The Ruby Intermediate Language

Michael Furr Jong-hoon (David) An Jeffrey S. Foster Michael Hicks

Department of Computer Science
University of Maryland
College Park, MD 20742
{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. Our system includes an extensible GLR parser for Ruby, and an automatic translation into RIL, 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 in its representation.

We demonstrate the usefulness of RIL by presenting a simple static analysis and source code transformation to eliminate null pointer errors in Ruby programs. We also describe several additional useful features of RIL, including a pretty printer that outputs RIL as syntactically valid Ruby code, a dataflow analysis engine, and a dynamic instrumentation library for profiling source code. 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.

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 [4, 3].

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" [9] 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), a parser and intermediate representation 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 is written in OCaml, which provides strong support for working with the RIL data structure, due to its data type language and pattern matching features.

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 data flow 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 [**?**]. RIL also has a dataflow analysis engine, and extensive support for runtime 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 [3]. (Section 4.)

Along with DRuby [4, 3], we have used RIL to build DRails, a tool that brings static typing to Ruby on Rails

DLS '09 Orlando, Florida USA

applications (a work in progress). 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 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 [10, 2].

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 near-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 [11]. 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.

We illustrate the challenges in parsing Ruby with two 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* (method names and local variables names are described by the same production), 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
```

[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 [**?**].

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 |x| puts x end
```

prints the values 1 through 5. Notice that the body of for is also a code block—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.)

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.

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 [8]. 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. The production primary, defined on line 6, handles expressions that may appear nested within other expressions, like a method call (line 7) or a for loop (line 8). On line 10, 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 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.

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 disam-

```
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   primary:
7     | command_name[cmd] code_block[cb] { ... }
8     | K_FOR[pos] formal_arg_list [vars] K_IN arg[guard]
9       do_sep stmt_list [body] K_IEND
10      { well_formed_do guard body; E_For(vars,range,body,pos) }
```

Figure 1: Example GLR Code

```
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 2: Nested Assignment

biguation 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 [**?**], 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.

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

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:

```
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 represents all four cases internally using the third form.

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 [7]; 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 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 data flow analysis [1].

To illustrate some of the complexities of evaluation order in Ruby, consider the code in Figure 2(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 2(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

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 3: RIL Linearization Example

through a method body ends with a return statement and that every path through a block ends with a next statement.

Another tricky 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 3 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 an 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 by definition.

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 all together and implicitly uses the same values passed to the current method. For example, in

```
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
```

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 2(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 explicit, RIL requires every rescue clause to have an explicit class name, and inserts StandardError to mimic this sugar.

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?"
```

Similarly 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. USING RIL

In this section, we demonstrate RIL using three examples. First, we develop a simple transformation that uses dynamic instrumentation to prevent methods from being called on nil. Second, we construct a simple dataflow analysis to improve the performance of the transformed code. Finally, we describe an instrumentation library we built to enable profile-driven static analysis, discussing type inference in DRuby as a motivating example. Along the way, we illustrate some of the additional features our implementation provides to make it easier to work with RIL.

A complete grammar for RIL appears in Appendix A. In our implementation, RIL is represented as an OCaml data structure, and hence all our examples below are written in OCaml [6].

4.1 Transformation

As a first example, we define a 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* [**?**] to method invocations on nil,

which typically cause exceptions. (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.) As an optimization, we will not transform a call if the receiver is self, since self can never be nil. This particular transformation may or may not be useful, but it works well to demonstrate the use of RIL.

The input to our transformation is the name of a file, which is then parsed, transformed, and printed back to stdout. The top-level code for this is as follows:

```
1  let main fname =
2    let loader = File_loader.create File_loader.EmptyCfg [] in
3    let stmt = File_loader.load_file loader fname in
4    let new_stmt = visit_stmt (new safeNil) stmt in
5      CodePrinter.print_stmt stdout new_stmt
```

First, we use RIL's File_loader module to parse the given file (specified in the formal parameter fname), binding the result to stmt (lines 2–3). Next, we invoke new safeNil to create an instance of our transformation visitor, and pass that to visit_stmt to perform the transformation (line 4). This step performs the bulk of the work, and is discussed in detail next. Finally, we use the CodePrinter module to output the transformed RIL code as syntactically valid Ruby code, which can be directly executed (line 5). RIL also includes an ErrorPrinter module, which DRuby uses to emit code inside of error messages—since RIL introduces many temporary variables, the code produced by CodePrinter can be hard to understand. Thus, ErrorPrinter omits temporary variables (among other things), showing only the interesting part. For instance, if t1 is a temporary introduced by RIL, then ErrorPrinter shows the call t1 = f() as just f().

RIL's visitor objects are modeled after those in CIL [7]. A visitor 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          (* ... transform ... *)
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 sub-expressions, 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 [?] that lets us mix concrete and abstract syntax. To use it, we simply call RIL's reparse function:

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

Here the string passed on line 2 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 datastructures 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 1 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.

Putting this all together, the actual visitor pattern matching case for transforming a method call is

