Concrete Partial Evaluation in Ruby

Andrew Keep Arun Chauhan
Dept. of Computer Science
Indiana University
{akeep,achauhan}@cs.indiana.edu

1. Introduction

Modern scientific research is a collaborative process, with researchers from many disciplines and institutions working toward a common goal. Dynamic languages, like Ruby, provide a platform for quickly developing simulation and analysis tools, freeing researchers to focus on research instead of spending time developing infrastructure. Ruby is a particularly good fit, allowing incorporation of existing C libraries, simplifying Domain Specific Language creation, and providing both REST and SOAP web-based API libraries. Ruby also provides RPC-style distributed programming. Concrete partial evaluation of Ruby begins to address Ruby's biggest flaw, performance.

The scientific community has already begun to recognize the potential of Ruby. An MPI extension to the language [3, 1] allows quick prototyping of MPI programs. More recently libraries [6, 5] supporting MapReduce [2] have appeared. Web frameworks, such as the popular Ruby on Rails framework, provide tools for producing and consuming REST APIs.

2. Difficult Static Analysis

Listing 1. Factorial method, multiple meanings

```
class Fixnum
  def fact
    (self == 0) ? 1 : self * (self - 1).fact
  end
end

5.fact => 120 # yields expected result

class Fixnum
  def * other
    self + other
  end
end

5.fact => 16 # * redefined as +, new result
```

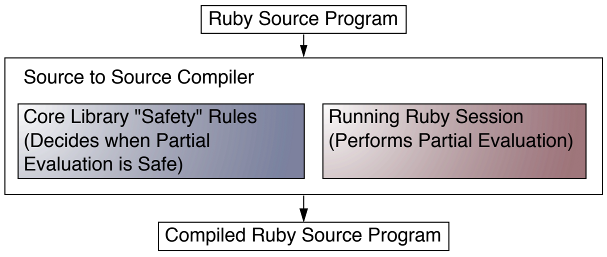
Ruby Source Program
Source to Source Compiler
Core Library "Safety" Rules (Decides when Partial Evaluation is Safe)
Running Ruby Session (Performs Partial Evaluation)
Compiled Ruby Source Program

Figure 1. The partial evaluator

Created by Yukihiro Matsumoto, combining features of his favorite programming languages [4], Ruby's dynamic-typing, open classes, and other meta-programming abilities provide power and flexibility to programmers, but thwart traditional compiler optimizations. As Listing 1 illustrates, the same expression `5.fact` can yield different results, depending on context. While this example is contrived, it demonstrates that a Ruby compiler must be aware of the context, before applying any kind of partial evaluation.

A Ruby compiler needs full program source to ensure context is known. Analysis then proceeds from the top down. Despite this limitation, using the interpreter, as shown in Figure 1, to perform partial evaluation allows it to handle more complex expressions than traditional compilers.

3. Partial Evaluation and Safety

Even using the interpreter, we cannot partially evaluate every expression, since some depend on data or user-input unavailable until runtime. The compiler must determine what to partially evaluate. The crux of this decision rests on the definition of what is "safe" to evaluate. Traditionally, safe operations do not modify the evaluation context or dependencies. This is too strict in Ruby, as many meta-programming techniques rely on modifying the evaluation context. Fortunately, Ruby exposes these events and the resulting changes. A

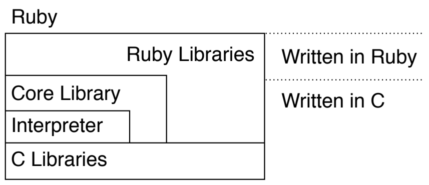

Figure 2. Ruby: Matz Ruby Interpreter 1.8.x

method is safe if the receiver and arguments are known, the block (if provided) is safe, no shared state is modified, and the methods it calls are safe. A block is safe if all its expressions are safe.

This recursive idea of safety leads to a stumbling block—how to decide which Ruby library methods are safe. No small task, given that the interpreter loads 16 Modules and 147 Classes defining 8236 methods. Even accounting for duplicate methods, the task is still daunting. Another wrinkle, illustrated in Figure 2, is that the Ruby library is written in C, for performance.

4. Analyzing the Core Library

As a young language Ruby is still changing, and the interpreter and core library span around 2600 C functions, making a tool highly desirable to reduce analysis effort and insulate against changes. First, "safety" needs to be carefully defined for C. Those C functions that do not access global variables or call unsafe functions are considered safe. Figure 3 illustrates the analysis process. The first pass constructs a function list, categorizing those that access global variables (unsafe), those that do not access global variables or call other functions (safe), and the rest (unresolved), recording their called functions. Roughly 200 external functions, largley from the C standard library, must be categorized manually. The external and defined functions are combined and a fixed-point analysis coalesces them into either safe or unsafe categories.

Currently, analysis finds less then 300 funcions safe and almost 2500 unsafe, including externals. Global variable accesses, such as those that record scope, the symbol table, and other interpreter flags, account for the large volume of unsafe functions. In many cases global variables, such as those Ruby uses to represent modules and classes are set once and never changed. Semantic analysis of these variables may allow us to categorize more functions as safe. This analysis reveals why it is so difficult to optimize Ruby. We are in the process of extending the analysis to include semantic information for some frequently used global variables to be able to partially evaluate more aggressively.

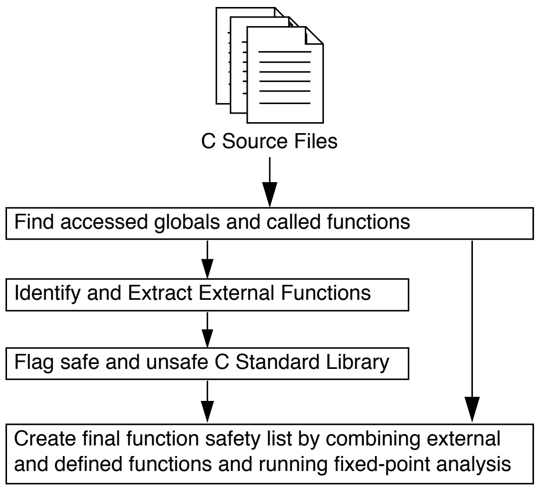
C Source Files
Find accessed globals and called functions
Identify and Extract External Functions
Flag safe and unsafe C Standard Library
Create final function safety list by combining external and defined functions and running fixed-point analysis

Figure 3. C analyzer process

5. Further Work

Once the C analysis is refined, implementing a method to decide what to partially evaluate can be added to existing infrastructure, to complete the partial evaluator. We are optimistic the partial evaluator will improve performance of many Ruby programs. Lessons learned while implementing it will also enable other optimizations such as method inlining. The C code analysis may also prove useful in bringing true thread parallelism to the C implementations of Ruby, something not currently available in Ruby 1.8.x or Ruby 1.9.

References

[1] C. C. Aycock. MPI Ruby with Remote Memory Access. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 280–281, 2004.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 137–150, 2004.

[3] E. Ong. MPI Ruby: Scripting in a Parallel Environment. *Computing in Science and Engineering*, 4(4):78–82, 2002.

[4] About Ruby. On the web. `http://www.ruby-lang.org/en/about/`.

[5] Skynet: A Ruby MapReduce Framework. On the web. `http://skynet.rubyforge.org/`.

[6] Starfish - Ridiculously Easy Distributed Programming with Ruby. On the web. `http://rufy.com/starfish/doc/`.