Rift Compiler

Spencer Tipping

July 7, 2011

Contents

1 Introduction **2**
1.1 Internal differences from Rubinius 2
1.2 Linguistic differences from Ruby 1.9 3

2 Rift-Core **4**
2.1 Design pattern: extensible subclassing 4
2.2 Syntax elements 5

3 Low-Level Implementation **7**

4 x86-64 Assembler **8**

Chapter 1

Introduction

Rift is structured as a self-hosting JIT compiler. The advantage of this approach is that it's fairly straightforward to bootstrap into a native tracing JIT. The main disadvantage is a highly convoluted build process that involves multiple compilations. Here's how the source code is structured:

`asm/` An ELF generator and x86-64 assembler implemented in pure Ruby

`core/` A compiler from Rift-core (a subset of Ruby) to x86-64 assembly implemented in Rift-core

`stdlib/` An alternative standard library implemented in Rift-core

`jit/` An optimizing trace compiler written using the alternative standard library

The build process consists of these two steps:

1. YARV or MRI is used to run the Rift compiler source on itself to produce `bin/rift`
2. `bin/rift` is used on its own source to produce an optimized `bin/rift`

1.1 Internal differences from Rubinius

Rubinius implements a bootstrapping JIT compiler for Ruby that is designed with many of the same considerations. The main deviation is in the semantic model for Ruby. Rubinius aims for 100% compatibility with the existing MRI toolchain, including C API source compatibility. Possibly as a result of this decision, certain features such as `call/cc` are unimplemented, yielding an arguably more complex interpreter.

Rift's implementation is centered around continuation-passing style, even to the point of defining an alternative assembly-level calling convention to support it. This results in very fast processing of nonlocal exits, `call/cc`, exceptions, etc. It may never support extensions written in C, though direct access to the assembly-level compiler should provide a viable alternative.

1.2 Linguistic differences from Ruby 1.9

Rift is centered around the Rift-core language mentioned earlier. Rift-core is a generalization of Ruby that can be customized to implement a superset of Ruby 1.9. However, it can also implement things outside of the official Ruby standard. In particular, it supports generalizations such as user-defined operators, first-class prefix and postfix modifiers (e.g. `rescue`, `until`), first-class declarative syntax (e.g. `class`, `def`), and numerous compiler-related extensions to the standard library.

Chapter 2

Rift-Core

Rift-core is a generalized subset of Ruby that forms the basis for the Rift compiler. As described earlier, it is designed to be specialized to conform to the existing Ruby implementation, though it can also be customized in other ways. This chapter describes its syntax and semantics.

Listing 2.1 `core/init.rb`

```
const_set :Rift, Module.new

require 'core/lib/init'
require 'core/tree/init'
```

2.1 Design pattern: extensible subclassing

Rather than have subclasses be "parallel" to their parents, the subclasses in Rift-core are generally stored as constants. So, for instance, if `A < B`, then `A`'s constant would be stored as `B::A`. `Rift::Extensible` provides a method to do this:

Listing 2.2 `core/lib/init.rb`

```
require 'core/lib/extensible'
```

Listing 2.3 `core/lib/extensible.rb`

```
Rift.const_set :Extensible, Module.new do
  define_method :extensible_subclass do |name, &initializer|
    child = Class.new self, &initializer
    const_set name, child
    self
  end
end
```

2.2 Syntax elements

The unspecialized Rift-core parser knows how to deal with the following syntax elements:

- Line comments
- Strings without `#{...}` escapes
- Unquoted symbols
- 64-bit signed integers
- Double-precision floating point numbers
- `true`, `false`, `nil`, and `self`
- Class variables (e.g. `@foo`)
- Global variables (e.g. `$bar`)
- Constants (e.g. `Foo`)
- Arrays of expressions
- Ruby 1.8-style hashes (e.g. {`:foo => :bar`})
- Ruby 1.9-style hashes (e.g. {`foo: :bar`})
- `do ... end` blocks
- Heredocs (this kind of hackery can't be generalized away)
- Braced blocks
- Destructuring block binds
- Last-parameter variadic binds
- Ruby infix and prefix operators (e.g. +, <<, =, `...`, `::`, etc.)

Notably absent from the general syntax are:

- `class` and `def`: These are shorthands for `Class.new` and `define_method`, respectively.
- `if`, `for`, `while`, etc: These are shorthands for method calls on `TrueClass`, `FalseClass`, and `Enumerable`.
- Regular expressions.
- Various quoted forms such as `%x/stuff/`.
- `=begin` and `=end`.

- Short-circuit logical operators. These are defined later because they impact the evaluation semantics.

The parser returns an instance of `Rift::Tree` that represents the original source code losslessly. Any given parser can be specialized to return new user-defined subclasses of `Rift::Tree`, each with their own compilation semantics.

Listing 2.4 `core/tree/init.rb`

```ruby
Rift.const_set :Tree, Class.new Struct.new(:operation, :children, :source) do
  include Enumerable
  extend  Extensible

  define_method :each do |&block|
    block.call self
    @children.each &block
    self
  end
end
```

Chapter 3

Low-Level Implementation

Most of Ruby is simple; the complexity arises from the presence of continuations, which are so useful that they should be both available and as performant as is reasonable. However, as many compiler writers know, implementing reentrant continuations efficiently in a lexically-scoped language is something of a challenge. Some of the challenge is alleviated by using CPS conversion, but this creates many closure variables and long-scope accesses.

Chapter 4

x86-64 Assembler

Rift-core will end up being implemented in x86-64 assembly, though thankfully this process is automated. This set of assembler classes provides the facility to generate and link low-level code using the x86-64 instruction set. This is then either wrapped in an ELF container (useful for generating self-contained executables) or referenced as the implementation of a `Method`. For simplicity's sake I've separated the ELF wrapper from the assembler modules, even though they share a source directory.