The Ruby Intermediate Language *

Michael Furr Jong-hoon (David) An Jeffrey S. Foster Michael Hicks
University of Maryland, College Park
{furr,davidan,jfoster,mwh}@cs.umd.edu

Abstract

Ruby is a popular, dynamic scripting language that aims to "feel natural to programmers" and give users the "freedom to choose" among many different ways of doing the same thing. While this arguably makes programming in Ruby easier, it makes it hard to build analysis and transformation tools that operate on Ruby source code. In this paper, we present the Ruby Intermediate Language (RIL), a Ruby front-end and intermediate representation that addresses these challenges. RIL includes an extensible GLR parser for Ruby, and an automatic translation into an easy-to-analyze intermediate form. This translation eliminates redundant language constructs, unravels the often subtle ordering among side effecting operations, and makes implicit interpreter operations explicit. We also describe several additional useful features of RIL, such as a dynamic instrumentation library for profiling source code and a dataflow analysis engine. We demonstrate the usefulness of RIL by presenting a static and dynamic analysis to eliminate null pointer errors in Ruby programs. We hope that RIL's features will enable others to more easily build analysis tools for Ruby, and that our design will inspire the creation of similar frameworks for other dynamic languages.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Languages

Keywords Ruby, Intermediate language, RIL, profile-guided analysis

1. Introduction

Ruby is a popular, object-oriented, dynamic scripting language inspired by Perl, Python, Smalltalk, and LISP. Over the last several years, we have been developing tools that involve static analysis and transformation of Ruby code. The most notable example is Diamondback Ruby (DRuby), a system that brings static types and static type inference to Ruby [5, 4].

As we embarked on this project, we quickly discovered that working with Ruby code was going to be quite challenging. Ruby aims to "feel natural to programmers" [19] by providing a rich syntax that is almost ambiguous, and a semantics that includes a significant amount of special case, implicit behavior. While the resulting language is arguably easy to use, its complex syntax and semantics make it hard to write tools that work with Ruby source code.

In this paper, we describe the Ruby Intermediate Language (RIL), an intermediate language designed to make it easy to extend, analyze, and transform Ruby source code. As far as we are aware, RIL is the only Ruby front-end designed with these goals in mind. RIL provides four main advantages for working with Ruby code. First, RIL's parser is completely separated from the Ruby interpreter, and is defined using a Generalized LR (GLR) grammar, which makes it much easier to modify and extend. In particular, it was rather straightforward to extend our parser grammar to include type annotations, a key part of DRuby. (Section 2.) Second, RIL translates many redundant syntactic forms into one common representation, reducing the burden on the analysis writer. For example, Ruby includes four different variants of `if-then-else` (standard, postfix, and standard and postfix variants with `unless`), and all four are represented in the same way in RIL. Third, RIL makes Ruby's (sometimes quite subtle) order of evaluation explicit by assigning intermediate results to temporary variables, making flow-sensitive analyses like dataflow analysis simpler to write. Finally, RIL makes explicit much of Ruby's implicit semantics, again reducing the burden on the analysis designer. For example, RIL replaces empty Ruby method bodies by `return nil` to clearly indicate their behavior. (Section 3.)

In addition to the RIL data structure itself, our RIL implementation has a number of features that make working with RIL easier. RIL includes an implementation of the visitor pattern to simplify code traversals. The RIL pretty printer can output RIL as executable Ruby code, so that transformed RIL code can be directly run. To make it easy to build RIL data structures (a common requirement of transformations, which often inject bits of code into a program), RIL includes a partial reparsing module [2]. RIL also has a dataflow analysis engine, and extensive support for run-time profiling. We have found that profiling dynamic feature use and reflecting the results back into the source code is a good way to perform static analysis in the presence of highly dynamic features, such as `eval` [4]. (Section 4.)

RIL is written in OCaml [10], which we found to be a good choice due to its support for multi-paradigm programming. For example, its data type language and pattern matching features provide strong support for manipulating the RIL data structure, the late binding of its object system makes visitors easier to re-use, and RIL's dataflow and profiling libraries can be instantiated with new analyses using functors.

Along with DRuby [5, 4], we have used RIL to build DRails, a tool that brings static typing to Ruby on Rails applications [6]. In addition, several students in a graduate class at the University of Maryland used RIL for a course project. The students were able to build a working Ruby static analysis tool within a few weeks. These

* This research was supported in part by DARPA ODOD.HR00110810073

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DLS'09, October 26, 2009, Orlando, Florida, USA.
Copyright © 2009 ACM 978-1-60558-769-1/09/10. . . $10.00

experiences lead us to believe that RIL is a useful and effective tool for analysis and transformation of Ruby source code. We hope that others will find RIL as useful as we have, and that our discussion of RIL's design will be valuable to those working with other dynamic languages with similar features. RIL is available as part of the DRuby distribution at `http://www.cs.umd.edu/projects/PL/druby`.

2. Parsing Ruby

The major features of Ruby are fairly typical of dynamic scripting languages. Among other features, Ruby includes object-orientation (every value in Ruby is an object, including integers); exceptions; extensive support for strings, regular expressions, arrays, and hash tables; and higher-order programming (code blocks). We assume the reader is familiar with, or at least can guess at, the basics of Ruby. An introduction to the language is available elsewhere [20, 3].

The first step in analyzing Ruby is parsing Ruby source. One option would be to use the parser built in to the Ruby interpreter. Unfortunately, that parser is tightly integrated with the rest of the interpreter, and uses very complex parser actions to handle the ambiguity of Ruby's syntax. We felt these issues would make it difficult to extend Ruby's parser for our own purposes, e.g., to add a type annotation language for DRuby.

