARV bytecode, which is fed into a virtual machine written in JavaScript. The result is Ruby program running in JavaScript environment. HotRuby contains supporting code and partial implementation of the Ruby core class library in JavaScript, very similar to the PHP runtime in the Ruby to PHP translator (see Chapter 4).

Of all Ruby implementations, HotRuby is the most similar to the translator, because it translates Ruby into other dynamic language. However, HotRuby's approach to the task is very different—it uses a parser and bytecode compiler contained in Ruby 1.9 and only implements the bytecode interpreter and supporting runtime code. This decision allowed the author to avoid reimplementing the Ruby parser, but limited the input to Ruby 1.9 programs. As most real-world Ruby programs are written for version 1.8, which is incompatible with 1.9 in many aspects, the practical usability of the solution is limited.

There needs to be added that it was impossible to use the YARV bytecode interpreter approach in the Ruby to PHP translator, even if this was desired, because Ruby 1.9 was not stable enough when the implementation began. The HotRuby project was started much later (in January 2008), when Ruby 1.9 bytecode interpreter was stable enough to serve as a base for other work.

The HotRuby project is not practically usable at the time of writing and its future is uncertain at best. Although it implements many Ruby constructs, it is not fully compatible with Ruby 1.9 yet and it implements only a skeleton of Ruby core class library. Commits to the source repository come only from one author and there was no activity since the end of January 2008. The project looks abandoned.

2.5.5.7 IronMonkey

IronMonkey [27] is a project that aims to port IronRuby and IronPython (.NET Python implementation) to the Tamarin JavaScript virtual machine. Tamarin was created by Adobe and its code was recently donated to Mozilla and will be used as a basis of its JavaScript interpreter in future versions of their products. The result of the IronMonkey project could be a native, cross-platform implementation of Ruby and Python in the Firefox web browser, enhancing a spectrum of languages available for client-side web application programming.

The project is in its beginnings and no achieved results or implementation details have been published so far.

[4] Flash uses ActionScript, a language based on JavaScript.

2.5.6 Summary

Despite being relatively young, the Ruby language is quite unique given the number of its implementations. Why were so many of them created?

The purpose of several implementations is the integration with commonly used platforms such as Java (JRuby), .NET (IronRuby) or Objective-C/Cocoa (MacRuby). These implementations are generally conservative, emulate behavior of the original Ruby implementation closely and do not extend the Ruby language significantly (except MacRuby). They are also limited by the platform limitations, resulting in different threading models, absence of continuation support and other minor differences. They are backed by the commercial vendors of the respective platforms, which ensures fast pace and continuation of their development.

Other implementations try to remedy deficiencies in the original Ruby implementation, and generally embrace Ruby's Smalltalk heritage. Rubinius tries to rewrite Ruby core in a fashion similar to Smalltalk implementation, MagLev is even using a real Smalltalk virtual machine and adds distribution support to Ruby. The success of these implementations will depend heavily on their performance and on the success in solving perceived MRI problems.

The rest of the implementations can be categorized either as duplicating efforts of other projects (XRuby, Ruby.NET) or as experimental (Cardinal, HotRuby, IronMonkey). These implementations are generally not practically usable and their future is uncertain (if they are not abandoned already).

The number of failed or incomplete implementations suggest that creating an implementation of Ruby is not an easy task. The creator has to overcome the lack of the formal specification and official language grammar, which requires reverse-engineering of the original C implementation. To achieve real-world compatibility, an extensive core class library has to be reimplemented. This library is only loosely documented, which again requires understanding the functionality from the source code of the original implementation. The net result is that implementing Ruby is quite an endeavor and only strong teams with several developers working full-time or near full-time have a chance to succeed.

As a summary, Table 2.1 lists the reviewed Ruby implementations along with their basic properties.

Name	Purpose	Platform	Implementation	Compatibility	Rails Support	License
MRI	original implementation	Windows, Unix-based systems	1.8: naive AST-walking interpreter; 1.9: bytecode-based interpreter with a stack-based virtual machine	N/A	yes	GPL/Ruby
JRuby	integration	JVM	interpreter/compiler into Java bytecode	1.8.6	yes	CPL/GPL/LGPL
IronRuby	integration	.NET	interpreter/compiler into CIL	1.8.6	partial	MPL
Rubinius	MRI rewrite	Unix-based systems	bytecode-based interpreter with a stack-based virtual machine	1.8.6	partial	BSD
MagLev	MRI rewrite	unknown	interpreter	unknown	none	unknown
MacRuby	integration	Mac OS X	interpreter	1.9	none	Ruby
XRuby	integration	JVM	compiler into Java bytecode	1.8.5	none	BSD
Ruby.NET	integration	.NET	compiler into CIL	1.8.2	none	BSD
Cardinal	experimental	unknown	interpreter using Parrot infrastructure	unknown	none	GPL/Ruby
HotRuby	integration/experimental	JavaScript/ActionScript	compiler of YARV instruction sequences into JavaScript representation + interpreter of that representation	1.9	none	unknown
IronMonkey	integration/experimental	Tamarin	unknown	unknown	none	unknown

Table 2.1: Ruby implementation summary.

Chapter 3

Translator Design

The general translator architecture has been described in Section 1.2.

This section describes the rationale behind critical design decisions that had to be made at the beginning of the translator implementation.

3.1 Implementation Language

It was decided to implement the compiler in Ruby.

The main reason for this decision is that Ruby is a very high level and elegant language, which is pleasant to read and write. It does not distract programmers with low-level operations such as memory allocation or pointer manipulation. This results in high speed of development and less opportunities to introduce bugs. Absence of static typing allows rapid changes without rewriting too much code. Also, the author does not deny his fondness for Ruby.

Another important reason for writing the compiler in Ruby was a theoretical possibility of implementing the `eval` function.[1] Its implementation requires presence of the Ruby compiler at runtime, therefore it requires the compiler to be available in PHP. This can be achieved either by writing the compiler directly in PHP, or by writing it in Ruby and using it to compile itself into PHP. When choosing between Ruby and PHP, all other considerations being equal, Ruby is a clear winner.

The choice of Ruby as an implementation language enabled better separation of concerns in the implementation, than would be available in other languages with the same amount of code. For an example, see Section 5.1.3.

Use of Ruby as an implementation language has a negative impact on the compiler speed because Ruby is quite slow compared with e.g. C or Java. This is justified by the gains in development speed and future source code modifications. Also, the translator would not be run very often by the user, so the lower compilation speed is not as critical as in conventional compilers.[2]

[1] The `eval` function takes a string and interprets its contents as a Ruby expression, which is evaluated.

[2] In the original "Rails in PHP" use-case described in the introduction, the compiler would be run only once before each application deployment.

3.2 Supported Ruby Subset

The translator supports only limited subset of the Ruby language. When selecting the implemented parts of the language, the emphasis was put on language elements which are used most often in casual Ruby programs or which are typical for the Ruby language. The implementation therefore contains support for most Ruby control-flow constructions, the whole object system, method dispatch mechanism and blocks. Unimplemented features are mostly those which could be emulated by implemented constructs, which are not used very often, or which were hard to implement.

Because Ruby implements many features in its core class library, it was necessary to implement many classes from it. The criteria for choosing what to implement were similar to the criteria used when choosing language elements.

During the whole implementation, maximum compatibility with Ruby 1.8.6-p114 was maintained. This is the version that most other Ruby implementations target as well.

Detailed overview of implemented language features and parts of the core class library can be found in Appendix B.

3.3 Supported PHP Version

The translator supports both PHP 4 and PHP 5 as a target language, by emitting PHP 4 code which is designed to work also with PHP 5. The decision to support PHP 4 was made at the beginning of the implementation, when PHP 4-only hostings were quite common. Even in summer 2008, PHP 4 is still widely used and the compiler support for the old PHP version is useful.

The decision to support PHP 4 resulted in several implementation difficulties, mainly in the area of object references (see Section 4.1) and implementation of exceptions (see Section 4.7).

With PHP 5 hostings becoming generally available and PHP 4 soon to be officially unsupported, it may be reasonable to drop PHP 4 support, should the translator be developed further.

3.4 Optimization

The Ruby language is rather difficult to optimize correctly. For example, consider a snippet of very trivial Ruby code in Listing 3.1.

Listing 3.1: Ruby program printing result of a simple addition.

```
print 1 + 2
```

In many languages, even the simplest compiler would probably apply constant folding optimization and substitute literal value `3` for the printed expression. However in Ruby, the `+` operator is in fact a method call on its left operand. And because Ruby allows method redefinitions at any time (even for built-in objects such as numbers), few lines of

code from Listing 3.2 inserted before the code in Listing 3.1 would cause the optimization to yield invalid result.

Listing 3.2: Ruby program redefining the `+` method of the `Fixnum` class.

```
class Fixnum
  def +(other)
    self - other # Redefine addition as subtraction.
  end
end
```

Implementation of such a basic optimization as constant folding would require either disallowing certain operations (such as redefinition of methods on built-in objects), which is impractical for the language users, or a complicated control-flow analysis determining if the method was in fact redefined or not. And even this control-flow analysis would be useless at the moment user uses the `eval` function with dynamically created parameter.[3]

The same problems would appear if the code was translated to PHP directly, i.e. if the Ruby `+` operator was rewritten to PHP `+` operator. The only reliable way to translate the snippet above correctly is to use functions emulating method calls on numbers (the emulation is needed because numbers are not objects in PHP and no methods can be called on them). The approximation of the PHP code produced by the translator is shown in Listing 3.3.[4]

Listing 3.3: Code from Listing 3.1 compiled into PHP.

```
r2p_call(r2p_self(), 'puts', r2p_call(1, '+', 2));
```

Situations similar to the one just shown appear at many places in the Ruby language. Thus, it was chosen to refrain from significant optimizations in the compiler and aim mainly for correctness and robustness. As a result, many Ruby syntax elements are not naively translated into their corresponding PHP counterparts, as they do not have identical behavior in all cases. This is described in more detail in later chapters.

3.5 Integration with PHP

When the Ruby code is translated into PHP, it is natural to ask whether it is able to cooperate with the surrounding PHP environment and similarly if the PHP code is able to interact with the translated Ruby program. The translator was not written with this goal specifically in mind (its main motivation was running whole Ruby applications on PHP webhostings, not code sharing and interaction), however certain degree of interaction between PHP and Ruby worlds is possible.

Next two sections demonstrate using Ruby code from PHP and also using PHP code from a compiled Ruby program.

[3] There is a third option: detect the method redefinition at runtime and rewrite the optimized code dynamically to an unoptimized version. While feasible in an interpreter or a virtual machine, this approach is impossible to use in a pure compiler.

[4] The exception handling code was omitted and existence of multiple method call emulating functions was hidden for simplicity.

3.5.1 Using Ruby Code in PHP

Compiled Ruby programs can be embedded into larger PHP applications just by including the compiled PHP code and the translator's PHP runtime (see Chapter 4).

When interacting with the compiled code or the PHP runtime, care must be taken to use wrapper classes and functions—for various reasons, most Ruby objects are not translated directly into equivalent PHP objects (see Section 4.1).

In Listing 3.4, a PHP code uses the translator's implementation of Ruby `String` and `Array` classes to join several strings using a separator and print the result.

Listing 3.4: PHP program working with Ruby objects.

```
<?
  /* Include the translator's PHP runtime. */
  include "runtime/lib/r2p.php";

  /* Create three Ruby strings. */
  $s1 = r2p_string("Alice");
  $s2 = r2p_string("Barbara");
  $s3 = r2p_string("Cindy");

  /* Put them into a Ruby array. */
  $a = r2p_array($s1, $s2, $s3);

  /* Call a "join" method on this array specifying a separator. */
  $joined = $a->ref->join(r2p_string(" and "));

  /* Print the joined strings ("Alice and Barbara and Cindy"). */
  echo $joined->ref->get_value();
?>
```

3.5.2 Using PHP Code from Compiled Ruby Programs

Compiled Ruby code can use the surrounding PHP code, however that code must be wrapped into classes/methods compatible with translators' PHP runtime.

In Listing 3.5, a PHP implementation of the Euclidean algorithm that finds the greatest common divisor of two numbers is wrapped into a method `gcd` in Ruby class `Algorithms`. This class can be used from the compiled Ruby code—see Listing 3.6.[5]

[5] In real world, this particular functionality would be more appropriately implemented inside a module, not as an instance method of a class. But the PHP code showing this would not be as simple.

