Profile-Guided Static Typing for Dynamic Scripting Languages

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

Abstract

Many popular scripting languages such as Ruby, Python, and Perl include highly dynamic language constructs, such as an eval method that evaluates a string as program text. While these constructs allow terse and expressive code, they have traditionally obstructed static analysis. In this paper we present $\mathcal{P}$Ruby, an extension to Diamondback Ruby (DRuby), a static type inference system for Ruby. $\mathcal{P}$Ruby augments DRuby with a novel dynamic analysis and transformation that allows us to precisely type uses of highly dynamic constructs. $\mathcal{P}$Ruby's analysis proceeds in three steps. First, we use run-time instrumentation to gather per-application profiles of dynamic feature usage. Next, we replace dynamic features with statically analyzable alternatives based on the profile. We also add instrumentation to safely handle cases when subsequent runs do not match the profile. Finally, we run DRuby's static type inference on the transformed code to enforce type safety.

We used $\mathcal{P}$Ruby to gather profiles for a benchmark suite of sample Ruby programs. We found that dynamic features are pervasive throughout the benchmarks and the libraries they include, but that most uses of these features are highly constrained and hence can be effectively profiled. Using the profiles to guide type inference, we found that DRuby can generally statically type our benchmarks modulo some refactoring, and we discovered several previously unknown type errors. These results suggest that profiling and transformation is a lightweight but highly effective approach to bring static typing to highly dynamic languages.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; D.2.5 [*Programming Languages*]: Testing and Debugging—Tracing

General Terms Languages

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

Keywords Ruby, profile-guided analysis, RIL, Scripting Languages

1. Introduction

Many popular, object-oriented scripting languages such as Ruby, Python, and Perl are dynamically typed. Dynamic typing gives programmers great flexibility, but the lack of static typing can make it harder for "little" scripts to grow into mature, robust code bases. Recently, we have been developing Diamondback Ruby (DRuby), a tool that brings static type inference to Ruby.[1] DRuby aims to be simple enough for programmers to use while being expressive enough to precisely type typical Ruby programs. In prior work, we showed that DRuby could successfully infer types for small Ruby scripts (Furr et al. 2009c).

However, there is a major challenge in scaling up static typing to large script programs: Scripting languages typically include a range of hard-to-analyze, highly dynamic constructs. For instance, Ruby lets programmers eval strings containing source code, use reflection to invoke methods via send, and define a method_missing method to handle calls to undefined methods. These kinds of features lend themselves to a range of terse, flexible, and expressive coding styles, but they also impede standard static analysis. In fact, in Ruby it is even hard to statically determine what source files to analyze, because scripts can perform computation to decide what other files to load.

In this paper, we present $\mathcal{P}$Ruby, an extension to DRuby that solves this problem by combining run-time profiling of dynamic features with static typing.[2] Our key insight is that even though script programs may use constructs that *appear* to be dynamic, in fact their use is almost always heavily constrained, so that in practice they *act* statically. As an extreme example, a call eval "$x + 2$" is morally the same as the expression $x + 2$, and can be typed just as easily with $\mathcal{P}$Ruby. Using profiling enables $\mathcal{P}$Ruby to statically check many other, much more complex and interesting examples. And while $\mathcal{P}$Ruby is specific to typing Ruby, our profile-guided transformation technique can be applied to many dynamic languages and many static analyses.

[1] `http://www.cs.umd.edu/projects/PL/druby/`

[2] $\mathcal{P}$Ruby uses $\mathcal{P}$rofiling to handle dynamic features ignored by DRuby.

$\mathcal{P}$Ruby analyzes Ruby code in three steps. First, it performs a source-to-source translation on the program to be analyzed so that when run, the program records a *profile* of how dynamic features were used in that execution. Among other information, we record what strings are passed to eval, what methods are invoked via send, and what invocations are handled by method_missing. Next, the user runs the program to gather a sufficient profile, typically using the program's test suite. Then $\mathcal{P}$Ruby uses the profile to guide a program transformation that removes highly dynamic constructs, e.g., by replacing eval calls with the source code seen in the profile. Lastly, $\mathcal{P}$Ruby applies type inference to the transformed program to detect any type errors. $\mathcal{P}$Ruby can also safely handle program runs that do not match the profile. In these cases, $\mathcal{P}$Ruby instruments newly seen code to include full dynamic checking and blame tracking, so that we can detect errors in the code and place the blame appropriately.

Notice that $\mathcal{P}$Ruby relies on the programmer to provide test cases to guide profiling. We think this is a reasonable approach because not only do most Ruby programs already come with test suites (testing is widely adopted in the Ruby community), but it gives the programmer an easy to understand trade-off: The more dynamic features covered in the profile, the more static checking is achieved. Moreover, run-time profiling gives $\mathcal{P}$Ruby very precise information for type inference. This is in contrast to using, e.g., purely static string analysis (Livshits et al. 2005; Christensen et al. 2003; Gould et al. 2004), which could easily over-approximate the set of strings seen at run time (Sawin and Rountev 2007). It also allows us to statically analyze effectful dynamic code. For example, in our experiments, we found many cases where eval'd strings define methods, and those methods are referred to in other parts of the program. As far as we are aware, techniques such as gradual typing (Siek and Taha 2006, 2007; Herman et al. 2007) would be unsound in the presence of such effects in dynamic code—static guarantees could be undermined if dynamically eval'd code overwrites a method used in statically typed code.

We formalized profiling, transformation, and type checking for TinyRuby, a small object-oriented language with eval, send, and method_missing. We have proven that our transformation is *faithful*, meaning it does not change the behavior of a program under its profile, and that transformed programs that pass our type checker never go wrong at run time, except possibly from code that was instrumented with blame tracking.