Thus, we opted to write a Ruby parser from scratch. The fundamental challenge in parsing Ruby stems from Ruby's goal of giving users the "freedom to choose" among many different ways of doing the same thing [21]. This philosophy extends to the surface syntax, making Ruby's grammar highly ambiguous from an LL/LR parsing standpoint. In fact, we are aware of no clean specification of Ruby's grammar.[1] Thus, our goal was to keep the grammar specification as understandable (and therefore as extensible) as possible while still correctly parsing all the potentially ambiguous cases. Meeting this goal turned out to be far harder than we originally anticipated, but we were ultimately able to develop a robust parser.

2.1 Language Ambiguities

We illustrate the challenges in parsing Ruby with three examples. First, consider an assignment $x = y$. This looks innocuous enough, but it requires some care in the parser: If y is a local variable, then this statement copies the value of y to x. But if y is a *method* (lower case identifiers are used for both method names and local variables), this statement is equivalent to $x = y()$, i.e., the right-hand side is a method call. Thus we can see that the meaning of an identifier is context-dependent.

Such context-dependence can manifest in even more surprising ways. Consider the following code:

```
1  def x()   return 4   end
2  def y()
3    if  false  then x = 1 end
4    x + 2       # error, x is  nil , not a method call
5  end
```

Even though the assignment on line 3 will never be executed, its existence causes Ruby's parser to treat x as a local variable from there on. At run-time, the interpreter will initialize x to nil after line 3, and thus executing $x + 2$ on line 4 is an error. In contrast, if line 3 were removed, $x + 2$ would be interpreted as $x() + 2$, evaluating successfully to 6. (Programmers might think that local variables in Ruby must be initialized explicitly, but this example shows that the parsing context can actually lead to implicit initialization.)

As a second parsing challenge, consider the code:

```
6  f() do |x| x + 1 end
```

Here we invoke the method f, passing a *code block* (higher-order method) as an argument. In this case the code block, delimited by do ... end, takes parameter x and returns $x + 1$.

It turns out that code blocks can be used by several different constructs, and thus their use can introduce potential ambiguity. For example, the statement:

```
7  for x in 1..5 do puts x end
```

prints the values 1 through 5. Notice that the body of for is also a code block (whose parameters are defined after the for token)—and hence if we see a call:

```
8  for x in f() do ... end ...
```

then we need to know whether the code block is being passed to f() or is used as the body of the for. (In this case, the code block is associated with the for.)

Finally, a third challenge in parsing Ruby is that method calls may omit parentheses around their arguments in some cases. For example, the following two lines are equivalent:

```
9  f(2*3, 4)
10 f 2*3, 4
```

However, parentheses may also be used to group sub-expressions. Thus, a third way to write the above method call is:

```
11 f (2*3),4
```

Of course, such ambiguities are a common part of many languages, but Ruby has many cases like this, and thus using standard techniques like refactoring the grammar or using operator precedence parsing would be quite challenging to maintain.

2.2 A GLR Ruby Parser

To meet these challenges and keep our grammar as clean as possible, we built our parser using the dypgen generalized LR (GLR) parser generator, which supports ambiguous grammars [14]. Our parser uses general BNF-style productions to describe the Ruby grammar, and without further change would produce several parse trees for conflicting cases like those described above. To indicate which tree to prefer, we use helper functions to prune invalid parse trees, and we use *merge* functions to combine multiple parse trees into a single, final output.

An excerpt from our parser is given in Figure 1. Line 9 delimits the OCaml functions defined in the preamble (lines 1–8), versus the parser productions (lines 10–27). A dypgen production consists of a list of terminal or non-terminal symbols followed by a semantic action inside of {}'s. The value of a symbol may be bound using []'s for later reference. For example, the non-terminal stmt_list[ss] reduces to a list of statements that is bound to the identifier ss in the body of the action.

The production primary, defined on line 10, handles expressions that may appear nested within other expressions, like a subexpression block (line 11), a method call (line 13) or a for loop (line 14). On line 16, the action for this rule calls the helper function well_formed_do to prune ill-formed sub-trees. The well_formed_do function is defined in the preamble of the parser file, and is shown on lines 1–4. This function checks whether an expression ends with a method call that includes a code block and, if so, it raises the Dyp.Giveup exception to tell dypgen to abandon this parse tree. This

[1] There is a pseudo-BNF formulation of the Ruby grammar in the on-line Ruby 1.4.6 language manual, but it is ambiguous and ignores the many exceptional cases [11].

```
1   let well_formed_do guard body = match ends_with guard with
2     | E_MethodCall(_,_,Some (E_CodeBlock(false,_,_,_)),_) →
3         raise Dyp.Giveup
4     | _ →()
5
6   let well_formed_command m args = match args with
7     | [E_Block _] → raise Dyp.Giveup
8     | ...
9   %%
10  primary:
11    | T_LPAREN[pos] stmt_list[ss] T_RPAREN
12      { E_Block(ss,pos) }
13    | func[f] { f }
14    | K_FOR[pos] formal_arg_list [vars] K_IN arg[guard]
15      do_sep stmt_list [body] K_IEND
16      { well_formed_do guard body; E_For(vars,range,body,pos) }
17
18  command:
19    | command_name[m] call_args[args]
20      { well_formed_command m args;
21        methodcall m args None (pos_of m)}
22    | ...
23
24  func:
25    | command_name[m] T_LPAREN call_args[args] T_RPAREN
26      { methodcall m args None (pos_of m) }
27    | ...
```

Figure 1. Example GLR Code

rule has the effect of disambiguating the for...do..end example by only allowing the correct parse tree to be valid. Crucially, this rule does not require modifying the grammar for method calls, keeping that part of the grammar straightforward.

We use a similar technique to disambiguate parentheses for method calls. The command (line 18) and func (line 24) productions define the rules for method calls with and without parentheses respectively. Each includes a list of arguments using the call_args production (not shown), which may reduce to the primary production (line 10). As with the action of the for production, the command production calls the helper function well_formed_command to prune certain parse trees. This function (line 6) aborts the parse if the method call has a single argument that is a grouped sub-expression (stored in the E_Block constructor).