Listing 3.5: PHP program wrapping existing function into a class usable from compiled Ruby code.

```
<?
  /* Include the translator's PHP runtime. */
  include "runtime/lib/r2p.php";

  /* Original PHP function (parameter checking omitted). */
  function original_gcd($a, $b) {
    while ($a != $b) {
      if ($a > $b) {
        $a -= $b;
      } else {
        $b -= $a;
      }
    }
    return $a;
  }

  /* Wrapper class. */
  class Algorithms extends R2PClass {
    function gcd($a, $b) {
      return original_gcd($a, $b);
    }

    function Algorithms() {
      global $R2P_OBJECT_CLASS;

      parent::R2PClass("Algorithms", $R2P_OBJECT_CLASS,
        $R2P_OBJECT_CLASS);

      $this->_define_public_methods_from_php_functions(
        array("gcd", 2)
      );
    }
  }

  /* Create the class instance. The Ruby code knows about the class
     from this point. */
  $R2P_ALGORITHMS_CLASS = new R2PRef(new Algorithms());
?>
```

Listing 3.6: Ruby program using the `Algorithms` class from Listing 3.5.

```
print Algorithms.new.gcd(15, 25) # Prints "5"
```

Chapter 4

PHP Runtime

The PHP runtime is a PHP code that supports compiled Ruby programs while they are running. It consists mainly of the core class library implementation and code supporting the compiler-generated code and runtime operations. The reasons for its existence are outlined in Section 1.2.

The design of the PHP runtime is tightly coupled to decisions how to map certain Ruby language constructs to PHP. Therefore, these decisions are described along with the runtime itself and the PHP runtime is also described before the compiler. Only the most important mappings are described here, more detailed description can be found in Section 5.2.3.

The PHP runtime itself is divided into two parts:

1. Low-level support for the compiler-generated code and runtime operations
2. Implementation of the core class library

The first part is scattered in files in the `/runtime/lib` directory. Code of the second part can be found in subdirectories `classes`, `modules` and `objects`.

Note that these two parts are very tightly coupled. This is necessary because in Ruby, a lot of functionality is implemented in the core class library that would be usually implemented directly in the language core in more conventional languages. Thus, the core classes need to access the language runtime.

The PHP runtime is object oriented (as far as compatibility with PHP 4 allows). Some parts of the runtime were impractical to implement in OOP way; these are usually implemented procedurally. This concerns mainly functions used by the compiler-generated code, but there were also many cases where the OOP could not be used because the Ruby objects are not always represented as PHP objects in PHP (see Section 4.1.2).

All globally visible PHP runtime identifiers are prefixed with `r2p` or `R2P` prefix and certain identifiers with internal usage are prefixed with `_r2p` or `_R2P` prefix. The prefixes supplement namespaces which are not available in PHP.[1]

[1] Namespaces will be available in PHP 5.3.

4.1 Ruby Object Representation

The most important decision in the PHP runtime design was how to represent Ruby objects. To explain this decision, it is necessary to discuss the representation of objects in the original Ruby implementation first.

4.1.1 Ruby Object Representation in MRI

All values in the original Ruby implementation are internally stored in the `VALUE` type, which is just a typedef for `unsigned long`. For objects of classes `NilClass`, `TrueClass`, `FalseClass`, `Fixnum` and `Symbol`, the `VALUE` contains immediate data of the object (objects of those classes are called *immediate values*). For objects of other classes, the `VALUE` stores a pointer to a structure with additional data (these are called *referenced values*). The exact type of the structure depends on the object class. This scheme optimizes handling of numbers and few other special values, saving heap allocations and pointer dereferences.

The implementation takes advantage of the fact that on modern architectures pointers are aligned at least on word boundaries and generally do not point to very small addresses, so there is a room to embed additional information into them. This information is used to distinguish between the immediate value and pointer case.

The interpreter determines the class of object stored in the `VALUE` using the following algorithm:

1. If the least significant bit of the `VALUE` is set to 1, the stored object is a `Fixnum`. Its value can be obtained by shifting the `VALUE` right by one bit.
2. Otherwise, if the eight least significant bits of the `VALUE` are equal to 0x0E, the stored object is a `Symbol`. Its ID can be obtained by shifting the `VALUE` right by eight bits.
3. Otherwise, if the `VALUE` is equal to 0, the stored object is `false` (the only instance of `FalseClass`).
4. Otherwise, if the `VALUE` is equal to 2, the stored object is `true` (the only instance of `TrueClass`).
5. Otherwise, if the `VALUE` is equal to 4, the stored object is `nil` (the only instance of `NilClass`).
6. Otherwise, the `VALUE` stores a pointer to a structure with additional information. The object's class can be determined using the structure's `klass` member, which is always present in the structure at a fixed offset.

When assigning variables, passing parameters to methods or blocks and obtaining return values, the Ruby interpreter internally just copies the `VALUE` around. This means that the immediate values have *pass-by-value* semantics, as all their data is stored directly in the `VALUE`. All other objects have *pass-by-reference* semantics, as only a pointer to

their data is stored in the `VALUE` and no deep copying is performed. However in reality, this distinction in the passing semantics is irrelevant because all immediate values are immutable by design. The user has no way to determine if the immediate value was copied or not.

4.1.2 Ruby Object Representation in PHP Runtime

In PHP, all values are of eight different types: `boolean`, `integer`, `float`, `string`, `array`, `object`, `resource` and `NULL`. The Ruby classes are mapped to PHP types according to Table 4.1.

Ruby Class	PHP Type
`NilClass`	`NULL`
`TrueClass`	`boolean`
`FalseClass`	`boolean`
`Fixnum`	`integer`
other	`object`

Table 4.1: Mapping of Ruby classes to PHP types.

With an exception of `Symbol`, which has no PHP equivalent, immediate values are mapped to their corresponding PHP types and all other objects to PHP `object`. This corresponds to storing the data directly in the `VALUE` for immediate values and storing a pointer to additional data for other objects in the original Ruby implementation. The only difference is that the PHP type system is used to distinguish between the cases instead of low-level bit manipulation.

The chosen mapping has several advantages:

- It is very close to the scheme used by the original implementation, so code in the PHP runtime duplicating parts of the Ruby interpreter can be written in the same way.
- Manipulation with `Fixnums` is fast.
- Ruby booleans are identical to PHP booleans and Ruby `nil` is identical to PHP `null`, which simplifies the PHP runtime code.
- Most Ruby objects are PHP objects. Given that PHP supports object-oriented programming, this is a natural mapping which simplifies the PHP runtime code and more importantly makes the interaction with the compiler-generated code from PHP quite natural.

Several other Ruby classes (namely `Float`, `String` and `Array`) could have been mapped to their corresponding PHP types, too. However, this mapping would have no significant advantages, it would deviate from the original Ruby implementation model and introduce many special cases in the PHP runtime code. Therefore, these classes were mapped to PHP objects.

4.1.2.1 Pass-by-value vs. Pass-by-reference Problem

In PHP 4, all values are passed by value when assigning, passing as parameters, etc.[2] Passing by reference can be generally enforced using the `&` prefix operator, but this choice is made statically—either all values on given place will be passed by value, or by reference. This conflicts with Ruby dynamic behavior, where the used passing mechanism depends on the object class. This mechanism is not possible to implement directly in PHP 4 and workaround had to be invented.

The pass-by-reference mechanism in Ruby works because in the original implementation a `VALUE` contains a pointer to additional data, not the data itself, and only that pointer is copied when passing the value around. This is emulated in PHP by wrapping all non-immediate values into a small `R2PRef` proxy class. This class contains no methods except the constructor and only one attribute: `ref`. This attribute is a PHP reference pointing to the actual object. When passing a value, the `R2PRef` instance itself is always copied by PHP, but since the copying is shallow, the reference in its `ref` attribute still points to the same object, resulting into pass-by-reference semantics for the referenced object.

The wrapping and unwrapping of objects causes a (very slight) performance degradation and makes the PHP runtime code a bit longer. This is a low price to pay for the correct value-passing semantics.

Note that in PHP 5, objects are always passed by reference. The described problem disappears here by itself, because Ruby objects represented by PHP `object` are exactly those with pass-by-reference semantics in Ruby. The PHP runtime code uses the `R2PRef` proxy object in PHP 5 anyway—the opposite would require two versions of many PHP runtime parts with obvious implications on the code size and complexity. As a result, the PHP runtime code must be carefully constructed not to rely on the pass-by-value vs. pass-by-reference semantics for PHP objects, so it is able to run in both PHP 4 and PHP 5.[3]

4.2 Implementation of Ruby Objects

As mentioned in the previous section, most Ruby objects are directly mapped to PHP objects. All PHP objects representing the Ruby ones are instances of `R2POObject` class or one of its descendants.

In principle, instance variables of Ruby objects could be stored in attributes of PHP objects. However the runtime implementation uses several attributes for its own purposes (at least `$id` and `$class`) and a conflict would arise if the translated Ruby program wanted to use those names. Rather that reserving certain attribute names or inventing some escaping scheme, the instance variables are stored in an associative array in the `$instance_vars` attribute. The disadvantage is a need to use a wrapper functions in the generated code to access these attributes instead of direct attribute access.

[2] In fact, the situation is more complicated—PHP employs copy-on-write semantics on all the values for performance reasons. But this is transparent to the user.

[3] This is true for all code which needs to run in both PHP 4 and PHP 5 in general.

4.2.1 Object Identity

All Ruby objects have an integer ID, which is unique for the lifetime of the object and can be obtained by calling the `object_id` method.

In the original Ruby implementation, immediate values derive their ID directly from their value and other objects use a value derived from a pointer to their data. PHP lacks similar object identity mechanism and the absence of pointers precluded its implementation in the same way as in the original Ruby implementation. The object identity was implemented simply by including an `$id` attribute into each `R2PObject` instance. This attribute contains the object ID and it is assigned in the constructor. The IDs are assigned sequentially using a static counter.

The disadvantage of this mechanism is its predictability. If any Ruby program relies on the fact that object IDs are essentially random for non-immediate values and generally different on each program invocation, it may stop working properly when translated by the translator. However, such a reliance would be a bad practice because the randomness of IDs is not guaranteed.

4.3 Classes and Modules

4.3.1 Classes

Classes in PHP are very similar to classes in other common languages, but Ruby classes are somewhat unusual and contain several uncommon features:

- Ruby classes are normal objects with attributes, methods, etc. that can be accessed by the programmer at runtime. This is not true in PHP.
- The corollary of the previous fact is that Ruby classes can be dynamically created, copied, deleted, etc. None of this is possible in PHP (without resorting to "hacks" using the `eval` function).
- Ruby code is present inside Ruby class definitions—the class definitions only change the context in which the code runs. No similar capability exists in PHP class definitions.
- In Ruby, classes themselves are highly flexible. Methods can be added, changed, aliased and deleted at runtime. The same holds for constants and class variables. Class definitions can be reopened long after initial creation of the class. Finally, modules can be included at any time—which technically changes the inheritance chain during the execution of the program. None of these operations are possible with PHP classes.

The difference between Ruby and PHP classes is so huge that Ruby classes cannot be directly mapped to PHP ones and have to be emulated entirely by using PHP objects.

Ruby classes are translated into instances of `R2PClass` class. They store a class name, list of constants, class variables and methods—all implemented as PHP associative arrays

mapping names to values (in case of constants and class variables) or objects containing additional information (in case of methods). Classes also contain a reference to their superclass in the `$superclass` attribute. Each object has a reference to its class in the `$class` attribute.

The beginning of the class definition is translated into a PHP runtime call which sets a proper scope and creates a new `R2PClass` instance (in case the class is new) or finds an existing one (in case the class is reopened) and sets is as an active class.

The end of the class definition is translated into a PHP runtime call which restores the previous scope and active class.

4.3.2 Modules

As modules are essentially non-instantiable classes, the module definitions are translated in a very similar way as classes. The only difference is that modules are represented as instances of the `R2PModule` class in the PHP runtime and they do not have a superclass.

4.3.3 Implications

The emulation of Ruby classes using PHP objects makes it impossible to use PHP mechanisms for inheritance, method dispatch and constant storage. These had to be emulated entirely.