We applied $\mathcal{P}$Ruby to a suite of benchmarks that use dynamic features, either directly, via the standard library, or via a third-party library. We found several interesting results. First, our experiments show that dynamic language features are used heavily in Ruby—across our benchmarks, profiled executions observed 664 unique strings passed to 66 syntactic occurrences of dynamic features, suggesting that handling such features is essential for any Ruby static analysis.

Second, we manually categorized all the dynamic feature usage in our sample runs, and we found that essentially all of them can be classified as "static." More precisely, approximately 2/3 of the time, dynamic features are used in a small, finite set of ways determined by the Ruby code that calls them. In the remaining cases, the calls to dynamic features depend on the local Ruby environment. We found no cases of arbitrarily dynamic code, e.g., there were no examples that eval'd a string read from the command line, or used send to call a method whose name was read from the network.

Finally, we found that DRuby initially reported many type errors on the transformed program code. Upon closer inspection, we found eight real type errors in widely used libraries. The remaining errors were false positives, but much of the code appeared "nearly" statically typable, despite being developed without a static type system in mind. To measure how statically typable this code is, we applied a range of refactorings to our benchmarks until they were accepted by DRuby. We found that the majority of refactorings point to potential improvements to DRuby, and a few more suggest places where Ruby coding style could be changed to be more amenable to static typing. We only found a few cases of code that uses untypable low-level object manipulation or requires dynamic typing.

Together, our results suggest that profile-guided transformation is an effective approach to help bring static typing to dynamic languages.

2. Motivation

Ruby is a class-based, imperative, object-oriented scripting language with a rich set of features such a module mixins, higher-order methods ("code blocks"), and strong regular expression support (Thomas et al. 2004; Flanagan and Matsumoto 2008). In this section, we motivate the need for $\mathcal{P}$Ruby by giving examples showing uses of its dynamic features. All of the examples in this section were extracted from the benchmarks in Section 5. $\mathcal{P}$Ruby also handles several other dynamic features of Ruby, discussed in Section 4.

Require To load code stored in a file, a Ruby program invokes the require method, passing the file name as a string argument. Since this is an ordinary method call, a Ruby program can actually perform run-time computation to determine which file to load. Figure 1(a) gives two examples of this. Lines 1–2, from the *sudokusolver* benchmark, call dirname to compute the directory containing the currently executing file, and then call File.join to create the path of the file to load. We have found similar calls to require (with computed strings) are common, occurring 11 times across 5 of our benchmarks. As another example, lines 4–7, from the *memoize* benchmark, first modify the load path on line 5 before loading the file memoize on line 7. This example shows

```
require File.join(File.dirname(__FILE__), '..',
                  'lib', 'sudokusolver')

Dir.chdir("..") if base == "test"
$LOAD_PATH.unshift(Dir.pwd + "/lib")
...
require "memoize"
```

(a) Using require with dynamically computed strings

```
1  alias gem_original_require require
2
3  def require(path)
4    gem_original_require path
5  rescue LoadError => load_error
6    (if spec = Gem.searcher.find(path) then
7       Gem.activate(spec.name, "= #{spec.version}")
8       gem_original_require path
9     else
10      raise load_error
11    end)
12 end end
```

(b) Example of require from *Rubygems* package manager

```
1  def initialize(args)
2    args.keys.each do |attrib|
3      self.send("#{attrib}=", args[attrib])
4  end end
```

(c) Use of send to initialize fields

```
1  ATTRIBUTES = ["bold", "underscore", ...]
2  ATTRIBUTES.each do |attr|
3    code = "def #{attr}(&blk) ... end"
4    eval code
5  end
```

(d) Defining methods with eval

```
1  def method_missing(mid, *args)
2    mname = mid.id2name
3    if mname =~ /=$/
4      ...
5      @table[mname.chop!.intern] = args[0]
6    elsif args.length == 0
7      @table[mid]
8    else
9      raise NoMethodError, "undefined method..."
10   end
11 end
```

(e) Intercepting calls with method_missing

Figure 1. Dynamic features in Ruby

that even when require is seemingly passed a constant string, its behavior may actually vary at run time.

For a much more complex use of require, consider the code in Figure 1(b). This example comes from *Rubygems*, a popular package management system for Ruby. In *Rubygems*, each package is installed in its own directory. *Rubygems* redefines the require method, as shown in the figure, so that require'ing a package loads it from the right directory. Line 1 makes an alias of the original require method. Then lines 3–11 give the new definition of require. First, line 4 attempts to load the file normally, using the old version of require. If that fails, the resulting LoadError exception is caught on line 5 and handled by lines 6–11. In this case, *Rubygems* searches the file system for a library of the same name (line 6). If found, the package is "activated" on line 7, which modifies the load path (as in Figure 1(a)), and then the file is loaded with the old require call on line 8.

This implementation is convenient for package management, but it makes pure static analysis quite difficult. Even if we could statically determine what string was passed to the new version of require, to find the corresponding file we would need to reimplement the logic of the Gem.searcher.find method. In $\mathcal{P}$Ruby, in contrast, we use dynamic profiling to discover which files are actually loaded, and thus no matter how complex the logic that finds them, we can determine the loaded files precisely.

Send When a Ruby program invokes e_0.send("meth", $e_1, \ldots, e_n$), the Ruby interpreter dispatches the call reflectively as e_0.meth($e_1, \ldots, e_n$). Figure 1(c) shows a typical use of this feature, from the *StreetAddress* benchmark. This code defines a constructor initialize that accepts a hash args as an argument. For each key attrib in the hash, line 3 uses send to pass args[attrib], the value corresponding to the key, to the method named "#{attrib} =", where #{e} evaluates expression e and inserts the resulting value into the string. For example, if initialize is called with the argument {"x" $\Rightarrow$ 1}, it will invoke the method self.x=(1), providing a lightweight way to configure a class through the constructor.

Another common use of send is in test drivers. For example, the Ruby community makes heavy use of Ruby's standard unit testing framework (not shown). To write a test case in this framework, the programmer creates a class with test methods whose names begin with test_. Given an instance of a test class, the framework uses the methods method to get a string list containing the names of the object's methods, and then calls the appropriate ones with send.