Figure 2 shows how our grammar would attempt to parse four variations of a Ruby method call. The first column shows the source syntax, and the last two columns show the result of using either func or command to parse the code. As the GLR parsing algorithm explores all possible parse trees, dypgen will attempt to use both productions to parse each method call. The first variation, $\mathsf{f}(\mathsf{x},\mathsf{y})$, fails to parse using the command production since the sub-expression production on line 11 must apply, but the pair x,y is not a valid (isolated) Ruby expression. Similarly, the second example does not parse using func since it would reduce $\mathsf{f}(\mathsf{x})$ to a method call, and be left with the pair $(\mathsf{f}(\mathsf{x})),\mathsf{y}$ which is invalid for the same reason. The third example, $\mathsf{f}\ \mathsf{x},\mathsf{y}$ would only be accepted by the command production as it contains no parentheses. Thus, our productions will always produce a single parse tree for method calls with at least 2 arguments. However, the last variation $\mathsf{f}(\mathsf{x})$ would be accepted by both productions, producing two slightly different parse trees: $\mathsf{f}(\mathsf{x})$ vs $\mathsf{f}((\mathsf{x}))$. To choose between these, the well_formed_command function rejects a function with a single E_Block argument (line 7), and thus only the func production will succeed.

By cleanly separating out the disambiguation rules in this way, the core productions are relatively easy to understand, and the parser is easier to maintain and extend. For example, as we discovered more special parsing cases baked into the Ruby interpreter, we needed to modify only the disambiguation rules and could leave the productions alone. Similarly, adding type annotations to individual Ruby expressions required us to only change a single production and for us to add one OCaml function to the preamble. We believe that our GLR specification comes fairly close to serving as a standalone Ruby grammar: the production rules are quite similar to the pseudo-BNF used now [11], while the disambiguation rules describe the exceptional cases. Our parser currently consists of 75 productions and 513 lines of OCaml for disambiguation and helper functions.

Ruby source	command	func
$\mathsf{f}\ (\mathsf{x},\mathsf{y})$	comma in sub-expr	success
$\mathsf{f}\ (\mathsf{x}),\mathsf{y}$	success	comma after reduction
$\mathsf{f}\ \mathsf{x},\mathsf{y}$	success	no parens
$\mathsf{f}\ (\mathsf{x})$	raise Giveup	success

Figure 2. Disambiguation example

Since Ruby has no official specification, we used three techniques to establish our parser is compatible with the existing "reference" parser, which is part of the Ruby 1.8 interpreter. First, we verified that our parser can successfully parse a large corpus of over 160k lines of Ruby code without any syntax errors. Second, we developed 458 hand-written unit tests that ensure Ruby syntax is correctly parsed into known ASTs. For example, we have tests to ensure that both $\mathsf{f}\ \mathsf{x},\mathsf{y}$ and $\mathsf{f}\ (\mathsf{x}),\mathsf{y}$ are parsed as two-argument method calls. Finally, we parse and then unparse to disk the test suite shipped with the Ruby interpreter. This unparsed code is then executed to ensure the test suite still passes. This last technique is also used to verify that our pretty printing module and the semantic transformations described in Section 3 are correct (these modules each have their own unit test suites as well).

3. Ruby Intermediate Language

Parsing Ruby source produces an abstract syntax tree, which we could then try to analyze and transform directly. However, like most other languages, Ruby AST's are large, complex, and difficult to work with. Thus, we developed the Ruby Intermediate Language (RIL), which aims to be low-level enough to be simple, while being high-level enough to support a clear mapping between RIL and the original Ruby source. This last feature is important for tools that report error messages (e.g., the type errors produced by DRuby), and to make it easy to generate working Ruby code directly from RIL.

RIL provides three main advantages: First, it uses a common representation of multiple, redundant source constructs, reducing the number of language constructs that an analysis writer must handle. Second, it makes the control-flow of a Ruby program more apparent, so that flow-sensitive analyses are much easier to write. Third, it inserts explicit code to represent implicit semantics, making the semantics of RIL much simpler than the semantics of Ruby.

We discuss each of these features in turn.

3.1 Eliminating Redundant Constructs

Ruby contains many equivalent constructs to allow the programmer to write the most "natural" program possible. We designed RIL to include only a small set of disjoint primitives, so that analyses need to handle fewer cases. Thus, RIL translates several different Ruby source constructs into the same canonical representation.

As an example of this translation, consider the following Ruby statements:

(1) **if** p **then** e **end**
(2) **unless** (**not** p) **then** e **end**
(3) e **if** p
(4) e **unless** (**not** p)

```
result =
  begin
    if p then a() end
  rescue Exception => x
    b()
  ensure
    c()
  end
```

(a) Ruby code

```
begin
  if p then
    t1 = a()
  else
    t1 = nil
  end
rescue Exception => x
  t1 = b()
ensure
  c()
end
result = t1
```

(b) RIL Translation

Figure 3. Nested Assignment

All of these statements are equivalent, and RIL translates them all into form (1).

As another example, there are many different ways to write string literals, and the most appropriate choice depends on the contents of the string. For instance, below lines 1, 2, 3, and 4–6 all assign the string Here's Johnny to s, while RIL represents all four cases internally using the third form:

```
1 s = "Here's Johnny"
2 s = 'Here\'s Johnny'
3 s = %{Here's Johnny}
4 s = <<EOF
5 Here's Johnny
6 EOF
```

RIL performs several other additional simplifications. Operators are replaced by the method calls they represent, e.g., x + 2 is translated into x.+(2); while and until are coalesced; logical operators such as and and or are expanded into sequences of conditions, similarly to CIL [12]; and negated forms (e.g., !=) are translated into a positive form (e.g., ==) combined with a conditional.