4.4 Variables

In Ruby, there are five types of variables: local, global, instance and class variables and constants. Local variables are limited in scope to the current method or class definition. Global variables can be accessed in the whole program. Instance variables are attributes of object instances, class variables and constants are attributes of classes (the main difference between them is that constants generate a warning when changed).

While all those variable types have their counterparts in PHP, they could not be mapped directly into them for various reasons. Those reasons and the mapping solutions are described in the following sections.

4.4.1 Local Variables

While similar, local variables in Ruby and in PHP have two important differences:

1. In Ruby, local variables can be used inside methods, at the top level, or inside class or module definitions. In PHP, they can only be used inside functions or methods. They would become global variables at the top level and no executable code is allowed inside class definitions in PHP.

2. Access to an undefined variable raises an exception in Ruby. Behavior of PHP is somewhat surprising—a default value is used (depending on the context, it may be `false`, `0`, `""` or `array()`) and an `E_NOTICE` level error is generated.

To accommodate these differences, the local variables were implemented independently on the PHP ones using a stack of associative arrays mapping variable names to their values. Items are pushed on this stack whenever new local variable scope is created and popped from the stack whenever it is closed. The stack is accessed via wrapper functions, which raise correct Ruby exception when necessary.

4.4.2 Global Variables

Compiling Ruby global variables to PHP global variables would require to emit `global` statement at the beginning of each PHP function in the compiler-generated code. However, global variables can be dynamically created and the static global statement cannot make newly created variables visible.

Also, some Ruby global variables are virtual—accessing them invokes a getter or a setter, which usually does some work or checks in addition to retrieving or setting the variable value. There is no such facility available for PHP global variables.

Those two reasons led to a decision to store all Ruby global variables in one PHP global variable, which contains an associative arrays mapping variable names to their values. The array is accessed via wrapper functions, avoiding any `global` statements (these need to be present only in the wrappers). The wrappers also handle virtual global variables.

4.4.3 Instance Variables, Class Variables and Constants

Because several attributes in `R2POObject`, `R2PModule` and `R2PClass` are used for implementation purposes, instance variables, class variables and constants are stored in special instance attributes containing associative arrays mapping variable names to their values. The arrays are again accessed via wrapper functions. Besides simplifying the generated code, these functions also implement correct variable lookup semantics, which is quite complicated in case of class variables and constants in Ruby.

4.5 Methods

In Ruby, classes and modules can have methods, but there is no concept of standalone functions. Top level functions that look like them are in fact methods of the class `Object`. On the other hand, PHP supports both functions and methods. Thus, the natural mapping of Ruby methods seems to be a conversion into PHP methods.

Sadly, this straightforward mapping cannot work by itself. As noted in Section 4.4, Ruby classes and modules are not implemented as PHP classes, but as objects with a list of method names and other properties. No PHP class is created when Ruby class or module definition is encountered. It makes no sense to implement Ruby methods as PHP methods, because they would not have any class to be in.

However, the implementation of core library classes and modules classes would benefit from the implementation of Ruby methods as PHP methods. Ruby core library classes and modules are implemented as objects of certain PHP classes (see Section 4.8.2) and it

would be useful to allow those classes to define PHP methods which would be available both to the PHP code and to the translated Ruby code.

These conflicting needs were solved by allowing the Ruby methods to be represented both as PHP functions and PHP methods. The record corresponding to given Ruby method in the class method list indicates the type of implementation. The implementation of Ruby methods as PHP methods is generally used only by the core library classes and modules; the compiler always translates casual Ruby methods into PHP functions.

4.5.1 Method Information

As mentioned in Section 4.3.1, each PHP object representing a Ruby class or module contains a list of its methods. This list is implemented as an associative array, mapping method names to `R2PMethodInfo` objects with information about the method. The information stored are the method implementation type (function or method), visibility (public, protected or private), number of parameters allowed (minimum and maximum) and the name of the PHP function implementing given method (or an array with a reference to a Ruby class object and a method name in case of method implementation).

4.5.2 Invocation

When a method is invoked in Ruby, the runtime looks into the class of the receiver and tries to find the `R2PMethodInfo` object corresponding to the method. The algorithm used follows the Ruby semantics (if the method is not found, the superclass is searched, then its superclass,...; `method_missing` is finally called if the method is not found) and it bypasses the PHP method dispatch mechanism entirely.

If the method information is found, the method visibility is checked and actual parameter counts are compared with the counts specified in the `R2PMethodInfo` object, potentially raising an exception if one of the checks fails.

Finally, if the checks do not find any problem, the method is called using `call_user_func_array` function. Usage of this function combined with a suitable format in which the implementation location is stored in the `R2PMethodInfo` object allows to easily call Ruby methods implemented both as PHP methods and as PHP functions. If the method is implemented as a function, a parameter representing `self` is prepended just before the call is executed.

The chosen method dispatch mechanism has a disadvantage that for one Ruby method call, many PHP calls are required, mostly to various internal functions handling the dispatching. Emulated Ruby method calls are therefore slower than the PHP ones. On the other hand, the mechanism exactly matches the semantics of Ruby method calls.

4.5.3 Parameters

The translator supports positional and rest parameters in Ruby method definitions and calls. During a method call, the actual parameters are packed into an array and that array is later passed to `call_user_func_array` function, which substitutes them into formal parameters of the called PHP function or method internally. This calling convention

means that Ruby parameters are exactly translated into PHP parameters, which simplifies both the PHP runtime and the compiler-generated code.

In Ruby, actual parameters are treated like local variables assigned to at the beginning of the method. Because PHP local variables are not used to emulate Ruby local variables directly, the formal parameter declaration is omitted in the compiler-generated functions that implement Ruby methods and instead the local variables are assignged in the `r2p_method_enter` function, called at the beginning of each PHP function representing a Ruby method. This function has knowledge about names and structure of the formal parameters and retrieves the actual values using `func_get_args` PHP function.

4.5.4 Stack

The parameters of compiled Ruby methods are passed using PHP native stack. However, the implementation needs to store more information about method calls than the PHP stack can accommodate: receiver of the method, method's Ruby name, its lexical parent (i.e. class it was defined in) and a passed block.

The PHP runtime stores this information on an additional stack, parallel to the PHP one. Frames of this stack are instances of `R2PStackFrame`, or more precisely of its descendants `R2PMethodStackFrame` and `R2PBlockStackFrame`. The first one is used for method calls, the second one for block invocations. The stack is stored in a global variable and it is maintained by the functions which wrap method calls and block invocations. The PHP runtime code does not need to care about this stack until it needs information stored there.

4.6 Blocks

Ruby blocks can be considered anonymous functions with closure properties. Because creating anonymous functions is very inconvenient in PHP[4], blocks are translated into named functions. The information about block's PHP function name is stored on the stack when a method associated with the block is called.

When invoking a block, information about its associated PHP function is found on the call stack and this function is called. All parameters are passed in a similar way as when calling a method, however the block PHP function handles them differently than a PHP function of a method—the parameters are assigned into variables directly in the body of the function. This behavior is enforced by richer parameter options for blocks (quite surprisingly, blocks can have any kind of variable as a parameter, even globals).

4.7 Exception Handling

Ruby contains support for exceptions, but PHP 4 lacks such feature. To make exceptions in translated programs work, the PHP runtime has to emulate them.

[4]The only method is to use the `create_function` function and pass the function body as a string.

When devising suitable emulation mechanism, the original Ruby implementation was searched for inspiration, because the C language also does not have exception facilities. It was found that exceptions are emulated using `setjmp/longjmp` mechanism, which is not available in PHP.

Two options remained—to emulate the exceptions using special value returned from functions or with a global variable containing the currently raised exception. (The second approach is possible because in Ruby only one exception can be raised at any given time.[5]) In both cases, the exception status needs to be checked after all PHP runtime function calls or other operations that could potentially raise exceptions.

It was decided to proceed with the second option—a global variable. It introduces more global state into the runtime, but it seems cleaner than overriding the return value. With good encapsulation, the code checking the global variable can be cleaner than the code checking the return value. This mechanism is also more compatible with original Ruby implementation.

The PHP runtime uses two global variables for its exception-emulation mechanism: one contains a reference to a currently raised exception and another contains a boolean flag indicating whether there is an exception in need of handling. The PHP runtime code does not manipulate these variables directly—it handles them using wrapper functions.

The code of the PHP runtime and the compiler-generated code checks the exception flag after each operation that may raise an exception and reacts appropriately (most typically by immediate return from the current function). The exception handler also checks this flag and if it handles the exception, it clears it together with the currently raised exception variable.

If the exception is not caught in the program and reaches to the top level, it is handled by a top level exception handler function, which displays the exception class and message and terminates the program.

Note that PHP 5 supports exceptions, which could have been used to emulate Ruby exceptions quite easily. However, the PHP 4 compatibility precluded this.

4.8 Core Library Classes and Modules

Large part of the PHP runtime consists of the Ruby core class library implementation in PHP. To explain how its classes and modules are implemented, it is necessary to discuss implementation of the core library in the original Ruby implementation first.

4.8.1 Core Library Classes and Modules in MRI

In the original Ruby implementation, core library classes (such as `String` or `Array`) are implemented in C and they are internally represented as C structs (such as `RString` or `RArray`). These structs contain members storing necessary instance data (such as string characters or array items) in addition to data common for all objects (pointer to its class and its instance variables).

[5] More precisely, only one exception can be raised at any given time *in one thread*. However this distinction is irrelevant, because the translator does not support threads.

Methods of the core library classes are implemented as normal C functions with additional parameter of the `VALUE` type representing `self`. There is a mechanism to translate Ruby method calls into the native C calling convention.

If a user creates a class derived from a built-in core library class, it will be internally stored in the same struct as the base class. This ensures that methods from the base class used in the the derived class have access to all base class instance data.

4.8.2 Core Library Classes and Modules in PHP Runtime

In the PHP runtime, PHP classes derived from `R2POObject` are used for implementation of Ruby core library classes. Their instance data is stored in the instance attributes. Inheritance is used to match relations between the classes on the Ruby side. Methods of the core library classes are implemented as methods of the implementing PHP classes (see the discussion in Section 4.5).

If the user creates a class derived from a built-in core library class, it will be internally stored in the same PHP class as the base class. Like in the original implementation, this ensures that methods from the base class used in the the derived class have access to all base class instance data.

Chapter 5

Compiler

The compiler consists of three parts:

1. **Parser**, which scans the Ruby source code and builds its in-memory representation in the form of abstract syntax tree (AST).
2. **Transformer**, which operates on the Ruby AST and transforms it gradually into an AST representing the equivalent program in PHP.
3. **Serializer**, which serializes the PHP AST into PHP source code.

The important architecture decision was to separate the transformer and the serializer, so that the transformation operates entirely on the AST level. The alternative would be to emit the PHP source code directly from the Ruby AST, but this would most probably result in fragile and duplicated code.

The following sections describe the compiler parts in more detail.

5.1 Parser

5.1.1 Problems with Parsing Ruby

The Ruby language is difficult to parse. It offers a rich and flexible syntax, which is convenient for the programmer, but it contains many features and irregularities that make it hard to process. Consider the example in Listing 5.1.

Listing 5.1: Ruby program using string interpolation.

```
a = 5
b = 6
print "#{a} + #{b} = #{a+b}" # Prints "5 + 6 = 11".
```

This example uses Ruby feature called *string interpolation*. When a programmer inserts a `#{...}` sequence into appropriate kind of string literal, the `...` part (which can be

any Ruby expression[1]) gets evaluated and the result is substituted in the string. This substitution happens at runtime, whenever the value of the string literal is evaluated.

Because the `#{...}` sequence can contain any Ruby expression, it can even contain other Ruby strings, which can again be interpolated. In other words, string interpolation can be nested recursively—see Listing 5.2.

Listing 5.2: Ruby program using nested string interpolation.

```
print "#{"one plus one is two: " + "#{1 + 1}"}"
```

For the Ruby parser writer, this means that string literal parsing is not a matter of lexical analysis (like in most languages). Instead, it has to be done on the syntax analysis level, so the parser can check proper nesting and recursively invoke itself to parse the embedded subexpressions.

The example shows just one occurrence of this problem; there are more similar situations in the Ruby grammar. The result is that the language grammar has to contain more detailed information than in other common languages and the Ruby lexer has to have a bit unusual structure (often emitting somewhat strange tokens such as "beginning of the interpolated string").