```
1  | MethodCall(_, {mc_target=#expr as targ}) →
2      let node' = reparse ~env:node.locals
3      reparse ~env:node.locals
4        "if %a.nil? then nil else %a end"
5        format_expr targ format_stmt node
6      in ChangeTo node'
```

Here we construct the new code as node′ and instruct the visitor to replace the existing node with this new statement (line 6).

4.2 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 simple 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 fact = MaybeNil | NonNil (* core dataflow facts *)
3    type t = fact StrMap.t
```

Next, we define top, which for our example will be the empty map:

```
4    let top = StringMap.empty
```

We choose the empty map rather than a map from all variables to NonNil since that way we can avoid computing the set of all local variables ahead of time. We omit the definitions of eq and to_string, which are straightforward.

Next, we define the meet function. Our meet semilattice uses the order MaybeNil < NonNil to describe the state of a single variable. We encode this relationship and extend it pointwise to maps:

```
5    (* compute meet of two facts *)
6    let meet_fact t1 t2 = match t1,t2 with
7      | MaybeNil, _ → MaybeNil
8      | _, MaybeNil → MaybeNil
9      | NonNil, NonNil → NonNil
10
11   (* update : string → fact → t → t *)
12   (* replace value of s in map with the meet of itself and v *)
13   let update s v map =
14     let fact =
15       try join_fact (StringMap.find s map) v
16       with Not_found → v
17     in StringMap.add s fact map
18
19   (* meet : t list → t *)
20   (* compute meet of all elements of lst *)
21   let meet lst =
22     List.fold_left
23       (fun acc map→StringMap.fold update map acc)
24       StringMap.empty lst
```

Finally, we define the transfer function, which, given the input dataflow facts map and a statement stmt 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 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. (Perhaps surprisingly, nil itself is actually an identifier in Ruby rather than a literal, and RIL follows the same convention.)

The function update_lhs has several cases, depending on the left-hand side. If it is a local variable, that variable's data flow 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 and then invoke the fixpoint function, which returns two hash tables of input and output facts at each statement:

```
1  module DataNil = Dataflow.Forwards(NilAnalysis)
2  let in_facts, out_facts = DataNil.fixpoint node in
```

Finally, we add a new case to our original visitor. We now check if a method target is a local variable and skip the instrumentation if it is:

```
1    ...
2  | MethodCall(_,
3      {mc_target=('ID_Var('Var_Local, var) as targ)}) →
4      let map = Hashtbl.find in_facts node in
5        begin match StrMap.find var map with
6          | NilAnalysis.MaybeNil → refactor targ node
7          | NilAnalysis.NonNil → SkipChildren
8        end
```

The complete code for this example appears in Appendix B.

4.3 Profile-Guided Analysis

The example presented so far has brushed aside an important detail: a significant amount of Ruby code, particularly in the Ruby standard library, pervasively uses highly dynamic methods 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. To precisely analyze code containing eval, then, 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 that would likely be quite imprecise.

Instead, we developed a dynamic analysis library that, among other things, lets us profile dynamic constructs. Using this library, we can keep track of what strings are passed to eval, and then use this information in the analysis from Section 4.2. Our most complex RIL client to date, Diamondback Ruby (DRuby), makes extensive use of this profiling infrastructure to very good effect [3].

Figure 4 shows the architecture of the analysis library, which 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 [3].

After we have discovered the set of application files, in stage two we instrument each file to record profiling information. This transformation is carried out like the one we saw in Section 4.1, except we modify the Ruby code to track strings passed to eval and several other dynamic constructs. We then unparse the modified source files to disk (using the CodePrinter module we already mentioned) 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.