All of these translations serve to make RIL much smaller than Ruby, and therefore there are many fewer cases to handle in a RIL analysis as compared to an analysis that would operate on Ruby ASTs.

3.2 Linearization

In Ruby, almost any construct can be nested inside of any other construct, which makes the sequencing of side effects tricky and tedious to unravel. In contrast, each statement in RIL is designed to perform a single semantic action such as a branch or a method call. As a result, the order of evaluation is completely explicit in RIL, which makes it much easier to build flow-sensitive analyses, such as dataflow analysis.

To illustrate some of the complexities of evaluation order in Ruby, consider the code in Figure 3(a). Here, the result of an exception handling block is stored into the variable result. If an analysis needs to know the value of the right-hand side and only has the AST to work with, it would need to descend into exception block and track the last expression on every branch, including the exception handlers.

Figure 3(b) shows the RIL translation of this fragment, which inlines an assignment to a temporary variable on every viable return path. Notice that the value computed by the ensure clause (this construct is similar to finally in Java) is evaluated for its side effect only, and is not returned. Also notice that the translation has added an explicit nil assignment for the fall-through case for if. (This is an example of implicit behavior, discussed more in Section 3.3.) These sorts of details can be very tricky to get right, and it took a significant effort to find and implement these cases. RIL performs similar translations for ensuring that every path through a method body ends with a return statement and that every path through a block ends with a next statement[2].

Ruby	Method Order	RIL
a(). f = b().g	a,b,g,f=	t1 = a() t3 = b() t2 = t3.g() t1 . f=(t2)
a(). f,x = b().g	b,g,a,f=	t2 = b() t1 = t2.g() (t4, x) = t1 t3 = a() t3 . f=(t4)

Figure 4. RIL Linearization Example

Another problematic case for order-of-evaluation in Ruby arises because of Ruby's many different assignment forms. In Ruby, fields are hidden inside of objects and can only be manipulated through method calls. Thus using a "set method" to update a field is very common, and so Ruby includes special syntax for allowing a set method to appear on the left hand side of an assignment. The syntax a.m = b is equivalent to sending the m= message with argument b to the object a. However, as this syntax allows method calls to appear on both sides of the assignment operator, we must be sure to evaluate the statements in the correct order. Moreover, the evaluation order for these constructs can vary depending on the whether the assignment is a simple assignment or a parallel assignment.

Figure 4 demonstrates this difference. The first column lists two similar Ruby assignment statements whose only difference is that the lower one assigns to a tuple (the right-hand side must return a two-element array, which is then split and assigned to the two parts of the tuple). The second column lists the method call order—notice that a is evaluated at a different time in the two statements. The third column gives the corresponding RIL code, which makes the evaluation order clear. Again, these intricacies were hard to discover, and eliminating them makes RIL much easier to work with.

3.3 Materializing Implicit Constructs

Finally, Ruby's rich syntax tries to minimize the effort required for common operations. As a consequence, many expressions and method calls are inserted "behind the scenes" in the Ruby interpreter. We already saw one example of this above, in which fall-though cases of conditionals return nil. A similar example is empty method bodies, which also evaluate to nil.

There are many other constructs with implicit semantics. For example, it is very common for a method to call the superclass's implementation using the same arguments that were passed to it. In this case, Ruby allows the programmer to omit the arguments altogether and implicitly uses the same values passed to the current method. For example, in the following code:

```
1 class A
2   def foo(x,y) ... end
3 end
4 class B < A
5   def foo(x,y)
6     ...
7     super
8   end
9 end
```

[2] next acts as a local return from inside of a block.

```
1  class Format
2    ['bold','underline'].each do |str|
3      eval "def #{str}() @#{str} end"
4    end
5  end
```

(a) Original, Dynamic Code

```
1  class Format
2    ['bold','underline'].each do |str|
3      case "def #{str}() @#{str} end"
4      when "def bold() @bold end"
5        def bold() @bold end
6      when "def underline() @underline end"
7        def underline() @underline end
8      else safe_eval "def #{str}() @#{str} end"
9      end
10   end
11 end
```

(b) Transformed Code

Figure 5. Profiling Example

the call on line 7 is the same as **super**(x,y), which is what RIL translates the call to. Without this transformation, every analysis would have to keep track of these parameters itself, or worse, mistakenly model the call on line 7 as having no actual arguments.

One construct with subtle implicit semantics is rescue. In Figure 3(b), we saw this construct used with the syntax **rescue** C => x, which binds the exception to x if it is an instance of C (or a subclass of C). However, Ruby also includes a special abbreviated form **rescue** => x, in which the class name is omitted. The subtlety is that, contrary to what might be expected, a clause of this form does not match arbitrary exceptions, but instead only matches instances of StandardError, which is a superclass of many, but not all exceptions. To make this behavior explicit, RIL requires every **rescue** clause to have an explicit class name, and inserts StandardError in this case.

Finally, Ruby is often used to write programs that manipulate strings. As such, it contains many useful constructs for working with strings, including the # operator, which inserts a Ruby expression into the middle of a string. For example, "Hi #{x.name}, how are you?" computes x.name, invokes its to_s method to convert it to a string, and then inserts the result using concatenation. Notice that the original source code does not include the call to to_s. Thus, RIL both replaces uses of # with explicit concatenation and makes the to_s calls explicit. The above code is translated as:

```
1  t1 = x.name
2  t2 = "Hi " + t1.to_s
3  t2 + ", how are you?"
```

Similar to linearization, by making implicit semantics of constructs explicit, RIL enjoys a much simpler semantics than Ruby. In essence, like many other intermediate languages, the translation to RIL encodes a great deal of knowledge about Ruby and thereby lowers the burden on the analysis designer. Instead of having to worry about many complex language constructs, the RIL user has fewer, mostly disjoint cases to be concerned with, making it easier to develop correct Ruby analyses.