Eval Ruby also provides an eval method that accepts a string containing arbitrary code that is then parsed and executed. Our experiments show that use of eval is surprisingly common in Ruby—in total, eval and its variants are used to evaluate 423 different strings across all our benchmark runs (Section 5). Figure 1(d) shows one example of metaprogramming with eval, taken from the *text-highlight* bench-

$$
\begin{array}{rcl}
e & ::= & x \mid v \mid d \mid e_1; e_2 \mid e_1 \equiv e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
& \mid & \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid e_0.m(e_1, \ldots, e_n) \\
& \mid & \mathsf{eval}_\ell\ e \mid e_0.\mathsf{send}_\ell(e_1, \ldots, e_n) \\
& \mid & \mathsf{safe\_eval}_\ell\ e \mid [\![e]\!]_\ell \mid \mathsf{blame}\ \ell \\
v & ::= & s \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{new}\ A \mid [\![v]\!]_\ell \\
d & ::= & \mathsf{def}_\ell\ A.m(x_1, \ldots, x_n) = e
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
x & \in & \textit{local variable names} & A & \in & \textit{class names} \\
m & \in & \textit{method names} & s & \in & \textit{strings} \\
\ell & \in & \textit{program locations} & & &
\end{array}
$$

Figure 2. TinyRuby source language

mark. This code iterates through the ATTRIBUTES array defined on line 1, creating a method named after each array element on lines 3–4. We found many other examples like this, in which Ruby programmers use eval to create methods via macro-style metaprogramming.

Method Missing Figure 1(e) gives an example use of method_missing, which receives calls to undefined methods. This code (slightly simplified) is taken from the *ostruct* library, which creates record-like objects. In this definition, line 2 converts the first argument, the name of the invoked method, from a symbol to a string mname. If mname ends with = (line 3), then on line 5 we update @table to map mname (with the = removed and interned back into a symbol) to the first argument. Otherwise there must be no arguments (line 6), and we read the value corresponding to the invoked method out of @table. For example, if o is an instance of the *ostruct* class, the user can call o.foo = (3) to "write" 3 to foo in o, and o.foo() to "read" it back. Notice that we can use method invocation syntax even though method foo was never defined. This particular use of method_missing from *ostruct* is one of two occurrences of method_missing that are dynamically executed by our benchmark test suites.

One interesting property of method_missing is that it cannot be directly modeled using other dynamic constructs. In contrast, the require and send methods are in a sense just special cases of eval. We could implement require by reading in a file and eval'ing it, and we could transform o.send(m, x, y) into eval("o.#{m}(x, y)").

3. Dynamic Features in TinyRuby

We model our approach to statically type checking dynamic language features with TinyRuby, shown in Figure 2. The core language includes local variables x (such as the distinguished local variable self) and *values* v. Values include strings s, booleans true and false, objects created with new A, and *wrapped* values $[\![v]\!]_\ell$, which indicate values with dynamic rather than static types. We annotate $[\![v]\!]_\ell$ with a *program location* ℓ so that we may later refer to it. In TinyRuby, objects do not contain fields or per-object methods, and so we can represent an object simply by its class name. We could add richer objects to TinyRuby, but we keep the language simple to focus on its dynamic features.

In TinyRuby, method definitions d can appear in arbitrary expression positions, i.e., methods can be defined anywhere in a program. A definition $\mathsf{def}_\ell\ A.m(x_1, \ldots, x_n) = e$ adds or replaces class A's method m at program location ℓ, where the x_i are the arguments and e is the method body. Note that TinyRuby does not include explicit class definitions. Instead, a program may create an instance of an arbitrary class A at any point, even if no methods of A have been defined, and as we see occurrences of $\mathsf{def}_\ell\ A.m(\ldots) = \ldots$, we add the defined method to a *method table* used to look up methods at invocation time. For example, consider the following code:

$$\mathsf{let}\ x = \mathsf{new}\ A\ \mathsf{in}(\mathsf{def}_\ell\ A.m() = \ldots); x.m()$$

The call to $x.m()$ is valid because $A.m()$ was defined before the call, even though the definition was not in effect at new A. This mimics the behavior of Ruby, in which changes to classes affect all instances, and allows eval to be used for powerful metaprogramming techniques, as shown in Figure 1(d). Our method definition syntax also allows defining the special method_missing method for a class, which, as we saw in Section 2, receives calls to non-existent methods.

Other language constructs in TinyRuby include sequencing $e_1; e_2$, the equality operator $e_1 \equiv e_2$, let binding, conditionals with if, and method invocation $e_0.m(e_1, \ldots, e_n)$, which invokes method m of receiver e_0 with arguments e_1 through e_n.

TinyRuby also includes two additional dynamic constructs we saw in Section 2. The expression $\mathsf{eval}_\ell\ e$ evaluates e to produce a string s, and then parses and evaluates s to produce the result of the expression. The expression $e_0.\mathsf{send}_\ell(e_1, \ldots, e_n)$ evaluates e_1 to a string and then invokes the corresponding method of e_0 with arguments e_2 through e_n. We annotate both constructs with a program location ℓ.

The last three expressions in TinyRuby, $\mathsf{safe\_eval}_\ell\ e$, $[\![e]\!]_\ell$, and $\mathsf{blame}\ \ell$, are used to support dynamic typing and blame tracking. These expressions are inserted by our translation below to handle uses of dynamic constructs we cannot fully resolve with profiling. Our approach is somewhat non-standard, but these constructs in our formalism closely match our implementation (Section 4), which performs blame tracking without modifying the Ruby interpreter. We delay discussing the details of these expressions to Section 3.3.