Finally, we read in the gathered profiles and use them to transform the original source code prior to applying our main analysis. For example, consider NilAnalysis again. Suppose our target program contains the code eval "o.#{m}()", and profiling reveals that at run-time this eval is called with string "o.run()" (i.e., m contained string "run" when the eval was executed). The stage 4 transformation then replaces the eval call by o.run() directly. (In fact, we use a more general transformation in case x does not always evaluate to the same string [3].) As a result, when we run NilAnalysis it can properly instrument this method call, handling the case that o turns out to be nil. Without profiling, NilAnalysis would not see the call, and thus fail to eliminate a potential failure due to a nil receiver. In our work on DRuby, we found that this dynamic profiling technique, while potentially incomplete, works very well in practice, due to the restricted way Ruby programmer use highly dynamic features [3].

Our profiling and transformation infrastructure is fairly general, and can be used for things other than profiling dynamic constructs. For example, we could choose to use just the first three stages to collect profiling information for later analysis. For example, we might want to track all possible values of the first argument (the filename) passed to File.open for later tabulation. We can do this quite simply by writing

```
1  intercept_args  File , :open do |*args|  args [0]. to_s  end
```

(The instrumentation for the third stage in Figure 4 is specified partially in OCaml and partially in Ruby.) Our library will then store the file, line number, and a list of values that were passed to open. We were able to make good use of this flexibility by reusing portions of the library in DRails, in which we profile calls to special Ruby on Rails methods like before_filter and validate.

5. RELATED WORK

There are several threads of related work.

Another project that allows access to the Ruby AST is ruby_parser [**?**]. 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 [**?**] such as 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 [**?**, **?**] 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 [**?**]. 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 might be appropriate for uses similar to RIL.

While the Ruby language is defined by its C implementation, several other implementations exist, such as JRuby [5], IronRuby [**?**], and MacRuby [**?**]. 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 [7], 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.

6. 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;

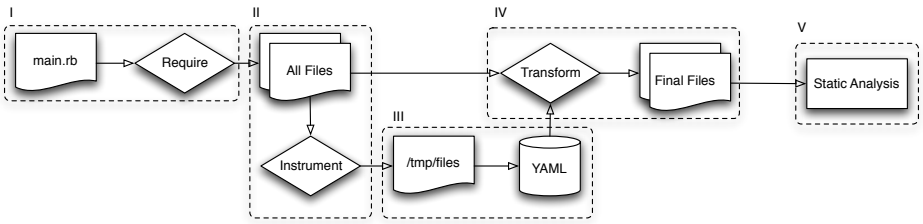

Figure 4: Dynamic Instrumentation Architecture

RIL makes Ruby's order of side effecting operations clear; and RIL makes explicit many implicit operations performed by the Ruby interpreter. 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.

7. REFERENCES

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

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

[3] 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. To appear.

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

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

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

[7] 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.

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

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

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

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

APPENDIX

A. RIL GRAMMAR

Figure 5 gives the full grammar for RIL. In the figure, optional elements are enclosed in $[\,]^?$. RIL separates statements s, which may have side effects, from expressions e, which are side-effect free.

$$
\begin{array}{rcl}
s & ::= & e \mid s; s \mid lval = e, \ldots, e \\
& \mid & \mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s\ \mathsf{end} \\
& \mid & \mathsf{case}\ e\ [\mathsf{when}\ e\ \mathsf{then}\ s]^?\ [\mathsf{else}\ s]^?\ \mathsf{end} \\
& \mid & \mathsf{while}\ e\ \mathsf{do}\ s\ \mathsf{end} \mid \mathsf{for}\ x = e\ \mathsf{in}\ s \\
& \mid & \mathsf{begin}\ s\ [\mathsf{rescue}\ e => x; s]^?\ [\mathsf{else}\ s]^?\ [\mathsf{ensure}\ s]^?\ \mathsf{end} \\
& \mid & [lval =]^? e.m(e, \ldots, e)\ [blk]^? \mid [lval =]^? \mathsf{yield}(e, \ldots, e) \\
& \mid & \mathsf{def}\ m(p)\ s\ \mathsf{end} \mid \mathsf{module}\ M; s\ \mathsf{end} \\
& \mid & \mathsf{class}\ C\ [< D]^?; s\ \mathsf{end} \mid \mathsf{class}\ <<\ e; s\ \mathsf{end} \\
& \mid & \mathsf{return}\ e \mid \mathsf{next}\ e \mid \mathsf{break}\ e \mid \mathsf{redo} \mid \mathsf{retry} \\
& \mid & \mathsf{alias}\ m\ m \mid \mathsf{undef}\ m \mid \mathsf{defined?}\ s \\
& \mid & \mathsf{BEGIN}\ \mathsf{do}\ s\ \mathsf{end} \mid \mathsf{END}\ \mathsf{do}\ s\ \mathsf{end}
\end{array}
$$