4. Profiling Dynamic Constructs

An important feature of Ruby that any static analysis must handle is its use of dynamic features such as eval. When eval(e) is called, the Ruby interpreter evaluates e, which must return a string, and then parses and evaluates that string as ordinary Ruby code. As this string may contain any valid Ruby expression, uses of eval can easily make a static analysis unsound. To precisely analyze code containing eval, we need to know what strings may be passed to eval at run time. We could try to do this with a purely static analysis (approximating what the string arguments could be), but doing so would likely be quite imprecise. Instead, we developed a dynamic analysis library that, among other things, lets us profile dynamic constructs. Our most complex RIL client to date, Diamondback Ruby, uses this infrastructure to replace dynamic constructs with static constructs according to a program's test suite [4].

For example, Figure 5(a) shows a Ruby program that uses eval to define a pair of methods, bold and underline. Without knowing the result of the eval statement, a static analysis would be unable to reason about the presence of these methods (e.g., DRuby would produce type errors at their call sites). However, an important property of this code is that the eval occurs at the top level of the class—and so it will always be executed—and it will always evaluate the same strings. Thus the effects of this call are essentially static.

RIL includes a transformation for eval statements (and other dynamic features) that, when invoked, will produce the code similar to that shown in Figure 5(b). Here, the eval statement has been replaced with a case statement that first checks the string against those observed by the program execution, and includes the corresponding program source directly in the method body. The result of this transformation is a modified program that makes the effects of the eval explicit in the source. Thus, any static analysis that walks the source tree could benefit from this transformation. We found that many dynamic features in Ruby are used in essentially static ways, making this technique effective in practice [4].

The architecture of RIL's dynamic analysis library is shown in Figure 6 and consists of five main stages. We assume that we have a set of test cases under which we run the program to gather profiles. First, we execute the target program (using the test cases), but with a special Ruby file preloaded that redefines require, the method that loads another file. Our new version of require behaves as usual, except it also records all of the files that are loaded during the program's execution. This is because require has dynamic behavior, like eval: Ruby programs may dynamically construct file names to pass to require (and related constructs) or even redefine the semantics of the require method.

After we have discovered the set of application files, in stage two we instrument each file to record profiling information. This transformation modifies the Ruby code to track strings passed to eval and several other dynamic constructs. We then unparse the modified source files to disk using RIL's pretty printer and execute the resulting program. Here we must be very careful to preserve the execution environment of the process, e.g., the current working directory, the name of the executed script (stored in the Ruby global $0), and the name of the file (stored in __FILE__ in Ruby). When the execution is complete, we serialize all of the profiled data to disk using YAML, a simple markup language supported by both Ruby and OCaml [22]. Finally, we read in the gathered profiles and use them to transform the original source code prior to applying our main analysis.

To use our dynamic analysis library, a client must provide the instrumentation used in stage two, and the transformation in stage four. We discuss each of these by example in Section 5.3.

5. Using RIL

In this section, we demonstrate RIL by developing a simple dynamic null pointer analysis that uses a source to source transformation to prevent methods from being called on nil. We then construct a dataflow analysis to improve the performance of the transformed

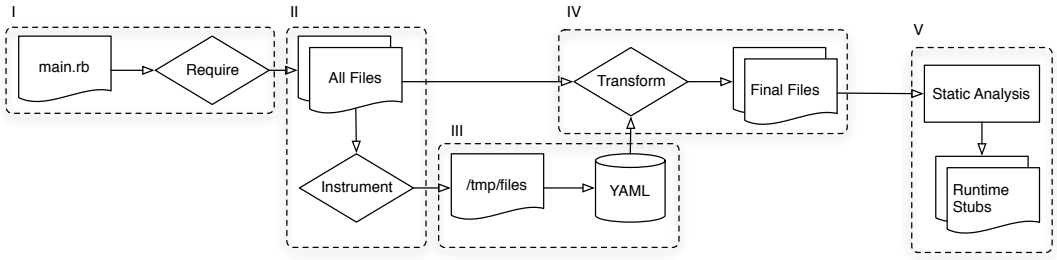

Figure 6. Dynamic Instrumentation Architecture

code. Finally, we use our profiling library to further optimize functions that are verified with a test suite. Along the way, we illustrate some of the additional features our implementation provides to make it easier to work with RIL. In our implementation, RIL is represented as an OCaml data structure, and hence all our examples below are written in OCaml.

5.1 Eliminating Nil Errors

We will develop an example Ruby-to-Ruby transformation written with RIL. Our transformation modifies method calls such that if the receiver object is nil then the call is ignored rather than attempted. In essence this change makes Ruby programs *oblivious* [16] to method invocations on nil, which normally cause exceptions[3]. As an optimization, we will not transform a call if the receiver is self, since self can never be nil. Our example transformation may or may not be useful, but it works well to demonstrate the use of RIL.

RIL includes an implementation of the visitor pattern that is modeled after CIL [12]. A visitor object includes a (possibly inherited) method for each RIL syntactic variant (statement, expression, and so on) using pattern matching to extract salient attributes. The code for our safeNil visitor class is as follows:

```
1  class safeNil = object
2    inherit default_visitor as super
3    method visit_stmt node = match node.snode with
4      | MethodCall(_, {mc_target='ID_Self}) → SkipChildren
5      | MethodCall(_, {mc_target=#expr as targ}) →
6          ChangeTo (transform targ node)
7      | _ → super#visit_stmt node
8  end
```

The safeNil class inherits from default_visitor (line 2), which performs no actions. We then override the inherited visit_stmt method to get the behavior we want: method calls whose target is self are ignored, and we skip visiting the children (line 4). This is sensible because RIL method calls do not have any statements as subexpressions, thanks to the linearization transformation mentioned in Section 3.2. Method calls with non-self receivers are transformed (lines 5–6). Any other statements are handled by the superclass visitor (line 7), which descends into any sub-statements or -expressions. For example, at an if statement, the visitor would traverse the true and false branches.

