Concrete Partial Evaluation in Ruby

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

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 developing infrastructure. Ruby fits this role well, allowing:

- Incorporation of existing C libraries
- Simple Domain Specific Language creation
- Support for REST and SOAP web APIs
- Distributed programming, through a simplified Java RMI style library

Why Ruby?

Ruby is already finding a place in the scientific community and industry:

- MPI extension [3, 1] for quick prototyping
- New libraries [6, 5] support MapReduce [2]
- The popular Ruby on Rails framework supports RESTful development

Ruby was originally developed by Yukihiro Matsumoto (Matz), based on his favorite languages [4]. The Ruby language:

- Is purely object-oriented and dynamically-typed
- Provides open classes, allowing meta-programming
- Supports closures and functional programming
- Provides simple iterators through Ruby blocks

The Challenge

The power of Ruby comes at a price. The current C-based interpreter, the Matz Ruby Interpreter (MRI), suffers poor performance since the abstract syntax tree (AST) is used directly by the interpreter. The flexibility of Ruby complicates compilation because context affects the meaning of expressions, as illustrated below.

```
class Fixnum
  def fact
    if self == 0
      1
    else
      self * (self - 1).fact
    end
  end
end

5.fact => 120 # expected result
```

```
class Fixnum
  def * other
    self + other
  end
end

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