Another important Ruby trait its that its lexer needs to keep a lot of state between parsing subsequent tokens. One example is parsing of the `__END__` directive, which can mean the end of the source code (when occurring at the beginning of the line) or a casual identifier (when occurring anywhere else). This and other similar ambiguities require the lexer to keep extensive state information to distinguish the ambiguous cases, which prevents implementing the lexer as a simple state automaton that parses tokens independently.

In fact, the situation with Ruby lexical analysis is even more interesting. The flexibility of the language causes identical constructs to have different meaning in various contexts.

For example, method names can be reserved words. To allow this, reserved word recognition in the lexer needs to be turned off when a method name is expected in the grammar. But the decision to turn off reserved word recognition cannot be done by the lexer alone, because it requires information from the parser about the context where the lexing happens. As a result, the lexer and the parser cannot be independent, but have to cooperate quite intimately. From a perspective of a programmer accustomed to the fact that the lexer is independent of the rest of the parser, this is very disappointing.

The last problem with parsing Ruby is that the language grammar lacks formal description and its only definition is the original Ruby implementation. Creating any tool that needs to deal with Ruby programs requires extensive study of the Ruby implementation and inferring the grammar rules directly from the source code of the original Ruby parser.

5.1.2 Parser Implementation

Rather than inferring the Ruby grammar from the Ruby parser source code and creating a parser from scratch, which (considering problems illustrated in Section 5.1.1) would be

[1] Or, more precisely, a sequence of Ruby statements. This distinction is irrelevant in the context of the Ruby parsing discussion.

a huge task with uncertain result, it was decided to emulate the Ruby parser behavior and it was basically rewritten from C to Ruby.[2]

To write the parser, *Racc* [28]—a parser generator for Ruby very similar to classical C language parser generator Bison—was utilized.[3] The similarity to Bison allowed to retain the structure of the original Ruby parser, which is written using it. The grammar rules were almost exactly copied from the original parser, but the semantic actions are different—they generate a cleaner and more object-oriented form of AST than the original implementation. The parser is encapsulated into its own class (`R2P::Parser::Parser`).

The lexer is a direct rewrite of the original Ruby lexer, which was written by hand and communicates with the parser using several global variables. To avoid introduction of a global state into the lexer implementation, the lexer was encapsulated into its own class (`R2P::Parser::Lexer`) whose attributes were used to communicate with the parser. Thus, a better lexer–parser separation was achieved than in the original implementation.

5.1.3 AST Representation

Nodes of the translator's AST are classes representing various types of syntactic elements found in the language. All classes are descendants of `AST::Node` class, which defines common behavior.

Because class definitions are open in Ruby (so the programmer can for example add or redefine methods at any point after the class creation), the chosen representation allowed to add behavior to AST classes in other parts of the translator, leading to the separation of concerns. This is used in the transformer and serializer, where methods for transformation and serialization of the nodes are attached to the classes in separate files—nicely grouped together at one place. In less flexible languages, the visitor design pattern [32] would have to be used to achieve the same level of separation, requiring certain amount of boilerplate code.

5.2 Transformer

The transformer operates on the Ruby AST and transforms it gradually into an AST representing the equivalent program in PHP. Because PHP is a high level language, this transformations is direct and no intermediate representation is used beyond the Ruby and PHP syntax trees.

In cases where there are equivalent constructions in both languages, the transformation is very straightforward with one node in the Ruby AST translated into one node in the PHP AST. However since Ruby is much more expressive than PHP, quite often the behavior of one Ruby language construct must be emulated by a series of PHP constructs and/or

[2] Creators of all alternative Ruby implementations whose source code is available did exactly that and rewrote the original parser to their implementation language. Only XRuby developers attempted to write Ruby grammar in ANTLR.

[3] Other Ruby parser generators were briefly evaluated, namely rbison, rockit [29], Coco/R [30] and ruby-yacc [31]. Sadly, each of them had serious issues (e.g. missing documentation, bad error handling or dependencies on other software) that prevented its usage.

calls into the PHP runtime. In these cases, one Ruby AST node is typically translated into multiple PHP AST nodes.

The following sections discuss the transformer design and describe how Ruby language constructs are compiled into PHP.

5.2.1 Transformer Design

The transformer does its job by a simple recursive walk through the Ruby AST tree, during which the resulting PHP AST tree is gradually built. The Ruby AST node classes are extended by defining a `transform` method, which returns the PHP AST node for the Ruby AST node it is called on. In many cases, the `transform` method recursively calls itself on descendants of the given node.

For technical reasons, the `transform` method of some AST nodes does not return the corresponding PHP nodes, but a structure with information used by the caller. This is further described in Section 5.2.3.

The `transform` method requires some context information, which is passed in the `context` parameter. For a detailed description of the context data, see the transformer source code (`compiler/lib/r2p/transformer.rb`).

5.2.2 Statement-Expression Mismatch

As typical for languages with functional programming elements, every statement in Ruby is an expression and has a value.[4] This is not true in PHP, where many statements are not expressions and it makes no sense to ask for their value.

For example, the value of Ruby `if` statement is the value of its taken branch, or `nil` if no branch was taken (this can happen only when the condition is evaluated as false, but there is no else-branch). The value of the branch is simply a value of the last contained statement. However, the `if` statement or any other branching statement in PHP is not an expression.[5] How is the Ruby code in Listing 5.3 translated into PHP?

Listing 5.3: Ruby program printing the value of an `if` statement.

```
# Prints "if-branch" or "else-branch" depending on the "condition"
# variable value.
print(
  if condition
    # Some statements may be here...
    "if-branch"
  else
    # Some statements may be here...
    "else-branch"
  end
)
```

[4] Actually, this is a simplification—there are several minor exceptions, but they are all handled by the parser, which reports an error when a value is required from a statement that is not an expression.

[5] Except the ternary operator, which has a limited use.

To explain a general solution of the presented problem, it is first necessary to describe two key concepts: the `expression?`[6] method and saving of the node value.

5.2.2.1 The `expression?` Method

For every Ruby AST node it is important to know if the PHP AST node resulting from its translation would be a PHP expression or not. All node classes provide this information using the `expression?` method.

For some node classes, the `expression?` methods returns always the same value (e.g. a Ruby class definition will never be transformed into a PHP expression). For other node classes, the result is determined dynamically according to a node attributes or its subnodes (e.g. a Ruby array literal will be transformed into a PHP expression if and only if all its items will be themselves transformed into PHP expressions). The second "dynamic" case is a reason why the `expression?` method is not implemented as a simple class (static) attribute.

5.2.2.2 Saving of the Node Value

On many occasions, the value of a Ruby AST node needs to be saved into a variable in PHP to be reused later. To do this, the `transform` method of that node needs to be called with `value_needed` parameter set to `true`. The emitted PHP AST will contain a computation of the node value and its assignment into a PHP variable. The variable name is based on the node ID, which is assigned to it during its creation.

The exact code generated depends on the node class. Generally, if the Ruby AST node is transformed into a PHP expression, it is sufficient to wrap a node generated in the `value_needed = false` case into a node representing PHP assignment. In other cases, the generated code is usually modified more.

5.2.2.3 Solving the Statement-Expression Mismatch

With understanding of the `expression?` method and saving of the node value, it is possible to explain how the code in Listing 5.3 would be compiled. The node representing the `print` method call (the "call node") will call the `expression?` method on the node representing its parameter (the "if node"). Its return value will be `false` (as Ruby `if` statement is not transformed into a PHP expression), so the call node will know that it needs to generate a node representing two statements: (*a*) the transformation of the if node together with saving its value into a variable (this is accomplished by calling `transform(true, ...)` on the if node) and (*b*) a `print` method call with a saved if node value as a parameter.

Many situations similar to the presented example can be constructed and all of them can be solved using the same technique as the one just described.

[6] The question mark at the end of the name is a Ruby convention to mark boolean methods.

5.2.3 Transformation of Ruby Constructs

This section describes the transformation of supported Ruby constructs into PHP. The structure of the section is based on the listing of supported language elements in Appendix B. The description is more precise and complete than the description of runtime representation of various Ruby constructs in PHP in Chapter 4. Still, many unimportant details were left out—see the `compiler/lib/r2p/transformer.rb` file in the translator source code for more information.

5.2.3.1 Literals

The **integer literals** in the `Fixnum` range are transformed into PHP integer literals. Integer literals outside the `Fixnum` range are transformed into `r2p_bignum` PHP runtime call, which creates a reference to `R2PBignum` instance representing Ruby `Bignum` class in PHP.

The **float literals** are transformed into `r2p_float` PHP runtime call, which creates a reference to `R2PFloat` instance representing Ruby `Float` class in PHP.

The **string literals** are transformed into `r2p_string` PHP runtime call, which creates a reference to `R2PString` instance representing Ruby `String` class in PHP. If the translated string is interpolated, the interpolation is translated into printf-like string and additional `r2p_interpolate` call is generated to handle the substitution. For example, Ruby string `"#{1} and #{2}"` is transformed into PHP code `r2p_string(r2p_interpolate('%s and %s', 1, 2))`.[7]

The **symbol literals** are translated in a similar way as strings but with `r2p_symbol` PHP runtime call used.

The **array literals** are transformed into `r2p_array` or `r2p_array_with_splat` PHP runtime calls, which create a reference to `R2PArray` instance representing Ruby `Array` class in PHP. Array items are passed as parameters to the call. The `r2p_array_with_splat` call is emitted when the splat operator is used and handles its applying (for a description of the splat operator, see Section 4.5.5.5 in [33]).

The **hash literals** are transformed into `r2p_hash` PHP runtime call, which creates a reference to `R2PHash` instance representing Ruby `Hash` class in PHP. Hash entries are passed as parameters to the call, with each entry represented as a two-element PHP array.

The **range literals** are transformed into `r2p_range_with_initialize` PHP runtime call, which creates a reference to `R2PRange` instance representing Ruby `Range` class in PHP. Range bounds and the exclusivity flag are passed as parameters to the call. The `_with_initialize` suffix emphasizes that the `initialize` method (which performs some checks on the range bounds) is called after creating the `R2PRange` instance.

5.2.3.2 Variables and Constants

At runtime, all **global variables** are stored in the `$r2p_global_vars` PHP global variable. It contains an associative array mapping variable names to their values or a pair of

[7] The reader maybe expected `%d` format specifier instead of `%s` in the format string. The use of `%s` format specifier is correct, because the interpolated values are converted into strings before interpolation by the `r2p_interpolate` function.

functions (getter and setter) in case of virtual variables.

Global variable reads are transformed into `r2p_global_var_get` PHP runtime call, which attempts to obtain the value of given variable from the `$r2p_global_vars` array. If the variable exists, its value is returned, otherwise `R2P_NIL` constant representing Ruby `nil` is returned.

Global variable writes are transformed into templates for `r2p_global_var_set` PHP runtime call, which stores given variable value in the `$r2p_global_vars` array. The template is filled-out with the set value and the calls are actually generated by the parent node in the AST, which is always an assignment.

At runtime, all **local variables** are stored in a stack in the `$r2p_local_vars` PHP global variable. Each item on the stack contains an associative array mapping variable names from a particular scope to their values. When a new local variable scope is created, an empty item is pushed on the stack; when a local variable scope is destroyed, the top stack item is popped.

Local variable reads are transformed into `r2p_local_var_get` PHP runtime call, which attempts to retrieve the value of given variable from the top of the local variable stack. If the variable exists, its value is returned, otherwise `R2P_NIL` constant representing Ruby `nil` is returned.

Local variable writes are transformed into templates for `r2p_local_var_set` PHP runtime call, which stores given variable value in the item on the top of the local variable stack. The template is filled-out with the set value and the calls are actually generated by the parent node in the AST, which is always an assignment.

At runtime, **instance variables** of any referenced object are stored in the object's `$instance_vars` attribute. It contains an associative array mapping variable names to their values. The instance variables of immediate values (which are represented as non-objects in PHP) are stored in the `$r2p_value_objects_instance_vars` global variable. It contains an associative array of associative arrays mapping object ID and a variable name to its value.

Instance variable reads and writes are transformed into `r2p_instance_var_get_in_context` and `r2p_instance_var_set_in_context` PHP runtime calls in a similar way as for other variable types. Both functions automatically determine the object to operate on from the current `self` value (thus the `_in_context` suffix).