To implement the transformation on line 6, we need to create RIL code with the following structure, where E is the receiver object and M is the method invocation:

```
1  if E.nil? then nil else M end
```

To build this code, RIL includes a partial reparsing module [2] that lets us mix concrete and abstract syntax. To use it, we simply call RIL's reparse function:

[3] In fact, nil is a valid object in Ruby and does respond to a small number of methods, so some method invocations on nil would be valid.

```
1  let transform targ node =
2    reparse ~env:node.locals
3      "if %a.nil? then nil else %a end"
4      format_expr targ format_stmt node
```

Here the string passed on line 3 describes the concrete syntax, just as above, with %a wherever we need "hole" in the string. We pass targ for the first hole, and node for the second. As is standard for the %a format specifier in OCaml, we also pass functions (in this case, format_expr and format_stmt) to transform the corresponding arguments into strings.

Note that one potential drawback of reparsing is that reparse will complain at run-time if mistakenly given unparsable strings; constructing RIL data structures directly in OCaml would cause mistakes to be flagged at compile-time, but such direct construction is far more tedious. Also, recall from Section 2 that parsing in Ruby is highly context-dependent. Thus, on line 2 we pass node.locals as the optional argument env to ensure that the parser has the proper state to correctly parse this string in isolation.

5.2 Dataflow Analysis

The above transformation is not very efficient because it transforms every method call with a non-self receiver. For example, the transformation would instrument the call to + in the following code, even though we can see that x will always be an integer.

```
1  if p then x = 3 else x = 4 end
2  x + 5
```

To address this problem, we can write a static analysis to track the flow of literals through the current scope (e.g., a method body), and skip instrumenting any method call whose receiver definitely contains a literal.

We can write this analysis using RIL's built-in dataflow analysis engine. To specify a dataflow analysis [1] in RIL, we supply a module that satisfies the following signature:

```
1  module type DataFlowProblem =
2  sig
3    type t                          (* abstract type of facts *)
4    val top : t                     (* initial fact for stmts *)
5    val eq : t → t → bool           (* equality on facts *)
6    val to_string : t → string
7
8    val transfer : t → stmt → t     (* transfer function *)
9    val meet : t list → t           (* meet operation *)
10 end
```

Given such a module, RIL includes basic support for forwards and backwards dataflow analysis; RIL determines that a fixpoint has been reached by comparing old and new dataflow facts with eq. This dataflow analysis engine was extremely easy to construct because each RIL statement has only a single side effect.

For this particular problem, we want to determine which local variables may be nil and which definitely are not. Thus, we begin our dataflow module, which we will call NilAnalysis, by defining the type t of dataflow facts to be a map from local variable names (strings) to facts, which are either MaybeNil or NonNil:

```
1  module NilAnalysis = struct
2    type t = fact StrMap.t
3    and fact = MaybeNil | NonNil (* core dataflow facts *)
4    ...
```

We omit the definitions of top, eq, to_string, and meet, as they are uninteresting. Instead, we will present only the transfer function and a small helper function to demonstrate working with RIL's data structures. The function transfer takes as arguments the input dataflow facts map, a statement stmt, and returns the output dataflow facts:

```
1    let rec transfer map stmt = match stmt.snode with
2      | Assign(lhs, #literal) → update_lhs NonNil map lhs
3      | Assign(lhs, 'ID_Var('Var_Local, rvar)) →
4          update_lhs (StrMap.find rvar map) map lhs
5      | MethodCall(Some lhs,_) | Yield(Some lhs,_)
6      | Assign(lhs, _) → update_lhs MaybeNil map lhs
7      | _ → map
8
9    and update_lhs fact map lhs = match lhs with
10     | 'ID_Var('Var_Local, var) → update var fact map
11     | #identifier → map
12     | 'Tuple lst → List.fold_left (update_lhs MaybeNil) map lst
13     | 'Star (#lhs as l) → update_lhs NonNil map l
14
15   (* val update : string → fact → t → t *)
16 end
```

The first case we handle is assigning a literal (line 2). Since literals are never nil, line 2 uses the helper function update_lhs to mark the left-hand side of the assignment as non-nil[4].

The function update_lhs has several cases, depending on the left-hand side. If it is a local variable, that variable's dataflow fact is updated in the map (line 10). If the left-hand side is any other identifier (such as a global variable), the update is ignored, since our analysis only applies to local variables. If the left-hand side is a tuple (i.e., it is a parallel assignment), then we recursively apply the same helper function but conservatively mark the tuple components as MaybeNil. The reason is that parallel assignment can be used even when a tuple on the left-hand side is larger than the value on the right. For example x,y,z = 1,2 will store 1 in x, 2 in y and nil in z. In contrast, the star operator always returns an array (containing the remaining elements, if any), and hence variables marked with that operator will never be nil (line 13). For example, x,y,*z = 1,2 will set x and y to be 1 and 2, respectively, and will set z to be a 0-length array.

Going back to the main transfer function, lines 3–4 match statements in which the right-hand side is a local variable. We look up that variable in the input map, and update the left-hand side accordingly. Lines 5–6 match other forms that may assign to a local variable, such as method calls. In these cases, we conservatively assume the result may be nil. Finally, line 7 matches any other statement forms that do not involve assignments, and hence do not affect the propagation of dataflow facts.

To use our NilAnalysis module, we instantiate the dataflow analysis engine with NilAnalysis as the argument:

```
1  module NilDataFlow = Dataflow.Forwards(NilAnalysis)
```

[4] Perhaps surprisingly, nil itself is actually an identifier in Ruby rather than a literal, and RIL follows the same convention.