$$
\begin{array}{rcl}
lit & ::= & n \mid \text{“str”} \mid [e, \ldots, e] \mid \ldots \\
id & ::= & \mathsf{self} \mid x \mid @x \mid @@x \mid \$x \mid A \mid id :: A \\
e & ::= & id \mid lit \mid *e \mid \&e \\
lval & ::= & id \mid lval, \ldots, lval \mid *lval \\
p & ::= & x_1, \ldots, [x_n = e_n, \ldots]^?, [*x_m]^? \\
blk & ::= & \mathsf{do}\ |p|\ s\ \mathsf{end}
\end{array}
$$

$$
\begin{array}{rcl}
x & \in & \textit{local variable names} \\
@x & \in & \textit{instance variable names} \\
@@x & \in & \textit{class variable names} \\
\$x & \in & \textit{global variable names} \\
A & \in & \textit{constants} \\
m & \in & \textit{method names}
\end{array}
$$

Figure 5: Ruby Intermediate Language (RIL)

RIL comprises 24 statement forms, including expressions, sequencing, and parallel assignment. RIL includes just two conditional forms, an `if` statement, which models all of the simple branching forms, and a `case` statement, which models multiple branches. RIL also includes both `while` and `for` loops. We could translate `case` into a sequence of `if`'s or `for` loops into `while` loops, but we chose not to on the theory that a user may want to distinguish these constructs; we may revisit this choice in a future version of RIL.

The next statement form in RIL represents Ruby's exception handling construct, delimited by `begin...end`. Each clause `rescue` $e => x; s$ handles exceptions that are instances of class `e` or its subclasses, binding `x` to the exception in `s`. Unlike Ruby, in RIL the caught class is always included, as discussed in Section 3.3. The optional `else` block acts as a fall through case, catching any exception, and the `ensure` block is executed whether the exception is caught or not.

In RIL, method calls appear one per statement, and may not be nested, as we require all method arguments to be expressions and may optionally store their return value to in an expression using a assignment form. As in Ruby, method

calls may optionally pass a single code block do $|p|$ s end. This code block may be invoked inside the called function using the yield construct.

Methods are defined using the def keyword. Note that, as in Ruby, a method definition may occur anywhere in a statement list, e.g., it may occur conditionally depending on how an if statement evaluates. (However, no matter whether or when a method definition is executed, the defined method is always added to the lexically enclosing class.) After the regular parameters in a method definition, a parameter list p may contain zero or more *optional* arguments and may end with at most one *vararg* parameter, written $*x_m$. If a vararg parameter is present, then any actual parameters passed in positions m or higher are gathered into an array that is passed as argument x_m.

Modules and classes are defined using module and class, respectively. Class definitions may either specify a class name and an optional superclass, or may use the << notation to open the *eigenclass* of an object. As in Ruby, methods defined inside an object's eigenclass (so-called *eigenmethods*) are available only to that instance. For example, suppose x is an instance of A. Writing **class** << x; **def** m ... **end end** adds a method to x's eigenclass but not to A; thus no instance of A except x can be used to invoke m.

RIL contains several control flow statements. The return construct exits the current method, as is standard in most languages. Inside of a code block, next exits the block (using return inside the block would cause the lexically enclosing method to return). Similarly, the break statement acts as a remote return, returning control to the statement immediately following a block definition. For example:

```
1 def f()
2   z = yield()          # call code block argument
3   return z + 1
4 end
5 def g()
6   a = f() {return 2} # exits g with value 2
7   a = f() {next 3}   # jumps to line 2 storing 3 in z, a=4
8   a = f() {break 5}  # exits f() at line 2, a = 5
9 end
```

The redo and retry statements are used to re-execute a code block or exception block respectively.

RIL also includes constructs for several special Ruby statements: alias, which defines two method names to be the same; undef, which removes a method; defined?, which tests whether an expression or method is defined; and BEGIN and END, which specify code that is executed when a script is first loaded and when it exits, respectively. Note that there is no special representation for method creation (new) or for loading in additional files (require), since, as in Ruby, these are simply method calls.