At runtime, **class variables** of any class are stored in the class's `$class_vars` attribute. It contains an associative array mapping variable names to their values.

Class variable reads and writes are transformed into `r2p_class_var_get_in_context` and `r2p_class_var_set_in_context` PHP runtime calls in a similar way as for other variable types. Both functions automatically determine the class to operate on from the current `self` value and their position in the source code (thus the `_in_context` suffix).

At runtime, **constants** of any class are stored in the class's `$consts` attribute. It contains an associative array mapping constant names to their values.

Class variable reads and writes are transformed into `r2p_const_get_in_context` and `r2p_const_set_in_context` PHP runtime calls in a similar way as for other variable types. Both functions automatically determine the class to operate on from the current `self` value and their position in the source code (thus the `_in_context` suffix).

5.2.3.3 Pseudo-variables

The `self` **pseudo-variables** are transformed into `r2p_self` PHP runtime calls, which retrieve the value of `self` from the self stack, placed in `$r2p_self_stack` global variable at runtime.

The `nil`, `true` **and** `false` **pseudo-variables** are transformed into `R2P_NIL`, `R2P_TRUE` and `R2P_FALSE` constants, which contain PHP values `null`, `true` and `false`.

The `__FILE__` **pseudo-variables** are transformed as if they were a Ruby string literals containing the current source file information. The string value is obtained from a node attribute (all Ruby AST nodes contain source file and line information).

The `__LINE__` **pseudo-variables** are transformed in a similar way as `__FILE__`, but they are integers.

5.2.3.4 Assignments

Assignments in Ruby can be classified as *simple assignments* or *parallel assignments*. Simple assignments have only single item on either side of the assignment expression. Parallel assignments have more than one item on left-hand-side (LHS), right-hand-side (RHS), or both sides of the assignment expression. Precise semantics of the parallel assignment is quite complicated and it is described in Section 4.5 in [33].

The **simple assignments** are transformed into a `r2p_global_var_set`, `r2p_local_var_set`, `r2p_instance_var_set_in_context`, `r2p_class_var_set_in_context`, `r2p_const_set_in_context` PHP runtime calls, or a PHP runtime call implementing a Ruby method call, depending on the type of the assigned value. The template of the call is generated by the node containing the assignment LHS. If the RHS contains a splat operator, the `r2p_pack_assigned_values` PHP runtime function is called on it before assigning to emulate applying of the operator.

The **parallel assignments** are transformed into a series of PHP statements and PHP runtime calls, which build an array of assigned values and then assign its items into the values on the LHS. The assignments of the values are implemented using the same mechanism as in case of simple statements.

5.2.3.5 Block Expressions

The **block expressions** (`begin/end`) are transformed into a sequence of PHP statements representing the transformed Ruby statements inside the code block.

If the block expression contains exception-handling statements (`rescue` and `else`), it is transformed differently, though. For a description, see Section 5.2.3.14.

5.2.3.6 Conditional Statements

The Ruby `if` **statements** are transformed into PHP `if` statements. The conditions are checked using the `r2p_true_like` PHP runtime function, which emulates the Ruby behavior that `nil` and `false` are the only values considered as boolean false and all other values are considered as boolean true.

The Ruby `else` **clauses** are transformed into PHP `else` clauses.

The Ruby `elsif` **clauses** are transformed into nested Ruby `if` statements by the parser and then into nested PHP `if` statements by the transformer. Although it would be possible to use PHP `elseif` statement as a translation of Ruby `elsif`, the transformation into a nested `if` simplifies the Ruby AST.

The Ruby `unless` **statements** are transformed into Ruby `if` statements with a negative condition by the parser and then into PHP `if` statements by the transformer.

Tail forms of the `if` and `unless` statements are represented in the same way as their normal forms in the Ruby AST and they are transformed as such.

5.2.3.7 Looping Statements

The Ruby `while` **statements** are transformed into PHP `while` statements in normal cases and into PHP `do...while` statements in case of tail form with `begin`/`end` code block as a body. The conditions are checked using the `r2p_true_like` PHP runtime function.

If the condition is an expression, it is checked directly in the condition of the PHP `while` (or `do...while`) statement. Otherwise, a `while(true) {...}` (or `do {...} while (true)`) cycle is generated, the condition is evaluated and checked in the beginning of the cycle body and the `break` statement is used to terminate the cycle if the condition is not satisfied.

The Ruby `until` **statements** are transformed into Ruby `while` statements with a negative condition by the parser and then into PHP `while` statements by the transformer.

Tail forms of the `while` and `until` statements are represented in the same way as their normal forms in the Ruby AST and they are transformed as such.

5.2.3.8 `break` and `next` Statements

Inside a cycle, the Ruby `break` **statements** are transformed into PHP `break` statements. Because there may be dummy cycles generated inside the PHP cycle representing a Ruby cycle (see Section 5.2.3.14), `break` ***n*** form may be generated to break to the correct level.

Outside a cycle, the Ruby `break` statements are transformed into `r2p_raise` PHP runtime calls that raise `LocalJumpError` Ruby exception.

The information about the statement placement and the dummy cycles is obtained from the context object passed to the `transform` method.

The Ruby `next` **statements** are transformed in a similar way to Ruby `break` statements, but the PHP `continue` statements are generated instead of PHP `break` statements.

5.2.3.9 Method Definitions, Redefinitions and Undefinitions

The **instance method definitions** are transformed into a PHP function definitions followed by `r2p_define_instance_method` PHP runtime calls.

The transformed PHP function contains transformed statements of the Ruby method, wrapped between the `r2p_method_enter` and `r2p_method_leave` PHP runtime calls. It

is not declared with any formal parameters. The function name is created to be unique (it uses the ID of the Ruby AST node containing the method definition) and has no resemblance to the original method name.

The `r2p_method_enter` function accepts method parameter description and actual parameters passed to the PHP function (obtained using `func_get_args` PHP function). It uses the description to assign passed parameters into local variables visible throughout the function. It also creates a new local variable scope and sets the correct value of `self`.

The `r2p_method_leave` function restores previous local variable scope and a value of `self`.

The `r2p_define_instance_method` function performs certain checks and then registers the generated PHP function as an implementation of given method at the active class.

The **singleton method definitions** are transformed in the same way as instance method definitions except one difference: `r2p_define_singleton_method` PHP runtime calls are used instead of `r2p_define_instance_method`.

The **method undefinitions** are transformed into `r2p_undefine_methods` PHP runtime calls.

5.2.3.10 Method Invocations

Method invocations in Ruby can be classified as *right-hand side* (RHS) or *left-hand side* (LHS). RHS method calls are on the right-hand side of the assignment expression or not inside an assignment expression at all. LHS method calls are on the left-hand side of the assignment expression. Both kinds of invocations are translated differently.

Note that some Ruby constructs are in fact masked method calls. Listing 5.4 illustrates some of them together with the RHS/LHS distinction. For more information, see Section 4.5.3 in [33].

Listing 5.4: Ruby program containing various statements that are in fact method calls.

```
girl.name = "marci".capitalize # RHS call of the "capitalize" method
                               # on a string with no parameters;
                               # LHS call of the "name=" method on the
                               # "girl" variable with one parameter
                               # (the capitalized string).

matrix[1,2] = 5                # LHS call of the "[]" method on the
                               # "matrix" variable with three
                               # parameters(1, 2 and 5).

puts list[index]               # RHS call of the "[]" method on the
                               # "list" variable with one parameter
                               # ("index" variable) and RHS call of the
                               # "puts" method on the top level object
                               # with one parameter (the list item).
```

The **RHS method invocations** with an explicit receiver, no associated block and no splat operator are transformed into `r2p_call` PHP runtime calls. Parameters of these calls are the method call receiver, the method name and parameters of the method call

(the `r2p_call` function has a variable parameter count to accommodate parameters of any Ruby method call). The `r2p_call` function calls other PHP runtime functions internally, which ensure that the PHP function implementing the method will be called, if the receiver handles the method. The behavior exactly matches Ruby method dispatch semantics.

If the invocation contains a splat operator, the last parameter is passed to the `r2p_expand_splat` function that emulates Ruby splat expansion. Resulting object (a PHP array) is merged with the positional parameters using `array_merge` PHP function, producing another PHP array. The `r2p_call` function is then called using `call_user_func_array` PHP function, which allows the PHP runtime to pass it the parameters using that array.

If the invocation does not contain an explicit receiver, the method calls are transformed into `r2p_call_without_receiver` or `r2p_call_maybe_local_var` calls. The second function is used when there is a possibility that the method call may be in fact a local variable (see Section 4.4 in [33] for description of this ambiguity). Both functions cause the PHP runtime to behave a bit differently when the receiver cannot handle the called method (for details, see the implementation of the `Kernel#method_missing` method in the `/runtime/lib/modules/kernel.php` file in the PHP runtime source code). In the generated code, nothing changes from the explicit receiver case except the function name.

If the invocation contains an associated block, the `_with_block` suffix is appended to the name of the PHP runtime function handling the invocation and the block variable is passed as its third parameter (after the method name). The `_with_block` version of the runtime functions ensure that the block is placed on the PHP runtime stack emulating Ruby call stack. For description of block variables, see Section 5.2.3.12.

The **LHS method invocations** are transformed in a similar way as variable writes—the `transform` method creates a template of the PHP runtime function call (the function name and parameters are determined in the same way as in RHS method invocations). This template is filled-out with the value set and the call is actually generated by the parent node in the AST, which is always an assignment.

5.2.3.11 `return` and `super` Statements

Inside a method, the Ruby **`return` statements** are transformed into PHP `return` statements, preceded with `r2p_method_leave` PHP runtime call. The call is needed to pop the current local variable scope and restore the previous value of `self` before exiting the function.

Outside a method, the Ruby `return` statements are transformed into `r2p_raise` PHP runtime calls that raise `LocalJumpError` Ruby exception.

The information about the statement placement is obtained from the context object passed to the `transform` method.

The **`super` statement invocations** with no associated block and no splat operator are transformed into `r2p_super` PHP runtime call. Parameters of these calls are the parameters of the `super` statement (the `r2p_super` function has a variable parameter count to accommodate parameters of any `super` statement). The `r2p_super` function calls other runtime functions internally that emulate Ruby `super` statement behavior.

The situation when the `super` statement contains a splat operator or has an associated block are resolved in the same way as in method invocations (see Section 5.2.3.10).

5.2.3.12 Blocks and the `yield` Statement

The **blocks** associated with method calls are transformed into block variable definitions followed by definitions of PHP functions representing the blocks.

The block variable definition is an assignment which defines a *block variable*. This variable contains information about the block—in the current implementation only a name of the PHP function representing the block. This variable is placed to the PHP runtime stack emulating Ruby call stack by a PHP runtime function handling a method call with an associated block. The variable name is created to be unique (it uses the ID of the Ruby AST node containing the block).

The PHP function contains transformed statements of the block wrapped between `r2p_block_enter` and `r2p_block_leave` PHP runtime calls. It is not declared with any formal parameters. The function name is created to be unique (it uses the ID of the Ruby AST node containing the block). Assignment to the block parameters are emitted after the `r2p_block_enter` call.

The `r2p_block_enter` function creates a new local variable scope and sets the correct value of `self`.

The `r2p_block_leave` function restores previous local variable scope and a value of `self`.

The `yield` **statements** with no splat operator are transformed into `r2p_yield` PHP runtime calls. Parameters of these calls are the parameters of the block invocation (the `r2p_yield` function has a variable parameter count to accommodate parameters of any block invocation). The `r2p_yield` function calls other PHP runtime functions internally, which ensure that the PHP function implementing current block will be called. The behavior exactly matches Ruby yielding semantics.

The situation when the `yield` statement contains a splat operator is resolved in the same way as in method invocations (see Section 5.2.3.10).

5.2.3.13 Class and Module Definitions and Redefinitions

The **class definitions** are transformed into `r2p_begin_class_definition` and `r2p_end_class_or_module_definition` PHP runtime calls, which wrap the transformed statements inside the class definition.

The `r2p_begin_class_definition` function makes some checks first. If they do not catch any problems, the behavior of the function depends on the situation:

- If the class is *defined* (i.e. opened for the first time), the function creates a new `R2PClass` instance for the defined class, it creates a constant for it (with a name equal to the class name) and sets it as the current class.
- If the class is *reopened* (i.e. it was defined already), the function finds the corresponding `R2PClass` instance and sets it as the current class.