Finally, we add two new cases to our original visitor. First, we handle method definitions (lines 2–5) where we invoke the fixpoint function on the body of the method. fixpoint takes a RIL statement and an initial set of dataflow facts (init_formals sets each formal argument to MaybeNil) and returns two hash tables, which provide the input and output facts at each statement (line 4 ignores the latter). Second, we check if a method target is a local variable and skip the instrumentation if it is NonNil (lines 6–12):

```
1    ... (* safeNil visitor *)
2    | Method(name,args,body) →
3      let init_facts = init_formals args MaybeNil in
4      let in_facts, _ = NilDataFlow.fixpoint body init_facts in
5        (* visit body *)
6    | MethodCall(_,
7        {mc_target=('ID_Var('Var_Local, var) as targ)}) →
8        let map = Hashtbl.find in_facts node in
9          begin match StrMap.find var map with
10           | MaybeNil → ChangeTo(transform targ node)
11           | NonNil → SkipChildren
12         end
```

5.3 Dynamic Analysis

Section 4 discussed the need for handling dynamic features such as eval. Indeed a call to eval could make our nil analysis unsound, since it could overwrite local variables with nil. However, because the eval analysis is orthogonal to DRuby's type system, our example analysis can instantly take advantage of it by operating on the transformed source like that presented in Figure 5(b).

However, our dynamic analysis library could also be used to reduce the overhead of our transformed code. Currently, we use an intraprocedural dataflow analysis, and so method parameters are conservatively assumed to be MaybeNil by our analysis. While public methods may need to handle nil values (possibly by simply throwing an informative exception), private methods might have stronger assumptions, since they can only be called from within the current class. If we could prove a private method is only invoked with non-nil values, our analysis could take this information into account. It may be possible to do this with a sufficiently advanced static analysis, but another option would be to use a dynamic analysis to ensure this property holds.

Our strategy is as follows. First, we will instrument the program to ensure that selected methods are only passed non-nil values. Second, we will execute the test suite, to check that these invariants hold. Finally, we will transform the program using a slightly improved dataflow analysis that takes this runtime data into account.

To simplify the presentation, we only track if all of a method's arguments are non-nil, instead of tracking them individually. In reality, we could treat each parameter separately without much difficulty.

Instrumentation To record data from a profiled execution, RIL provides a small runtime library for collecting and storing application values to disk. Data is passed to this library by inserting explicit calls into the application using RIL's instrumentation pass (stage II in Figure 6). Thus, the first step in building a new dynamic analysis is to register a new collection class with our runtime library:

```
1  class DRuby::Profile
2    class Dynnil < Interceptor
3      def initialize(collector)
4        super(collector, "dynnil")
5      end
6      def Dynnil.extract(recv,mname,*args)
7        args.all? {|x| !x.nil?}
8      end
9    end
10 end
```

Here, we define the class Dynnil, which inherits from the base class Interceptor that is part of RIL. Our class then defines two methods, a constructor on lines 3–5 that registers our module with the data collection object under the name "dynnil," and a class method extract, which collects the data of interest. In our case, it returns true if all of the arguments to a method call are non-nil, and false otherwise. The RIL runtime library will instantiate this class automatically at runtime, providing an appropriate instance of collector to the constructor.

To use this class, we create an OCaml module that uses RIL's visitors to instrument the application's methods, inserting calls to the runtime library.

```
1  module NilProfile : DynamicAnalysis = struct
2    let name = "dynnil"
3
4    let instrument mname args body pos =
5      let file, line = get_pos pos in
6      let code = reparse ~env:body. lexical_locals
7        "DRuby::Profile::Dynnil.watch('%s',%d,self,'%a',[%a])"
8          file line format_def_name mname
9          (format_comma_list format_param) args
10     in
11     let body' = seq [code;body] body.pos in
12       meth mname args body' pos
```

We begin our module, called NilProfile, by declaring the name of our Ruby collection on line 2 (matching the string used in the constructor above). Next, lines 4–12 define the function instrument_method, which inserts a call to the watch method, a method that Dynnil inherits from the Interceptor class. This method calls our overloaded extract method as a subroutine and stores the boolean result along with the given filename, line number, and method name in the YAML store. Like our example in Section 5.1, we construct a call to watch using our reparsing module (lines 6–9), and prepend it to the front of the method body using the seq function (line 11), which creates a sequence of statements. Finally, we construct a new method definition using the helper meth (line 12). The effect of this instrumentation is shown below:

```
1  def add(x,y) # before
2    x + y
3  end
4
5  def add(x,y) # after
6    DRuby::Profile::Dynnil.watch("file.rb",1,self,"add",[x,y])
7    x + y
8  end
```

Lastly, we invoke the instrument_method function by constructing a new visitor object that instruments a method on line 6 if should_instrument (not shown) returns true and otherwise keeps the existing method definition (line 7).

```
1  class instrument_visitor = object(self)
2    inherit default_visitor as super
3    method visit_stmt stmt = match stmt.snode with
4      | Method(mname,args,body) →
5        if should_instrument stmt
6        then ChangeTo (instrument mname args body stmt.pos)
7        else SkipChildren
8      | _ → super#visit_stmt stmt
9  end
```

Transformation After our application has been profiled, we can transform it using our safeNil visitor as defined in Section 5.2. The only modification we must make is that RIL's dynamic analysis library provides us with an additional lookup function, which can be used to query the YAML store. Thus, we will update the Method definition case in our visitor, by first checking the YAML store to see if the arguments are all non-nil, and if so, use stronger dataflow facts. Our NilProfile OCaml module continues:

```
13   module Domain = Yaml.YString
14   module CoDomain = Yaml.YBool
15
16   (* lookup : Domain.t → pos → CoDomain.t list *)
17   let really_nonnil lookup mname pos =
18     let uses = lookup mname pos in
19       if uses = [] then false
20       else not (List.mem false uses)
21
22   ... (* safeNil visitor *)
23   | Method(mname,args,body) →
24     let init_facts =
25       if really_nonnil lookup mname body.pos
26       then init_formals args NonNil
27       else init_formals args MaybeNil
28     in
29     let in_facts, _ = NilDataFlow.fixpoint body init_facts in
30       (* visit body *)
31 end
```

Lines 13–14 define two submodules which define the types we expect in the YAML store, i.e., a mapping from strings (method names) to boolean values (only non-nil arguments). RIL will automatically parse the YAML data and ensure it matches these types, giving us the type-safe lookup function described on line 16. This function takes a method name and a source location and returns a list of boolean values recorded at that position, one for each observation. This function is then used by the really_nonnil function which returns false on line 19 if the list is empty (e.g., the method was never called or not instrumented) or returns false if any of the observations returned false (line 20). Finally, we update the safeNil visitor to set the initial dataflow facts for the method formal arguments to NonNil based on the result of really_nonnil.