3.1 An Instrumented Semantics

To track run-time uses of eval, send, and method_missing, we use the instrumented big-step operational semantics shown in Figure 3. Since most of the rules are straightforward, we show only selected, interesting reduction rules, and similarly for the other formal systems we discuss below. Full

$$\text{(VAR)}\quad \frac{}{\langle M, V, x\rangle \to \langle M, \emptyset, V(x)\rangle} \qquad \text{(DEF)}\quad \frac{}{\langle M, V, d\rangle \to \langle (d, M), \emptyset, \mathsf{false}\rangle}$$

$$\text{(EVAL)}\quad \frac{\langle M, V, e\rangle \to \langle M_1, \mathcal{P}_1, s\rangle \qquad \langle M_1, V, \mathit{parse}(s)\rangle \to \langle M_2, \mathcal{P}_2, v\rangle}{\langle M, V, \mathsf{eval}_\ell\ e\rangle \to \langle M_2, (\mathcal{P}_1 \cup \mathcal{P}_2 \cup [\ell \mapsto s]), v\rangle}$$

$$\text{(SEND)}\quad \frac{\begin{array}{c}\langle M, V, e_1\rangle \to \langle M_1, \mathcal{P}_1, s\rangle \qquad m = \mathit{parse}(s) \\ \langle M_1, V, e_0.m(e_2, \ldots, e_n)\rangle \to \langle M_2, \mathcal{P}_2, v\rangle\end{array}}{\langle M, V, e_0.\mathsf{send}_\ell(e_1, \ldots, e_n)\rangle \to \langle M_2, (\mathcal{P}_1 \cup \mathcal{P}_2 \cup [\ell \mapsto s]), v\rangle}$$

$$\text{(CALL-M)}\quad \frac{\begin{array}{c}\langle M_i, V, e_i\rangle \to \langle M_{i+1}, \mathcal{P}_i, v_i\rangle \qquad i \in 0..n \qquad v_0 = \mathsf{new}\ A \\ (\mathsf{def}_\ell\ A.m(\ldots) = \ldots) \notin M_{n+1} \\ (\mathsf{def}_{\ell'}\ A.\mathsf{method\_missing}(x_1, \ldots, x_{n+1}) = e) \in M_{n+1} \\ s = \mathit{unparse}(m) \qquad m \neq \mathsf{method\_missing} \\ V' = [\mathsf{self} \mapsto v_0, x_1 \mapsto s, x_2 \mapsto v_1, \ldots, x_{n+1} \mapsto v_n] \\ \langle M_{n+1}, V', e\rangle \to \langle M', \mathcal{P}', v\rangle\end{array}}{\langle M_0, V, e_0.m(e_1, \ldots, e_n)\rangle \to \langle M', (\bigcup_i \mathcal{P}_i) \cup \mathcal{P}' \cup [\ell' \mapsto s], v\rangle}$$

Figure 3. Instrumented operational semantics (partial)

proofs are available in a companion technical report (Furr et al. 2009a). In our implementation, we add the instrumentation suggested by our semantics via a source-to-source translation.

Reduction rules in our semantics have the form $\langle M, V, e\rangle \to \langle M', \mathcal{P}, v\rangle$. Here M and M' are the initial and final *method tables*, containing a list of method definitions; V is a *local variable environment*, mapping variables to values; e is the expression being reduced; v is the resulting value; and $\mathcal{P}$ is a *profile* that maps program locations (occurrences of eval, send, and method_missing definitions) to sets of strings. In these rules, we use $\mathit{parse}(s)$ to denote the expression produced by parsing string s, and we use $\mathit{unparse}(e)$ to denote the string produced by unparsing e.

The first rule, (VAR), looks up a variable in the local environment and produces the empty set of profiling information. To see why we opted to use environments rather than a substitution-based semantics, consider the program $\mathsf{let}\ x = 2\ \mathsf{in}\ \mathsf{eval}_\ell\ \text{“}x + 1\text{”}$. In a substitution-based semantics, we would rewrite this program as $(\mathsf{eval}_\ell\ \text{“}x+1\text{”})[x \mapsto 2]$, but clearly that will not work, since this is equal to $(\mathsf{eval}_\ell\ \text{“}x + 1\text{”})$, i.e., substitution does not affect strings. We could try extending substitution to operate on string arguments to eval, but since the string passed to eval can be produced from an arbitrary expression, this will not work in general. Other choices such as delaying substitution until later seemed complicated, so we opted for the simpler semantics using variable environments.

The next rule, (DEF), adds a method definition to the front of M and returns false. When we look up a definition of $A.m$ in M, we find the leftmost occurrence, and hence (DEF) replaces any previous definition of the same method.

The last three rules in Figure 3 handle the novel features of TinyRuby. (EVAL) reduces its argument e to a string s, parses s and then reduces the resulting expression to compute the final result v. The resulting profile is the union of the profiles $\mathcal{P}_1$ (from evaluating e), $\mathcal{P}_2$ (from evaluating $\mathit{parse}(s)$), and $[\ell \mapsto s]$, which means s should be added to the set of strings associated with ℓ. In this way, we track the relationship between $\mathsf{eval}_\ell\ e$ and the string s passed to it a run-time.

(SEND) behaves analogously. We evaluate the first argument, which must produce a string, translate this to a method name m, and finally invoke m with the same receiver and remaining arguments. In the output profile, we associate the location of the send with the string s.

Finally, (CALL-M) handles invocations to undefined methods. In this rule we evaluate the receiver and arguments, but no method m has been defined for the receiver class. We then look up method_missing of the receiver class and evaluate its body in environment V', which binds the first formal parameter to s, the name of the invoked method, and binds self and the remaining formal parameters appropriately. The output profile associates s with ℓ, the location where method_missing was defined.