In both cases, the function creates a new local variable scope and sets a value of `self` to the defined class.

The setting of `self` and current class ensures that methods defined inside the class definition will be added to the defined class and that constant and class variable lookup will work correctly.

The `r2p_end_class_or_module` function restores previous local variable scope, current class and the value of `self`.

The **metaclass definitions** and **module definitions** are transformed in the same way as casual classes. The only significant difference is that `r2p_begin_metaclass_definition` and `r2p_begin_module_definition` PHP runtime calls are used instead of `r2p_begin_class_definition`.

5.2.3.14 Exception Raising and Handling

The `raise` **statement** is in fact a method defined in the `Kernel` module. It is implemented in the PHP runtime and the transformer treats its invocation as any other method call. When invoked, the method sets the exception flag and saves the raised exception into a global variable.

The compiler generates *exception checks* after each construction that might raise an exception (e.g. method calls, local variable reads, etc.). It is a simple `if` statement that checks the exception flag and interrupts the statement sequence—using a `break` statement inside a rescued statements (see below) or a `return` statement everywhere else. The chain of exception checks placed after method calls causes stack unrolling as if real exceptions were used.

The information about the exception check placement is obtained from the context object passed to the `transform` method.

The `rescue` **clauses** are transformed into wrappers around rescued statements and series of tests that match the `rescue` clauses and check if any exception should be caught by the handler.

The rescued statements are wrapped inside a `do {...} while(false)` cycle. If no exception is raised in the rescued statements, the cycle will not play any role in the program execution. If any exception is raised, it will be detected by the nearest exception check and a break statement will be issued, skipping all remaining statements in the cycle and transferring control to the following exception handler (the transformed `rescue` clauses).

If a rescue clause does not contain an exception class, it catches exceptions of the `StandardError` class and its subclasses. The raised exception is tested using `r2p_raised_exception_is_standard_error` PHP runtime call in this case.

If a rescue clause contains one or more exception classes, it catches exception of these classes or their subclasses. The raised exception is tested using `r2p_raised_exception_is_a` or `r2p_raised_exception_is_one_of` PHP runtime calls in this case.

If a rescue clause contains a variable, an assignment is generated after the exception class check, which assigns the raised exception into that variable.

Tail form of the `rescue` clause is represented in the same way as its normal form in the Ruby AST and it is transformed as such.

The `else` **clauses** are transformed by the parser already. Their statements are simply appended to the rescued statements.

5.3 Serializer

The serializer is the final step in the compilation chain. It serializes the PHP AST into a PHP source code. It does this job by a simple recursive walk through the tree, during which it emits appropriate PHP statements for the nodes. To aid debugging, the generated code is properly indented and whitespace is used quite liberally.

Technically, the serializer is implemented in a very similar way to the transformer. It extends the AST node classes by defining an `emit` method, which emits the PHP code for given node and recursively calls itself on its descendants as needed.

Chapter 6

Related Work

While many translators between static languages exist, translators between dynamic languages are far less common. The main reason is probably that the dynamic languages are not used as widely as static languages and also that writing a dynamic language translator is more difficult due to the nature of the dynamic languages.

This chapter describes several projects which are similar to the Ruby to PHP translator in some way—they either compile a dynamic language into another language or use PHP as the target language of the translation.

6.1 HotRuby

The most closely related project is HotRuby [26], which is a Ruby virtual machine written in JavaScript, in effect serving as a compiler of Ruby into JavaScript. The approach used is very different from the translator's strategy—HotRuby uses a parser and bytecode compiler contained in Ruby 1.9 and only implements the bytecode interpreter and supporting runtime code on the JavaScript side. This decision allowed its author to avoid reimplementing the Ruby parser. For details about the HotRuby project, see Section 2.5.5.6.

Other Ruby implementations than HotRuby are either interpreters or compilers and they do not translate Ruby into other language.

6.2 Python to OCaml Compiler

An interesting project is the Python to OCaml compiler developed by Raj Bandyopadhyay (together with Walid Taha and Ken Kennedy) as a part of a research at the Rice University. The idea is to write an efficient compiler for Python without the need to write its backend. The resulting implementation uses OCaml as an intermediate language and reuses its compiler for compilation into the native code. OCaml was chosen mainly because of its good memory management runtime, highly expressive type system, ease of translation and the foreign function interface, which allows it to integrate well with external libraries. [34]

The compiler itself is written in OCaml and has a very similar structure to the Ruby to PHP translator: the Python source code is parsed into an AST, this AST is transformed into OCaml AST, and finally the OCaml code is emitted, which can be compiled by the OCaml compiler into the native code.

At runtime, a Python runtime environment written in OCaml is used. It is modeled after the original Python implementation and it uses several native OCaml classes to implement Python built-in classes (such as dictionaries or arrays). It is functionally similar to the PHP runtime in the Ruby to PHP translator.

According to the author, the challenges of the project were mainly the Python language dynamism, rich function call protocol and exception handling (which requires the stack traces). [34] These challenges are similar to the challenges met when developing the Ruby to PHP translator.

6.3 haXe

haXe [35] is an attempt to design an open source language similar to Java or C#, but with some modern features such as closures or type inference. The haXe language has an unique approach to the compilation—the source code can be translated into JavaScript, ActionScript 3, Flash (`.swf` file), PHP or a bytecode for the Neko virtual machine [36]. This allows to use haXe in many different contexts (client-side browser programming, server-side web applications, etc.) while avoiding binding to any specific platform.

The haXe language is strongly and statically typed, but allows for certain dynamic constructs and reflection. In most aspects (including syntax), it is similar to Java or C#, but it contains many "modern" features usually present in dynamic languages, such as first-class functions, closures or regular expression literals. It offers an advanced type system supporting generics, type inference, structural subtyping, anonymous types, polymorphic methods and other features.

6.3.1 PHP Backend

The generated PHP code supports all the haXe language and requires at least PHP 5.1.3 for running. It is quite similar to the original haXe code (e.g. haXe statements are mapped to equivalent PHP statements, haXe classes and methods are mapped to PHP classes an methods). This is a direct consequence of the haXe's nature (which is not nearly as flexible as Ruby) and it is very different from the Ruby to PHP translator, where many Ruby constructs must be emulated indirectly in PHP.

The haXe compiler uses a runtime environment similar to the PHP runtime to provide libraries and other necessary support code to the translated program. This environment is much more object-oriented than the PHP runtime, which is partially caused by the design of the haXe language and also by the chosen minimal supported PHP version. In the Ruby to PHP translator implementation, several design decisions that precluded full object orientation of the PHP runtime were made and the original Ruby implementation, which is not written in object-oriented fashion, was partially followed (see Section 4.1.2 for details).

6.4 Summary

From the presented projects, it is clearly visible that writing a translator of a dynamic language and a static language are very different tasks. The author of the Python to OCaml compiler had to overcome similar difficulties as the thesis author and his approach to the task was similar. On the other hand, the translation of the static haXe language into PHP seems to be a relatively simple task and the language constructs are mapped quite directly into PHP. In all cases, a supporting runtime environment written in the target language was needed.

Chapter 7

Conclusion

The primary objective of this work was to design and implement a compiler translating a significant subset of the Ruby language into PHP, with emphasis on the correct translation of dynamic and functional language elements and compatibility with the original Ruby language implementation.

This goal was successfully reached. A practically usable compiler was created demonstrating that a very high-level and dynamic language such as Ruby can be successfully implemented using another dynamic language. Part of the original motivating idea—running Ruby applications on a web server—also materialized: the implementation allows to translate simple Ruby programs into PHP, upload them on a web server and run in that environment like any other PHP code.

The generated PHP code is very different from the original Ruby code and requires a presence of the PHP runtime with supporting functionality. Both traits are a direct result of the Ruby language dynamism and its reliance on the core class library for many basic tasks.

During the implementation, most of the time was spent on studying source code of other Ruby implementations (mainly the original one) in order to understand details of the interpreter behavior, which are not documented anywhere. Unexpected number of inconsistencies, strange edge-cases and other "surprises" was found when examining the internals. Most of this behavior was closely replicated in the translator. The result of this painstaking work is that the translator is closely compatible with the original implementation and it supports its many features.

The only thing to regret is that the officially-blessed book containing a thorough language description, *The Ruby Programming Language* [33], was published in January 2008 and author obtained it several months later, when the implementation was almost finished. Many hours of work could be saved had it been available earlier.

7.1 Inherent Limitations

Although it is possible for the compiler and PHP runtime to implement almost all Ruby language in a way compatible with the original Ruby implementation, there are certain inherent limitations which preclude full implementation of some features.

The most important limitation is absence of threads in PHP. They are not supported by the language and as far as the author knows, their support cannot be added by any library or extension. This means that the `Thread` and `ThreadGroup` core library classes, which encapsulate threading in Ruby, must remain unimplemented and correct translation of multithreaded programs is currently not possible.

Another limitation is PHP's inability to support continuations. Continuations require explicit call stack manipulation—a feature PHP does not offer (in PHP, the call stack can be manipulated only implicitly by calling functions or methods, returning from them and by throwing exceptions). This means that the `Continuation` core library class and the `Kernel#callcc` method, which encapsulate continuations, must remain unimplemented and correct translation of programs using continuations will not ever be possible.[1]

The core library classes `IO` and `File` provide some methods that do not have their counterparts in PHP. These are mainly low level methods such as `IO#ioctl`. Implementation of these methods is simply impossible in unmodified PHP. The only solution would be to write a PHP extension which would implement counterparts to these methods using the C standard library and make them visible in PHP. This solution is far from ideal.

7.2 Future Work

Although the translator in its current state compiles most of the Ruby language, there is a room for improvement and future work. The author's intention is to continue to improve the translator after the thesis finalization, to open the source code and possibly accept contributions from others.

7.2.1 Increasing Ruby Language Coverage

One direction in which the translator can be improved is adding support for unimplemented constructs of the Ruby language. Current version does not support some constructs at all (e.g. `for` and `case` statements[2]) and it supports some constructs only partially (e.g. the `return` statement presently cannot take multiple parameters, `break`/`next` statements cannot take parameters at all).

Another obvious area for future improvement is the Ruby core class library. Although current translator version already supports more than one third of methods in the library classes, the rest remains to be implemented.

7.2.2 Optimization

At the beginning of the thesis, an example of a Ruby code which is hard to optimize (Listing 3.1) was presented together with an explanation, why it was chosen not to optimize

[1] Ruby implementations implemented on the top of the Java and .NET virtual machines do not support continuations either, for the same reasons as the translator. The continuation support must be hard-wired into the underlying virtual machine.

[2] Lack of support of seemingly essential statements like `for` and `case` may seem surprising. However in real Ruby applications, the `for` statement is almost never used (it is usually replaced by the functional constructs) and the `case` statement can be easily replaced by a series of `if` statements.

the generated code significantly in this work. But this does not mean that the generated code it is not worth optimizing at all.

There are two levels of possible optimization: *algorithmic* and *implementation*.

On the algorithmic level, there are several possibilities to optimize the generated code with well-known optimization techniques (such as method caching, which is often employed in dynamic language implementations to speed up the method dispatch mechanism), along with simple streamlining of specific code sequences (for example, in certain situations lots of unnecessary temporary variables are created in the complier-generated PHP code, which could be eliminated with some additional work).

Optimization on the implementation level would require using some PHP profiler tool to find performance bottlenecks in typical translated programs and devising optimization techniques to remove them.

7.2.3 Ruby and PHP Integration

Integration of the generated code with PHP is not perfect at this point. It simply was not the main goal of the thesis (see Chapter 1). Despite this, it is possible to easily embed translated Ruby programs into larger PHP applications and use their features. The opposite—use of PHP code from the translated Ruby program—requires wrapping the PHP code into objects and classes compatible with the translator's PHP runtime. This could be remedied in the future by extending the PHP runtime to handle normal PHP objects.

7.2.4 Multiple Platform Support

The translator is currently only supported on the Linux operating system. During the development, it was also tested on Windows and Mac OS X platforms, but compatibility issues appeared on both platforms.

On Windows, the translator itself appears to work, but the regression testing framework used in the implementation could not be made to run because of bugs in the Ruby libraries emulating common Unix system calls. As a result, the translator correctness cannot be guaranteed. The solution would be to redesign and rewrite the testing framework.

