Rift Compiler

Spencer Tipping

July 7, 2011

Contents

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`

```
1  Object.const_set :Rift, Module.new
2  require 'core/tree'
```

2.1 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

- Braced blocks

- 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 2.2 core/tree.rb

```
1  Rift.const_set :Tree, Class.new do
2    include Enumerable
3    attr :data, :children, :source
4
5    define_method :initialize do |data, *children, source|
6      @data, @children, @source = data, children, source
7    end
8
9    define_method :each do |&block|
10     @children.each block
11     yield self
12   end
13
14   define_method "[]".to_sym do |*args|
15     @children[*args]
16   end
17 end
```

5