3.2 Translating Away Dynamic Features

After profiling, we can translate a TinyRuby program into a simpler form that eliminates features that are hard to analyze statically. Figure 4 gives a portion of our translation. Excluding the final rule, our translation uses judgments of the form $\mathcal{P} \vdash e \rightsquigarrow e'$, meaning given profile $\mathcal{P}$, we translate expression e to expression e'. For most language forms, we either do nothing, as in $(\text{REFL}_{\rightsquigarrow})$, or translate sub-expressions recursively, as in $(\text{SEQ}_{\rightsquigarrow})$; we omit other similar rules.

The first interesting rule is $(\text{EVAL}_{\rightsquigarrow})$, which translates $\mathsf{eval}_\ell\ e$. First, we recursively translate e. Next, recall that (EVAL) in Figure 3 includes in $\mathcal{P}(\ell)$ any strings evaluated by this occurrence of eval. We parse and translate those strings s_j to yield expressions e_j. Then we replace the call to eval by a conditional that binds e' to a fresh variable x (so that e' is only evaluated once) and then tests x against the strings in $\mathcal{P}(\ell)$, yielding the appropriate e_j if we find a match. If not, we fall through to the last case, which evaluates the string with $\mathsf{safe\_eval}_\ell\ x$, a “safe” wrapper around eval that adds additional dynamic checks we describe below (Section 3.3). This catch-all case allows execution to continue even if we encounter an unprofiled string, and also allows us to blame the code from location ℓ if it causes a subsequent run-time type error. In our formalism, adding the form $\mathsf{blame}\ \ell$ allows us to formally state soundness: TinyRuby programs that are profiled, transformed, and type checked never get stuck at run time, and reduce either to values or to blame. In practice, by tracking blame we can also give the user better error messages.

$$\frac{e \in \{x, v, \mathsf{blame}\ \ell\}}{\mathcal{P} \vdash e \rightsquigarrow e}\ (\textsc{Refl}_{\rightsquigarrow}) \qquad \frac{\mathcal{P} \vdash e_1 \rightsquigarrow e_1' \quad \mathcal{P} \vdash e_2 \rightsquigarrow e_2'}{\mathcal{P} \vdash e_1; e_2 \rightsquigarrow e_1'; e_2'}\ (\textsc{Seq}_{\rightsquigarrow})$$

$$\frac{\mathcal{P} \vdash e \rightsquigarrow e' \quad \mathcal{P} \vdash \mathit{parse}(s_j) \rightsquigarrow e_j \quad s_j \in \mathcal{P}(\ell) \quad x \text{ fresh} \quad e'' = \left(\begin{array}{l}\mathsf{let}\ x = e'\ \mathsf{in} \\ \quad \mathsf{if}\ x \equiv s_1\ \mathsf{then}\ e_1 \\ \quad \mathsf{else\ if}\ x \equiv s_2\ \mathsf{then}\ e_2 \ldots \\ \quad \mathsf{else\ safe\_eval}_\ell\ x\end{array}\right)}{\mathcal{P} \vdash \mathsf{eval}_\ell\ e \rightsquigarrow e''}\ (\textsc{Eval}_{\rightsquigarrow})$$

$$\frac{\mathcal{P} \vdash e_i \rightsquigarrow e_i' \quad i \in 0..n \quad s_j \in \mathcal{P}(\ell) \quad x \text{ fresh} \quad e' = \left(\begin{array}{l}\mathsf{let}\ x = e_1'\ \mathsf{in} \\ \quad \mathsf{if}\ x \equiv s_1\ \mathsf{then}\ e_0'.\mathit{parse}(s_1)(e_2', \ldots, e_n') \\ \quad \mathsf{else\ if}\ x \equiv s_2\ \mathsf{then}\ e_0'.\mathit{parse}(s_2)(e_2', \ldots, e_n') \ldots \\ \quad \mathsf{else\ safe\_eval}_\ell\ \text{“}e_0'.\text{”} + x + \text{“}(e_2', ..., e_n')\text{”}\end{array}\right)}{\mathcal{P} \vdash e_0.\mathsf{send}_\ell(e_1, \ldots, e_n) \rightsquigarrow e'}\ (\textsc{Send}_{\rightsquigarrow})$$

$$\frac{\mathcal{P} \vdash e \rightsquigarrow e' \quad s_j \in \mathcal{P}(\ell) \quad e'' = \left(\begin{array}{l}\mathsf{def}_\ell\ A.\mathit{parse}(s_1)(x_2, \ldots, x_n) = (\mathsf{let}\ x_1 = s_1\ \mathsf{in}\ e'); \\ \mathsf{def}_\ell\ A.\mathit{parse}(s_2)(x_2, \ldots, x_n) = (\mathsf{let}\ x_1 = s_2\ \mathsf{in}\ e'); \ldots\end{array}\right)}{\mathcal{P} \vdash \mathsf{def}_\ell\ A.\mathsf{method\_missing}(x_1, \ldots, x_n) = e \rightsquigarrow e''}\ (\textsc{Meth-Missing}_{\rightsquigarrow})$$

$$\frac{\mathcal{P} \vdash e \rightsquigarrow e' \quad (\mathsf{def}_{\ell_j}\ A^j.m^j(x_1^j, \ldots, x_n^j) = \ldots) \in e' \quad e_d = \left(\begin{array}{l}\mathsf{def}_{\ell_1}\ A^1.m^1(x_1^1, \ldots, x_{n1}^1) = \mathsf{blame}\ \ell_1; \\ \mathsf{def}_{\ell_2}\ A^2.m^2(x_1^2, \ldots, x_{n2}^2) = \mathsf{blame}\ \ell_2; \ldots\end{array}\right)}{\mathcal{P} \vdash e \rightrightarrows (e_d; e')}\ (\textsc{Prog}_{\rightsquigarrow})$$

Figure 4. Transformation to static constructs (partial)

($\textsc{Send}_{\rightsquigarrow}$) is similar to ($\textsc{Eval}_{\rightsquigarrow}$). We recursively translate the receiver e_0 and arguments e_i. We replace the invocation of send with code that binds fresh variable x to the first argument, which is the method name, and then tests x against each of the strings s_j in $\mathcal{P}(\ell)$, which were recorded by (SEND) in our semantics. If we find a match, we invoke the appropriate method directly. While our formal rule duplicates e_i' for each call to send, in our implementation these expressions are side-effect free (i.e., they consist only of literals and identifiers), and so we actually duplicate very little code in practice. Otherwise, in the fall-through case, we call safe_eval with a string that encodes the method invocation—we concatenate the translated expressions e_i' with appropriate punctuation and the method name x. (Note that in this string, by e_i' we really mean *unparse*(e_i'), but we elide that detail to keep the formal rule readable.)