On Mac OS X, the installed PHP version does not contain the `bcmath` built-in library for handling arbitrarily big numbers, which is used in the implementation of the `Bignum` Ruby class. The workaround would be to rewrite the implementation to make it independent on this library. Other than that, the translator appears to work correctly.

Problems encountered on the Windows and Mac OS X platforms could not be resolved in a timely manner and required declaring other platforms than Linux unsupported. However the issues found are not insolvable and generally require only certain amount of time. Theoretically, the translator should be able to run on any platform supporting both Ruby and PHP.

7.2.5 Other Possible Improvements

Some parts of the translator code (mainly the lexer and parser) are based on the original Ruby implementation. The code structure of these parts is not ideal (it copies bad structure of the original implementation) and could be improved.

During the translator development, the implementation was sometimes hindered by maintaining PHP 4 compatibility. In the future, dropping PHP 4 support could be considered. This would open up the possibility of mapping Ruby exceptions into PHP exceptions directly, the `R2PRef` "hack" (see Section 4.1.2.1) would become unnecessary and various other improvements to the PHP runtime code would be possible.

References

[1] Ruby Visual Identity Team: Ruby Programming Language.
http://ruby-lang.org/, retrieved on 2008-07-15.

[2] The PHP Group: PHP: Hypertext Preprocessor.
http://www.php.net/, retrieved on 2008-07-15.

[3] 37signals, Inc.: Ruby on Rails.
http://www.rubyonrails.com/, retrieved on 2008-07-15.

[4] Reenskaug T. (1979): Thing-Model-View-Editor.
http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf, retrieved on 2008-07-15.

[5] Reenskaug T. (1979): Models-Views-Controllers.
http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf, retrieved on 2008-07-15.

[6] Fowler, M. (2003): *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston.

[7] Phusion: Phusion Passenger.
http://www.phusion.nl/, retrieved on 2008-07-15.

[8] Ford B. et al.: RubySpec.
http://www.rubyspec.org/, retrieved on 2008-07-15.

[9] Cake Development Corporation, Inc.: CakePHP.
http://www.cakephp.org/, retrieved on 2008-07-15.

[10] Matsumoto, Y. (2008): EURUKO 2008 Keynote.
http://www.euruko2008.org/system/assets/documents/0000/0010/matsumoto-ruby-past-future-and-present-euruko08.pdf, retrieved on 2008-07-21.

[11] Kalnbach, K. (2008): Ruby 1.8 vs. 1.9 Benchmarks.
http://www.rubychan.de/share/yarv_speedups.html, retrieved on 2008-07-15.

[12] JRuby team: JRuby.
http://jruby.codehaus.org/, retrieved on 2008-07-15.

[13] Nutter, Ch. (2007): Paving the Road to JRuby 1.0: Performance.
http://headius.blogspot.com/2007/04/paving-road-to-jruby-10-performance.html, retrieved on 2008-07-15.

[14] Microsoft, Inc.: IronRuby.
http://www.ironruby.com/, retrieved on 2008-07-15.

[15] Phoenix, E. et al.: Rubinius: The Ruby Virtual Machine.
http://rubini.us/, retrieved on 2008-07-15.

[16] GemStone Systems, Inc.: MagLev – Ruby that scales.
http://ruby.gemstone.com/, retrieved on 2008-07-15.

[17] Apple, Inc.: MacRuby.
http://www.macruby.org/, retrieved on 2008-07-15.

[18] The RubyCocoa Project: RubyCocoa.
http://rubycocoa.sourceforge.net/, retrieved on 2008-07-15.

[19] Zhi, X. Y. et al.: XRuby.
http://xruby.com/, retrieved on 2008-07-15.

[20] Zhi, X. Y. (2008): Re: Questions about XRuby. Thesis author's correspondence with Zhi, X. Y.

[21] Zhi, X. Y. (2007): XRuby is faster than Ruby 1.8.5 in most benchmarks.
http://xruby.blogspot.com/2007/03/xruby-runs-most-benchmark-faster-than.html, retrieved on 2008-07-15.

[22] Kelly, W. et al.: Ruby.NET.
http://rubydotnet.googlegroups.com/web/Home.htm, retrieved on 2008-07-15.

[23] Kelly, W. (2008): The future of Ruby.NET.
http://groups.google.com/group/RubyDOTNET/browse_thread/thread/1752830c857620b0, retrieved on 2008-07-15.

[24] Tew, K.: Cardinal: Ruby Interpreter for Parrot.
http://cardinal2.rubyforge.org/, retrieved on 2008-07-15.

[25] Nilsson, N. (2007): Is it too late for Parrot VM?
http://www.infoq.com/news/2007/09/is-it-too-late-for-parrot, retrieved on 2008-07-15.

[26] Kobayashi, Y.: HotRuby – Ruby on JavaScript & Flash.
http://hotruby.accelart.jp/, retrieved on 2008-07-15.

[27] Mozilla Foundation: IronMonkey.
http://wiki.mozilla.org/index.php?title=Tamarin:IronMonkey&oldid=65786, retrieved on 2008-07-15.

[28] Aoki, M.: Racc.
http://i.loveruby.net/en/projects/racc/, retrieved on 2008-07-15.

[29] Feldt, R.: rockit.
http://rockit.sourceforge.net/, retrieved on 2008-07-15.

[30] Mössenböck, H., Löberbauer, M., Wöß, A.: The Compiler Generator Coco/R.
http://www.ssw.uni-linz.ac.at/coco/, retrieved on 2008-07-15.

[31] Grosse, H.: ruby-yacc.
http://raa.ruby-lang.org/project/ruby-yacc/, retrieved on 2008-07-15.

[32] Gamma, E. et al., (1995): *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston.

[33] Flanagan, D., Matsumoto, Y: (2008). *The Ruby programming language*. O'Reilly, Beijing.

[34] Bandyopadhyay, R. (2007): Compiling Dynamic Languages.
http://video.google.com/videoplay?docid=-2077755378178864152, retrieved on 2008-08-02.

[35] Motion-Twin: haXe.
http://www.haxe.org/, retrieved on 2008-08-02.

[36] Motion-Twin: NwkoVM.
http://www.nekovm.org/, retrieved on 2008-08-02.

Appendix A

Installation and Usage

This appendix describes requirements, installation and basic usage of the translator.

A.1 Requirements

A.1.1 Operating System

The translator is currently only supported on the Linux operating system. It was successfully tested with Ubuntu 8.04.1 on the IA-32 architecture.

During the development, the translator was also tested on Windows and Mac OS X platforms, but compatibility issues appeared on both. As a result, the translator is able to run on Windows on Mac OS X with some limitations, but it is not supported officially (see Section 7.2.4 for details).

A.1.2 Ruby

The translator requires Ruby 1.8.6 installed together with *rake*, *activesupport* and *diff-lcs* gems.[1] It was successfully tested with Ruby 1.8.6-p111, rake 0.8.1, activesupport 2.1.0 and diff-lcs 1.1.2.

To install Ruby, visit `http://www.ruby-lang.org/` and follow the installation instructions for your platform. You can also use the package manager of your operating system. Make sure that you install the latest 1.8 version, not the 1.9 development branch (you can always check the installed Ruby version using the `ruby -v` command).

To install the RubyGems tool (needed to install the required gems), visit `http://www.rubygems.org/` and follow the installation instructions in the user guide. Note that if you install Ruby using the one-click installer for Windows, RubyGems tool is preinstalled already.

[1] The *diff-lcs* gem is not needed for normal usage, only for running the translator tests.

To install the required gems, use following commands:

```
gem install -y rake
gem install -y activesupport
gem install -y diff-lcs
```

A.1.3 PHP

The generated PHP programs require PHP 4.4.x or 5.2.x. The translator was successfully tested with PHP 4.4.8 and PHP 5.2.6. Older or newer versions of PHP than specified may work too.

The generated PHP programs are able to run both in PHP executed from the command-line or running on the web server. PHP must be configured to allow call-time pass-by-reference specification (this can be enabled in `php.ini` by setting the `allow_call_time_pass_reference` configuration option to `On`).

To install PHP, visit http://www.php.net/ and follow the installation instructions for your platform in the documentation. You can also use the package manager of your operating system.

A.2 Installation

To install the translator, just copy contents of the `Implementation` directory on the CD to any directory.

A.3 Usage

The translator can be invoked via the `compiler/bin/r2p` script.[2]

The translator accepts the Ruby source code to translate on the standard input and outputs the generated PHP code on the standard output. Alternatively, you can specify a file with the Ruby source code on the command line, in which case the output will be written into a file with the same name but the extension changed to `.php`. Several files with sample programs to translate can be found in the `examples` directory.

For example, to translate the `examples/hello_world.rb` sample program, execute following command:

```
ruby compiler/bin/r2p examples/hello_world.rb
```

The `hello_world.php` file will be created, which can be executed by PHP:

```
php examples/hello_world.php
```

[2] To execute the script in Unix environment, just type its name in the shell. On Windows, you have to execute it using the `ruby` command (e.g. `ruby compiler/bin/r2p`).

The translator accepts several command-line options. For a full list, see the translator help, which can by displayed using the `--help` command-line option.

Note that the generated PHP programs require the PHP runtime, whose PHP source code can be found in the `runtime/lib` directory. The translator embeds correct relative path to the PHP runtime into the generated PHP files.[3] When copying the generated PHP files into different location, either copy the PHP runtime too or edit the path to the PHP runtime inside the generated PHP files.

[3] This can be overridden using the `--runtime-path` command-line option.

Appendix B

Supported Features

This appendix lists Ruby features supported by the compiler and the PHP runtime. Because Ruby depends heavily on its core class library, the appendix also includes a listing of implemented classes and modules from this library, along with a list of supported predefined global constants[1] (Table B.1) and variables (Table B.2).

In general, the translator is compatible with Ruby 1.8.6-p114.

B.1 General Limitations

For simplicity, the compiler and the PHP runtime assume 32-bit little-endian architecture (such as IA-32). Adding support for other architectures should not be hard, as there are only few architecture-dependent places in the code and they are all clearly marked.

Generally, Ruby programs can be written in 7-bit ASCII, Kanji (using EUC or SJIS), or UTF-8. To simplify the implementation, the translator officially supports only 7-bit ASCII. However non-ASCII characters are preserved in the string literals and ignored in the comments, so in fact, most reasonable encodings (such as Windows 1250, ISO-8859-2 or UTF-8) can be used.

Many limitations specific to certain language constructs or methods in the core library are described in the footnotes.

B.2 Language Elements

The compiler supports following Ruby language elements.

- Literals for classes `Fixnum`, `Bignum`, `Float`, `Symbol`, `String`, `Array`, `Hash` and `Range`[2], including all alternate syntaxes

[1] Strictly speaking, there are no global constants in Ruby, as all constants are defined in some class. We define global constants as constants defined in the `Object` class without constants representing Ruby modules and classes.

[2] So called "flip-flops" are not supported.

- Operators (both built-in and method-based)
- Local, global, instance and class variables
- Constants
- Pseudo-variables `self`, `nil`, `true`, `false`, `__FILE__` and `__LINE__`
- Assignments, including parallel assignments and splat operator[3]
- Block expressions (`begin`/`end`)
- Conditional statements (`if`/`elsif`/`else` and `unless`/`else`, including their tail forms)
- Looping statements (`while` and `until`, including their tail forms)
- `break` and `next` statements[4]
- Method definitions (`def`), redefnitions and undefinitions (`undef`) with positional and rest parameter specification[5]
- Method visibility setting and enforcing
- Singleton methods
- Method invocations with positional and rest parameter passing
- `return`[6] and `super`[7] statements
- Blocks and the `yield` statement
- Class and module definitions and redefinitions (`class`, `module`)
- Singleton classes
- Exception raising (`raise`) and handling (`rescue`/`else`, including `rescue` tail form)
- `BEGIN` and `END` blocks
- End of the source code directive (`__END__`)[8]
- Comments

When the parser encounters an unsupported elements, it reports an error with appropriate message for the user and the compilation is halted.

[3]Qualified constant assignments and assignments using `&&=` and `||=` operators are not supported.

[4]The `break` and `next` statements can be used only in cycles and without any parameters. In Ruby, both statements can be used in blocks too and they accept parameters.

[5]Nested method definitions are not supported.