6. Related Work

There are several threads of related work.

Another project that provides access to the Ruby AST is ruby_parser [17]. This parser is written in Ruby and stores the AST as an S-expression. ruby_parser performs some syntactic simplifications, such as translating unless statements into if statements, but does no semantic transformations such as linearizing effects or reifying implicit constructs. The authors of ruby_parser have also developed several tools to perform syntax analysis of Ruby programs [18]: *flay*, which detects structural similarities in source code; *heckle*, a mutation-based testing tool; and *flog*, which measures code complexity. We believe these tools could also be written using RIL, although most of RIL's features are tailored toward developing analyses that reason about the semantics of Ruby, not just its syntax.

Several integrated development environments [15, 13] have been developed for Ruby. These IDEs do some source code analysis to provide features such as code refactoring and method name completion. However, they are not specifically designed to allow users to develop their own source code analyses. Integrating analyses developed with RIL into an IDE would be an interesting direction for future work.

The Ruby developers recently released version 1.9 of the Ruby language, which includes a new bytecode-based virtual machine. The bytecode language retains some of Ruby's source level redundancy, including opcodes for both if and unless statements [23]. At the same time, opcodes in this language are lower level than RIL's statements, which may make it difficult to relate instructions back to their original source constructs. Since this bytecode formulation

is quite new, it is not yet clear whether it would make it easier to write analyses and transformations, as was the goal of RIL.

While the Ruby language is defined by its C implementation, several other implementations exist, such as JRuby [8], IronRuby [7], and MacRuby [9]. These projects aim to execute Ruby programs using different runtime environments, taking advantage of technologies present on a specific platform. For example, JRuby allows Ruby programs to execute on the Java Virtual Machine, and allows Ruby to call Java code and vice versa. While these projects necessarily include some analysis of the programs, they are not designed for use as an analysis writing platform.

Finally, RIL's design was influenced by the C Intermediate language [12], a project with similar goals for C. In particular, the authors' prior experience using CIL's visitor class, and CIL's clean separation of side-effect expressions from statements, lead to a similar design in RIL.

7. Conclusion

In this paper, we have presented RIL, the Ruby Intermediate Language. The goal of RIL is to provide a representation of Ruby source code that makes it easy to develop source code analysis and transformation tools. Toward this end, RIL includes a GLR parser designed for modification and extensibility; RIL translates away redundant constructs; RIL makes Ruby's order of side effecting operations clear; RIL makes explicit many implicit operations performed by the Ruby interpreter; and RIL includes dynamic analysis framework for profiling executions of programs. Combined, we believe these features minimize redundant work and reduce the chances of mishandling certain Ruby features, making RIL an effective and useful framework for working with Ruby source code.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1988.

[2] Akim Demaille, Roland Levillain, and Benoît Sigoure. Tweast: a simple and effective technique to implement concrete-syntax ast rewriting using partial parsing. In Sung Y. Shin and Sascha Ossowski, editors, *SAC*, pages 1924–1929. ACM, 2009.

[3] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly Media, Inc, 2008.

[4] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *Proceedings of the twenty fourth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 2009.

[5] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static Type Inference for Ruby. In *OOPS Track, SAC*, 2009.

[6] Jong hoon (David) An, Avik Chaudhuri, and Jeffrey S. Foster. Static Typing for Ruby on Rails. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, Auckland, New Zealand, November 2009. To appear.

[7] IronRuby - Ruby implementation for the .net platform, May 2009. `http://www.ironruby.net/`.

[8] JRuby - Java powered Ruby implementation, February 2008. `http://jruby.codehaus.org/`.

[9] MacRuby - Ruby implementation built on top of the objective-c common runtime and garbage collector, May 2009. `http://www.macruby.org/`.

[10] Xavier Leroy. The Objective Caml system, August 2004.

[11] Yukihiro Matsumoto. *Ruby Language Reference Manual*, version 1.4.6 edition, February 1998. `http://docs.huihoo.com/ruby/ruby-man-1.4/yacc.html`.

[12] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC*, pages 213–228, 2002.

[13] NetBeans - integrated development environment with support for the Ruby language, May 2009. `http://www.netbeans.org/`.

[14] Emmanuel Onzon. `dypgen` User's Manual, January 2008.

[15] RadRails - Ruby on Rails authoring environment, May 2009. `http://aptana.com/rails/`.

[16] Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.

[17] Ruby Parser - Ruby parser written in pure Ruby, May 2009. `http://parsetree.rubyforge.org/`.

[18] Flog, Flay, and Heckle - Ruby source analysis tools, May 2009. `http://ruby.sadi.st/`.

[19] Bruce Stewart. An Interview with the Creator of Ruby, November 2001. `http://www.linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html`.

[20] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2nd edition, 2004.

[21] Bill Venners. The Philosophy of Ruby: A Conversation with Yukihiro Matsumoto, Part I, September 2003. `http://www.artima.com/intv/rubyP.html`.

[22] Yaml: Yaml ain't markup language, July 2009. `http://www.yaml.org/`.

[23] Yarv bytecode table, May 2009. `http://lifegoo.pluskid.org/upload/doc/yarv/yarv_iset.html`.