($\textsc{Meth-Missing}_{\rightsquigarrow}$) follows a similar pattern. First, we recursively translate the body as e'. For each string s_j in $\mathcal{P}(\ell)$ (which by (CALL-M) in Figure 3 contains the methods intercepted by this definition), we define a method named s_j that takes all but the first argument of method_missing. The method body is e', except we bind x_1, the first argument, to s_j, since it may be used in e'.

Our approach to translating method_missing completely eliminates it from the program, and there is no fall-through case. There are two advantages to this approach. First, a static analysis that analyzes the translated program need not include special logic for handling method_missing. Second, it may let us find places where method_missing intercepts the wrong method. For example, if our profiling runs show that A.method_missing is intended to handle methods foo and bar, DRuby's type system will complain if it sees a call to an undefined A.baz method in the translated program. We believe this will prove more useful to a programmer than simply assuming that a method_missing method is intended to handle arbitrary calls. However, one consequence of this approach is that if a program is rejected by DRuby's type system, then unprofiled calls to method_missing would cause the program to get stuck.

The last step in the translation is to insert "empty" method definitions at the top of the program. We need this step so we can formally prove type soundness. For example, consider a program with a method definition and invocation:

$$\ldots \mathsf{def}_\ell\ A.m(\ldots) = e; \ldots; (\mathsf{new}\ A).m(\ldots); \ldots$$

The challenge here is that the definition of $A.m$ might occur under complex circumstances, e.g., under a conditional, or deep in a method call chain. To ensure $(\mathsf{new}\ A).m(\ldots)$ is valid, we must know $A.m$ has been defined.

One solution would be to build a flow-sensitive type system for TinyRuby, i.e., one that tracks "must be defined" information to match uses and definitions. However, in our experience, this kind of analysis would likely be quite complex, since definitions can appear anywhere, and it may be hard for a programmer to predict its behavior.

Instead, we assume that any method syntactically present in the source code is available everywhere and rely on dynamic, rather than static, checking to find violations of our assumption. Translation $\mathcal{P} \vdash e \rightrightarrows (e_d; e')$, defined by ($\textsc{Prog}_{\rightsquigarrow}$) in Figure 4, enforces this discipline. Here e_d is a sequence of method definitions, and e' is the translation of e using the other rules. For each definition of $A.m$ occurring in e', we add a mock definition of $A.m$ to e_d, where the body of the mock definition signals an error using blame ℓ to blame the location of the actual definition.

We could also have built e_d from the method definitions actually seen during execution, e.g., (DEF) in Figure 3 could record what methods are defined. We think this would also be a reasonable design, but would essentially require that users have tests to drive profiling runs in order to statically analyze their code, even if they do not use features such as eval. Thus for a bit more flexibility, we build e_d based

(SEVAL)

$$\frac{\langle M, V, e\rangle \to \langle M', \mathcal{P}, s\rangle \quad \mathit{parse}(s) \hookrightarrow_\ell e' \quad \langle M', V, [\![e']\!]_\ell\rangle \to \langle M'', \mathcal{P}', v\rangle}{\langle M, V, \mathsf{safe\_eval}_\ell\ e\rangle \to \langle M', \mathcal{P} \cup \mathcal{P}', v\rangle}$$

(IF$_\hookrightarrow$)

$$\frac{e_1 \hookrightarrow_\ell e_1' \quad e_2 \hookrightarrow_\ell e_2' \quad e_3 \hookrightarrow_\ell e_3'}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow_\ell \mathsf{if}\ [\![e_1']\!]_\ell\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3'}$$

(CALL$_\hookrightarrow$)

$$\frac{e_i \hookrightarrow_\ell e_i' \quad i \in 0..n}{e_0.m(e_1, \ldots, e_n) \hookrightarrow_\ell [\![e_0']\!]_\ell.m(e_1', \ldots, e_n')}$$

(DEF$_\hookrightarrow$)

$$\frac{}{\mathsf{def}_{\ell'}\ A.m(x_1, \ldots, x_n) = e \hookrightarrow_\ell \mathsf{blame}\ \ell'}$$

(IF-WRAP-T)

$$\frac{\langle M, V, e_1\rangle \to \langle M_1, \mathcal{P}_1, [\![\mathsf{true}]\!]_\ell\rangle \quad \langle M_1, V, e_2\rangle \to \langle M_2, \mathcal{P}_2, v_2\rangle}{\langle M, V, \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \to \langle M_2, (\mathcal{P}_1 \cup \mathcal{P}_2), v_2\rangle}$$

(IF-WRAP-BLAME)

$$\frac{\langle M, V, e_1\rangle \to \langle M_1, \mathcal{P}_1, v\rangle \quad v \in \{[\![s]\!]_\ell, [\![\mathsf{new}\ A]\!]_\ell\}}{\langle M, V, \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \to \langle M_1, \mathcal{P}_1, \mathsf{blame}\ \ell\rangle}$$

(CALL-WRAP)

$$\frac{\begin{array}{c}\langle M_i, V, e_i\rangle \to \langle M_{i+1}, \mathcal{P}_i, v_i\rangle \quad i \in 0..n \quad v_0 = [\![\mathsf{new}\ A]\!]_{\ell''} \\ (\mathsf{def}_\ell\ A.m(x_1, \ldots, x_n) = e) \in M_{n+1} \\ m \neq \mathsf{method\_missing} \\ V' = [\mathsf{self} \mapsto v_0, x_1 \mapsto [\![v_1]\!]_{\ell''}, \ldots, x_n \mapsto [\![v_n]\!]_{\ell''}] \\ \langle M_{n+1}, V', e\rangle \to \langle M', \mathcal{P}', v\rangle\end{array}}{\langle M_0, V, e_0.m(e_1, \ldots, e_n)\rangle \to \langle M', (\bigcup_i \mathcal{P}_i) \cup \mathcal{P}', [\![v]\!]_{\ell''}\rangle}$$

Figure 5. Safe evaluation rules (partial)

on static occurrences of definitions, but we might make dynamic method definition tracking an option in the future.