[6]The `return` statement can return only one value and it is allowed only in methods. In Ruby, multiple values can be returned, the splat operator can be used and the `return` statement can be used in blocks too.

[7]The `super` statement cannot be invoked from blocks. There is no such limitation in Ruby.

[8]Lines after the `__END__` directive are not available to the script. In Ruby, they can be read using the global `IO` object `DATA`.

B.3 Core Classes

B.3.1 ArgumentError

Does not define any constants nor methods.

B.3.2 Array

Public methods: `&`, `*`, `+`, `-`, `<<`, `<=>`, `==`, `[]`[9], `[]=`[10], `assoc`, `at`, `clear`, `collect`, `collect!`, `compact`, `compact!`, `concat`, `delete_at`, `each`, `each_index`, `empty?`, `first`, `hash`, `include?`, `index`, `insert`, `inspect`, `join`, `last`, `length`, `map`, `map!`, `nitems`, `pop`, `push`, `rassoc`, `replace`, `reverse`, `reverse!`, `rindex`, `shift`, `size`, `sort`[11], `sort!`[12], `to_a`, `to_ary`, `to_s`, `transpose`, `uniq`, `uniq!`, `unshift`, `|`

Private methods: `initialize`[13]

B.3.3 Bignum

Public methods: `%`, `&`, `*`, `+`, `-`, `-@`, `/`, `<=>`, `==`, `^`, `abs`, `coerce`, `div`, `eql?`, `hash`, `modulo`, `size`, `to_f`, `to_s`, `|`, `~`

B.3.4 Class

Public methods: `allocate`, `new`, `superclass`

Private methods: `initialize`[14]

B.3.5 EOFError

Does not define any constants nor methods.

B.3.6 Exception

Class methods: `exception`

Public methods: `backtrace`, `exception`, `inspect`, `message`, `set_backtrace`, `to_s`, `to_str`

Private methods: `initialize`

[9] The `[]` method accepts only one numerical index.
[10] The `[]=` method accepts only one numerical index.
[11] The `sort` method does not protect against array modification during the call.
[12] The `sort!` method does not protect against array modification during the call.
[13] The `initialize` method always initializes an empty array.
[14] The `initialize` method does not protect against reinitialization for architectural reasons.

B.3.7 FalseClass

Public methods: `&`, `^`, `to_s`, `|`

B.3.8 fatal

Does not define any constants nor methods.

B.3.9 File

Constants: `PATH_SEPARATOR`, `SEPARATOR`, `Separator`

B.3.10 Fixnum

Public methods: `%`, `&`, `*`, `+`, `-`, `-@`, `/`, `<`, `<=`, `<=>`, `==`, `>`, `>=`, `^`, `abs`, `div`, `divmod`, `id2name`, `modulo`, `quo`, `size`, `to_f`, `to_s`, `to_sym`, `zero?`, `|`, `~`

B.3.11 Float

Public methods: `%`, `*`, `+`, `-`, `-@`, `/`, `<`, `<=`, `<=>`, `==`, `>`, `>=`, `abs`, `ceil`, `coerce`[15], `divmod`, `eql?`, `finite?`, `floor`, `hash`[16], `infinite?`, `modulo`, `nan?`, `round`, `to_f`, `to_i`, `to_int`, `to_s`[17], `truncate`, `zero?`

B.3.12 FloatDomainError

Does not define any constants nor methods.

B.3.13 Hash

Public methods: `==`[18], `[]`, `[]=`, `clear`, `default`, `default=`, `default_proc`, `delete`, `each`[19], `empty?`, `has_key?`, `has_value?`, `include?`, `index`[20], `inspect`[21], `invert`, `key?`, `keys`, `length`, `member?`, `rehash`, `size`, `to_a`, `to_hash`, `to_s`, `value?`[22], `values`, `values_at`

Private methods: `initialize`

[15] The `coerce` method does not convert from strings as in Ruby.

[16] The `hash` method uses hashing algorithm form Ruby 1.8.5, as it is more reliable in the absence of unsigned integer type in PHP than the algorithm used in version 1.8.6.

[17] The `to_s` method outputs numbers in slightly different format than Ruby.

[18] The `==` method does not protect against modification or rehashing during the call.

[19] The `each` method does not protect against modification or rehashing during the call.

[20] The `index` method does not protect against modification or rehashing during the call.

[21] The `inspect` method does not protect against modification or rehashing during the call.

[22] The `value` method does not protect against modification or rehashing during the call.

B.3.14 IOError

Does not define any constants nor methods.

B.3.15 IndexError

Does not define any constants nor methods.

B.3.16 Integer

Public methods: `ceil`, `downto`[23], `floor`, `integer?`, `next`, `round`, `succ`, `times`, `to_i`, `to_int`, `truncate`, `upto`[24]

B.3.17 LoadError

Does not define any constants nor methods.

B.3.18 LocalJumpError

Public methods: `exit_value`, `reason`

B.3.19 Module

Public methods: `==`, `===`, `ancestors`, `class_variable_defined?`, `class_variables`, `const_defined?`, `const_get`, `const_missing`, `const_set`, `constants`, `include?`, `included_modules`, `instance_methods`, `method_defined?`, `name`, `private_instance_methods`, `private_method_defined?`, `protected_instance_methods`, `protected_method_defined?`, `public_instance_methods`, `public_method_defined?`, `to_s`

Private methods: `append_features`, `attr`, `attr_accessor`, `attr_reader`, `attr_writer`, `class_variable_get`, `class_variable_set`, `include`, `initialize`[25], `module_function`[26], `private`[27], `protected`[28], `public`[29], `remove_class_variable`, `remove_const`

B.3.20 NameError

Public methods: `name`, `to_s`

Private methods: `initialize`

[23] The `downto` method is implemented only for `Fixnum`.
[24] The `upto` method is implemented only for `Fixnum`.
[25] The `initialize` method ignores passed block.
[26] The `module_function` variant where method names are passed in the parameters is not implemented.
[27] The `private` variant where method names are passed in the parameters is not implemented.
[28] The `protected` variant where method names are passed in the parameters is not implemented.
[29] The `public` variant where method names are passed in the parameters is not implemented.

B.3.21 NameError::message

Class methods: `!`, `_load`

Public methods: `_dump`[30], `to_str`[31]

B.3.22 NilClass

Public methods:, `&`, `^`, `inspect`, `nil?`, `to_a`, `to_f`, `to_i`, `to_s`, `|`

B.3.23 NoMemoryError

Does not define any constants nor methods.

B.3.24 NoMethodError

Public methods: `args`

Private methods: `initialize`

B.3.25 NotImplementedError

Does not define any constants nor methods.

B.3.26 Numeric

Public methods: `+@`, `<=>`, `eql?`, `integer?`, `nonzero?`, `zero?`

B.3.27 Object

Private methods: `initialize`

B.3.28 Proc

Class methods: `new`

Public methods: `[]`[32], `call`[33], `to_proc`

[30] The `message` and `method` parameters of the `_dump` method must be strings.

[31] The `message` and `method` parameters of the `to_str` method must be strings.

[32] Parameter count of the `[]` method is not checked—excessive parameters are ignored and missing parameters are set to `nil`.

[33] Parameter count of the `call` method is not checked—excessive parameters are ignored and missing parameters are set to `nil`.

B.3.29 Range

Public methods: ==, ===, begin, each, end, eql?, exclude_end?, first, hash, include?, inspect, last, member?, to_s

Private methods: initialize

B.3.30 RangeError

Does not define any constants nor methods.

B.3.31 RegexpError

Does not define any constants nor methods.

B.3.32 RuntimeError

Does not define any constants nor methods.

B.3.33 ScriptError

Does not define any constants nor methods.

B.3.34 SecurityError

Does not define any constants nor methods.

B.3.35 StandardError

Does not define any constants nor methods.

B.3.36 String

Public methods: *, +, <<, <=>, ==, capitalize, capitalize!, casecmp, center, chop, chop!, concat, crypt, downcase, downcase!, dump, empty?, eql?, hash, include?, insert, inspect, intern, length, ljust, next, next!, replace, reverse, reverse!, rjust, size, succ, succ!, swapcase, swapcase!, to_s, to_str, to_sym, upcase, upcase!

Private methods: initialize

B.3.37 Symbol

Class methods: all_symbols

Public methods: ===, id2name, inspect, to_i, to_int, to_s, to_sym

B.3.38 SyntaxError

Does not define any constants nor methods.

B.3.39 SystemExit

Public methods: `status`, `success?`

Private methods: `initialize`

B.3.40 SystemStackError

Does not define any constants nor methods.

B.3.41 ThreadError

Does not define any constants nor methods.

B.3.42 TrueClass

Public methods: `&`, `^`, `to_s`, `|`

B.3.43 TypeError

Does not define any constants nor methods.

B.3.44 ZeroDivisionError

Does not define any constants nor methods.

B.4 Core Modules

B.4.1 Comparable

Public methods: `<`, `<=`, `==`, `>`, `>=`, `between?`

B.4.2 Enumerable

Public methods: `all?`, `any?`, `collect`, `detect`, `each_with_index`, `entries`, `find`, `find_all`, `grep`, `include?`, `inject`, `map`, `max`, `member?`, `min`, `partition`, `reject`, `select`, `sort`, `to_a`

B.4.3 Kernel

Class methods: `block_given?`, `fail`, `iterator?`, `lambda`[34], `method_missing`, `p`[35], `print`[36], `proc`[37], `puts`[38], `raise`, `warn`

Public methods: `==`, `===`, `__id__`, `__send__`, `class`, `eql?`, `equal?`, `hash`, `id`, `inspect`, `instance_of?`, `instance_variable_defined?`, `instance_variable_get`, `instance_variable_set`, `instance_variables`, `is_a?`, `kind_of?`, `methods`, `nil?`, `object_id`, `private_methods`, `protected_methods`, `public_methods`, `respond_to?`, `send`, `singleton_methods`, `to_a`, `to_s`, `type`

Private methods: `block_given?`, `fail`, `iterator?`, `lambda`[39], `method_missing`, `p`[40], `printf`[41], `proc`[42], `puts`[43], `raise`, `remove_instance_variable`, `warn`

B.4.4 ObjectSpace

Class methods: `each_object`

Public methods: `each_object`

B.5 Predefined Constants

Constant	Value	Description
`ARGV`	(depends on execution)	program arguments
`FALSE`	`false`	synonym for `false`
`NIL`	`nil`	synonym for `nil`
`PLATFORM`	`"php"`	alias for `RUBY_PLATFORM`
`RELEASE_DATE`	`"2008-08-08"`	alias for `RUBY_RELEASE_DATE`
`RUBY_PATCHLEVEL`	`114`	patchlevel of the Ruby language
`RUBY_PLATFORM`	`"php"`	platform identifier
`RUBY_RELEASE_DATE`	`"2008-08-08"`	release date
`RUBY_VERSION`	`"1.8.6"`	version of the Ruby language
`TRUE`	`true`	synonym for `true`
`VERSION`	`"1.8.6"`	alias for `RUBY_VERSION`

Table B.1: Predefined constants.

[34] The `lambda` method creates a proc, not lambda as in Ruby.
[35] The `p` method always prints to PHP standard output.
[36] The `print` method does not print `$_` when no parameters are given, always prints to PHP standard output.
[37] The `proc` method creates a proc, not lambda as in Ruby.
[38] The `puts` method always prints to PHP standard output.
[39] The `lambda` method creates a proc, not lambda as in Ruby.
[40] The `p` method always prints to PHP standard output.
[41] The `print` method does not print `$_` when no parameters are given, always prints to PHP standard output.
[42] The `proc` method creates a proc, not lambda as in Ruby.
[43] The `puts` method always prints to PHP standard output.

B.6 Predefined Global Variables

Global Variable	Initial Value	Description
`$!`	`nil`	currently raised exception
`$*`	(depends on execution)	program arguments
`$,`	`nil`	output field separator
`$-v`	(depends on execution)	alias for `$VERBOSE`
`$-w`	(depends on execution)	alias for `$VERBOSE`
`$/`	`"\n"`	input record separator
`$0`[44]	(depends on execution)	program name
`$VERBOSE`	(depends on execution)	verbosity level
`$\`	`nil`	output record separator

Table B.2: Predefined global variables.

Also note that special global variables `$1`, `$2`,... that capture regular expression matches are treated as regular global variables.