Ruby MRI presents another challenge by implementing the core Ruby library in C, for performance reasons. The C implementation buries much of the semantic information about the core library (see figure 1).

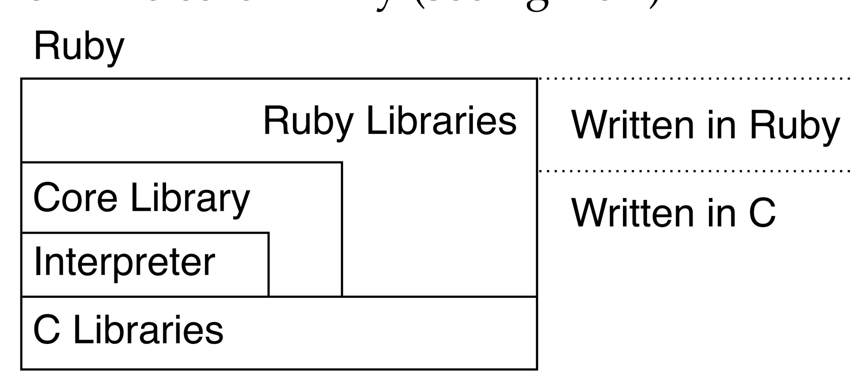

Figure 1: Ruby: Matz Ruby Interpreter 1.8.x

Our Approach

Despite these challenges, Ruby MRI provides a full evaluation environment and access to the AST, allowing a partial evaluator to evaluate source at compile time, as the following figure illustrates.

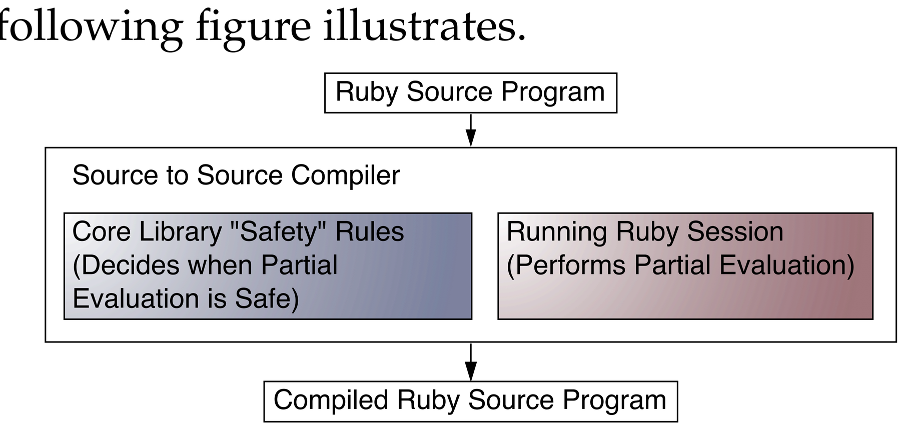

Figure 2: The partial evaluator

In order to use this environment, the core library must be analyzed to determine the safety of each method.

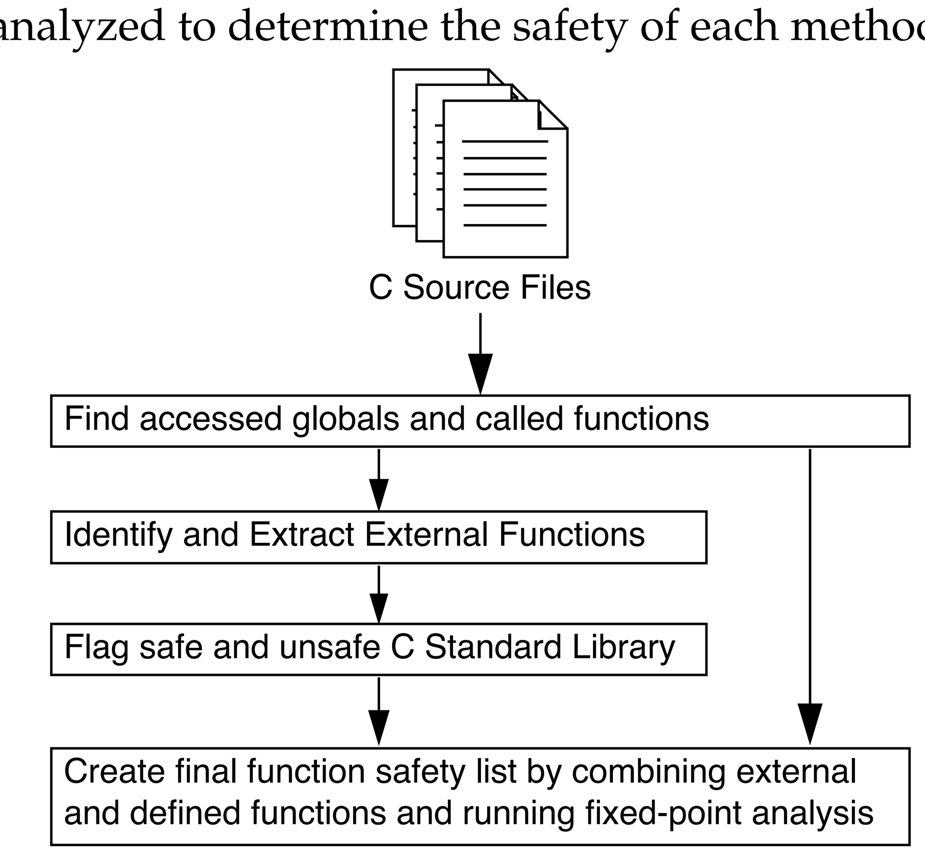

Figure 3: C analyzer process

Safe methods are those that do not perform I/O and do not rely on global variables. A tool to perform the analysis of the C code was developed, both to speed the analysis process, and insulate against changes in Ruby. Figure 3 illustrates how C source files are analyzed.

The Algorithm

Our algorithm first analyzes, then uses the results to partially evaluate Ruby source, producing a new Ruby file. Currently, the partial evaluator targets whole programs due to Ruby's dynamic nature, though we are confident it can be extended to handle libraries. Analysis proceeds top-down from the beginning of a program, following a traditional data-flow approach.

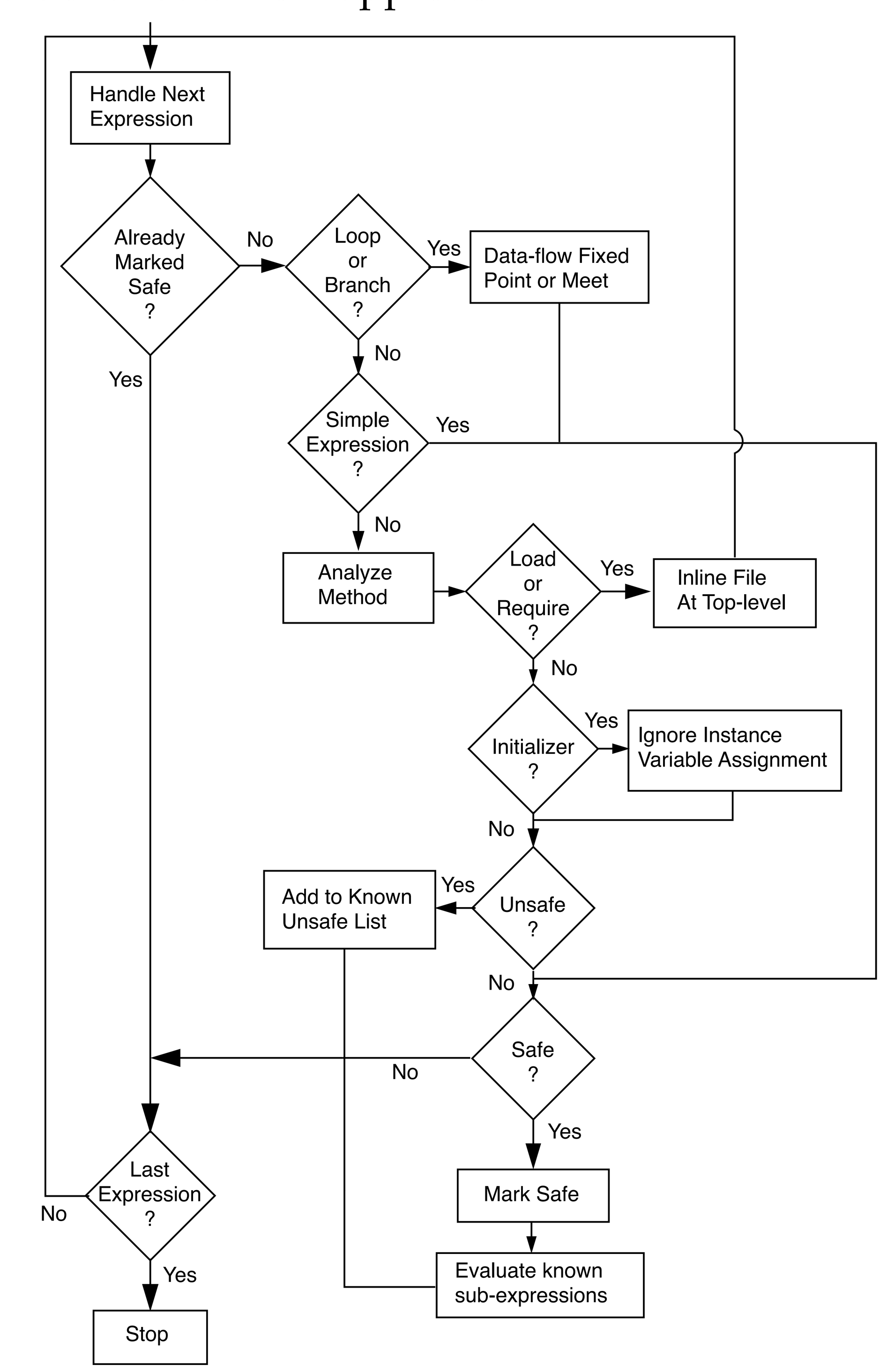

Figure 4: Ruby analyzer process

Method Analysis

Method call analysis is the crux of the algorithm. This analysis provides:

- A list of called methods with their contexts for further analysis
- A list of accessed global, class, and instance variables
- A list of parameters and variables captured in any provided block, which have been modified
- A list of files loaded or required by the method

For methods implemented in C, we rely on the previously performed C analysis step.

Method Call Safety

Safe method calls do not access shared state, have safe blocks, and only call safe methods, though we relax the shared state requirement by taking an "all-or-nothing" approach, marking methods safe, only if all methods accessing a shared variable are safe. Analysis runs iteratively, until a fixed point is reached, ensuring information about shared state is as complete as possible.

Special Cases

Object Initialization The special method `new` first allocates space, then calls the `initialize` method for newly created objects. The partial evaluator must ensure `initialize` calls only safe methods.

Loading Modules Ruby loads supporting modules through the `load` and `require` methods, loading files into the top-level environment, unless a module is specified. The partial evaluator must mimic this behavior, analyzing loaded source code as if it were part of the original program.

Future Work

Once the partial evaluator is complete for whole programs, we hope to use the insights gained during development to:

- Partially evaluate modules, inserting runtime checks to ensure correctness
- Analyze C extensions to Ruby, exposing their semantic properties to the partial evaluator
- Perform further specialization, such as method-inlining

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/`.