3.3 Safe Evaluation

To handle uses of dynamic features not seen in a profile, our translation in Figure 4 inserts calls to $\mathsf{safe\_eval}_\ell\ e$, a "safe" wrapper around eval. Figure 5 gives some of the reduction rules for this form. In the first rule, (SEVAL), we reduce $\mathsf{safe\_eval}_\ell\ e$ by evaluating e to a string s, parsing s, translating the result to e' via the $\hookrightarrow_\ell$ relation (a source-to-source transformation), and then evaluating $[\![e']\!]_\ell$, a wrapped e'. The expression $[\![e']\!]_\ell$ behaves the same as e', except if it is used type-unsafely then our semantics produces $\mathsf{blame}\ \ell$, meaning there was an error due to dynamic code from ℓ. This is contrast to type-unsafe uses of unwrapped values, which cause the semantics to go wrong (formally, reduce to *error*). In practice, we implement $[\![e']\!]_\ell$ by a method that accepts an object and modifies it to have extra run-time checking (Section 4).

The relation $e \hookrightarrow_\ell e'$ rewrites the expression e, inserting $[\![\cdot]\!]_\ell$ where needed. We give three example rewrite rules. (IF$_\hookrightarrow$) rewrites each subexpression of the if, wrapping the guard since its value is consumed. Similarly, (CALL$_\hookrightarrow$) wraps the receiver so that at run time we will check the receiver's type and blame ℓ if the call is invalid. Lastly, (DEF$_\hookrightarrow$) replaces a method definition by blame—we cannot permit methods to be redefined in dynamically checked code, since this could undermine the type safety of statically typed code.

When wrapped values are used, we unwrap them and either proceed as usual or reduce to blame ℓ. For example, (IF-WRAP-T) evaluates the true branch of an if given a guard that evaluates to $[\![\mathsf{true}]\!]_\ell$, whereas (IF-WRAP-BLAME) evaluates to blame ℓ if the guard evaluates to a non-boolean. Notice the contrast with ordinary reduction, which would instead go wrong when if is used with a non-boolean guard.

(CALL-WRAP) handles a method invocation in which the receiver is a wrapped object. Here we must be careful to also wrap the arguments (in the definition of V') when evaluating the method body; because we did not statically check that this call was safe, we need to ensure that the arguments' types are checked when they are used in the method body. Similarly, we must wrap the value returned from the call so that it is checked when used later.

Notice that our semantics for $\mathsf{safe\_eval}_\ell\ e$ does not use any static type information. Instead, it performs extensive object wrapping and forbids method definitions in dynamic code. One alternative approach would be to run DRuby's type inference algorithm at run time on the string e returns. However, this might incur a substantial run-time overhead (given the space and time requirements of $\mathcal{P}$Ruby's type inference system), and it disallows any non-statically typed parts of the program. Another alternative would be to only keep objects wrapped until they are passed to statically typed code. At that point, we could check their type against the assumed static type, and either fail or unwrap the object and proceed. This would be similar to gradual typing (Siek and Taha 2006, 2007; Herman et al. 2007). We may explore this approach in the future, as having static types available at run time could reduce the overhead of our wrappers at the expense of additional space overhead.

3.4 Formal Properties

It should be clear from the discussions above that our translation preserves the character of the original program, with respect to the core behavior and the dynamic features seen during the profiling run(s). We can prove this formally:

THEOREM 1 (Translation Faithfulness). *Suppose* $\langle \emptyset, \emptyset, e\rangle \to \langle M, \mathcal{P}', v\rangle$ *and* $\mathcal{P}' \subseteq \mathcal{P}$ *and* $\mathcal{P} \vdash e \rightrightarrows e'$. *Then there exist* $M_\mathcal{P}$ *such that* $\langle \emptyset, \emptyset, e'\rangle \to \langle M_\mathcal{P}, \emptyset, v\rangle$.

In other words, if we translate an expression based on its profile (or a superset of the information in its profile), both the original and translated program produce the same result.

Also, since our translation has removed all dynamic features, we will record no additional profiling information in the second execution, making the final profile $\emptyset$.

In our formal system, an expression e always evaluates to the same result and produces the same profile, but in practice, programs may produce different profiles under different circumstances. For example, if we want to test the behavior of e, we could evaluate $e; e_1$, where e_1 is a test case for the expression e, and $e; e_2$, where e_2 is a different test case. Based on the above theorem, if our profiling runs are sufficient, we can use them to translate programs we have not yet profiled without changing their behavior:

COROLLARY 2. *Suppose* $\langle \emptyset, \emptyset, (e; e_1) \rangle \rightarrow \langle M_1, \mathcal{P}_1, v_1 \rangle$. *Further, suppose that* $\langle \emptyset, \emptyset, (e; e_2) \rangle \rightarrow \langle M_2, \mathcal{P}_2, v_2 \rangle$. *Then if* $\mathcal{P}_2 \subseteq \mathcal{P}_1$ *and* $\mathcal{P}_1 \vdash (e; e_2) \rightrightarrows e'$, *then* $\langle \emptyset, \emptyset, e' \rangle \rightarrow \langle M_2', \emptyset, v_2 \rangle$.