In addition to statements, RIL also includes side-effect free expressions, identifiers and literals. Literals include values for Ruby's built in types such as integers n, strings "str", and arrays of expressions. Identifiers *id* include the distinguished variable self, local (x), instance (@x), and class (@@x) variables, as well as globals ($x) and *constants* A, which always begin with a capital letter. Constants can be assigned to exactly once. For example, $A = 1$ creates the constant A, which is read-only from the assignment statement forward. Class names are also constants, but are initialized with class rather than assignment. Constants may be nested inside of classes. The syntax C :: A extracts the constant A in class (or module) C. RIL expressions e consist of either a literal or an identifier and may include the $*$ and & unary operators. These operators are used to convert expressions to and from arrays and code blocks respectively. For example,x = [1,2]; a,b,c = 3,*x, assigns 3 to a, 1 to b, and 2 to c, and f(&p) calls f, passing the Proc object p (representing a higher-order method) as if it were a code block.

Finally, an *lval*, which may appear on the left-hand side of an assignment, is either an identifier or a sequence of *lval*s, which can be used for parallel assignment from an array. *lval*s may also use the $*$ operator to collect values into an array.

B. EXAMPLE CODE

Below is the complete code for the dataflow analysis described in Section 4.2.

```
open Cfg
open Cfg_printer
open Visitor
open Utils
open Cfg_refactor
open Cfg_printer.CodePrinter

module NilAnalysis = struct
  type fact = MaybeNil | NonNil

  let meet_fact t1 t2 = match t1,t2 with
    | MaybeNil, _
    | _, MaybeNil → MaybeNil
    | NonNil, NonNil → NonNil

  let update s v map =
    let fact =
      try meet_fact (StrMap.find s map) v
      with Not_found → v
    in StrMap.add s fact map

  let meet lst =
    List.fold_left (fun acc map →
                      StrMap.fold update map acc)
                    StrMap.empty lst

  let fact_to_s = function MaybeNil → "MaybeNil"
                         | NonNil → "NonNil"

  type t = fact StrMap.t
  let top = StrMap.empty
  let eq t1 t2 = StrMap.compare Pervasives.compare t1 t2 = 0
  let to_string t = strmap_to_string fact_to_s t

  let rec update_lhs fact map lhs = match lhs with
    | 'ID_Var('Var_Local, var) → update var fact map
    | # identifier → map
    | 'Tuple lst → List.fold_left (update_lhs MaybeNil) map lst
    | 'Star (#lhs as l) → update_lhs NonNil map l

  let transfer map stmt = match stmt.snode with
    | Assign(lhs, # literal) → update_lhs NonNil map lhs
    | Assign(lhs, 'ID_Var('Var_Local, rvar)) →
        update_lhs (StrMap.find rvar map) map lhs
    | Class(Some lhs,_,_) | Module(Some lhs,_,_)
    | MethodCall(Some lhs,_) | Yield(Some lhs,_)
    | Assign(lhs, _) → update_lhs MaybeNil map lhs

    | _ → map
end
```

```
module DataNil = Dataflow.Forwards(NilAnalysis)

let refactor targ node =
  let node' = freparse ~"env:node. lexical_locals
                         "unless %a.nil? then %a end"
    format_expr targ format_stmt node
  in ChangeTo node'

class safeNil (ifs,ofs) = object
 inherit default_visitor as super

 method visit_stmt node = match node.snode with
   | MethodCall(_, {mc_target='ID_Self}) → SkipChildren
   | MethodCall(_,
       {mc_target=('ID_Var('Var_Local,var) as targ)}) →
       let map = Hashtbl.find ifs node in
         begin match StrMap.find var map with
           | NilAnalysis.MaybeNil → refactor targ node
           | NilAnalysis.NonNil → SkipChildren
         end

   | MethodCall(_,
       {mc_target=#expr as targ}) → refactor targ node
   | _ → super#visit_stmt node
end
let main fname =
  let loader = File_loader.create File_loader.EmptyCfg [] in
  let s = File_loader.load_file loader fname in
  let () = compute_cfg s in
  let () = compute_cfg_locals s in
  let df = DataNil.fixpoint s in
  let s' = visit_stmt (new safeNil df) s in
    print_stmt stdout s'
```