In other words, if the dynamic profile $\mathcal{P}_1$ of $(e; e_1)$ covers all the dynamic behavior of $(e; e_2)$, then using $\mathcal{P}_1$ to translate $e; e_2$ will not change its behavior. In our experiments, we found that many dynamic constructs have only a limited range of behaviors, and hence can be fully represented in a profile. Thus, by this theorem, most of the time we can gather a profile and then use that to transform many different uses of the program.

Finally, the last step is to show that we can perform sound static analysis on the translated program. A companion technical report gives a (mostly standard) type system for this language (Furr et al. 2009a). Our type system proves judgments of the form $MT \vdash e$, meaning under *method type table MT*, a mapping from method names to their types, program e is well-typed. In order for our type system to be sound, we forbid well-typed programs from containing eval, send, or method_missing (since we cannot check these statically), though programs may contain uses of safe_eval and $[\![\cdot]\!]_\ell$ (which are checked dynamically). We can formally prove the following type soundness theorem, where r stands for either a value, blame ℓ, or *error*, an error generated if the expression goes wrong:

THEOREM 3 (Type Soundness). *If* $\emptyset \vdash e$ *and* $\langle \emptyset, \emptyset, e \rangle \rightarrow \langle M, \mathcal{P}, r \rangle$, *then* r *is either a value or* blame ℓ. *Thus,* $r \neq$ *error.*

This theorem says that expressions that are well-typed in this language do not go wrong.

Recall that the translation from Section 3.2 eliminates the three dynamic features that this type system does not permit, and inserts appropriate mock definitions at the beginning of the program. Thus, if we start with an arbitrary program, gather information about its dynamic feature usage via the instrumentation in Figure 3, and translate it according to Figure 4, we can then apply sound static type checking to the resulting program, while still precisely modeling uses of eval, send, and method_missing in the original program.

4. Implementation

As discussed earlier, $\mathcal{P}$Ruby is an extension to Diamondback Ruby (DRuby), a static type inference system for Ruby. DRuby accepts standard Ruby programs and translates them into the Ruby Intermediate Language (RIL), a much simpler subset of Ruby designed for analysis and transformation (Furr et al. 2009b). DRuby performs static type inference internally on RIL code, and reports any type errors to the user. DRuby supports a wide range of typing constructs, including intersection and union types, optional method arguments and varargs methods, self types, object types with fields, parametric polymorphism, mixins, tuple types, and first class method types, among others (Furr et al. 2009c).

$\mathcal{P}$Ruby is a drop-in replacement for the regular Ruby interpreter. The user runs $\mathcal{P}$Ruby with the command

```
druby --dr-profile filename.rb
```

This command runs *filename.rb* to gather a profile, transforms the program to eliminate dynamic constructs according to the profile (as in Section 3.2), and then runs DRuby's type inference on the resulting program. In the future, we expect profiling to be done separately and the results saved for later use, but for experimental purposes our current all-in-one setup is convenient. Altogether, $\mathcal{P}$Ruby, which includes the enhanced DRuby source, comprises approximately 16,000 lines of OCaml and 800 lines of Ruby.

There are three interesting implementation issues in $\mathcal{P}$Ruby: performing profiling, handling additional dynamic constructs, and implementing safe_eval and its relatives.

4.1 Profiling

$\mathcal{P}$Ruby creates profiles by running an instrumented version of the source code. $\mathcal{P}$Ruby first must discover what source files, in addition to the one specified on the command line, are executed and hence need to be instrumented; as we saw in Section 2, this is hard to determine statically. To find the set of executed files, $\mathcal{P}$Ruby runs the original program but with special code prepended to replace the definitions of require and load[3] with new methods that record the set of loaded files and log them to disk when the program exits. Since both methods are affected by the current load path, which may be changed by the program, we log that as well.

Next, $\mathcal{P}$Ruby parses all files seen in require and load calls, translates them into RIL, and adds instrumentation to record uses of eval, send, method_missing, and other dynamic features, to mimic the semantics in Section 3.1. Finally, we unparse the transformed RIL code into `/tmp`, and then run the output code to compute a profile. The instrumentation is generally straightforward, though care must be taken to ensure the program runs correctly when executed in `/tmp`.

[3] Ruby's load is similar to require, but it always (re-)evaluates the given file, even if previously loaded, while require evaluates a file only once.

4.2 Additional Dynamic Constructs

In addition to the constructs discussed in Section 3, $\mathcal{P}$Ruby also handles several other closely related dynamic features. Similarly to eval, Ruby includes instance_eval, class_eval, and module_eval methods that evaluate their string argument in the context of the method receiver (an instance, class, or module, respectively). For example, calling

x.class_eval("def foo()...end")

adds the foo method to the class stored in variable x. We profile these methods the same way as eval, but we use a slightly different transformation. For example, we replace the above code by

x.class_eval() do def foo()...end end

Here we keep the receiver object x in the transformed program, because the definition is evaluated in x's context. DRuby recognizes this form of class_eval (which is also valid Ruby code) specially, analyzing the body of the code block in x's context. Our transformation for instance_eval and module_eval is similar.

Ruby includes four methods for accessing fields of objects, {instance, class}_variable_{get, set}, which take the name of the instance or class variable to read or write. When $\mathcal{P}$Ruby profiles these methods, it records the variable name and transforms the expression into calls to {instance, class}_eval. For example, we transform a.instance_variable_set ("@x", 2) into a.instance_eval do @x = 2 end.

Finally, $\mathcal{P}$Ruby also includes support for attr and attr_{reader, writer, accessor}, which create getter/setter methods given a field name, and also for const_{get, set}, which directly read or write constants (write-once variables). $\mathcal{P}$Ruby profiles calls to these methods, and replaces the non-literal field or constant name arguments with the string literals seen at run time. DRuby then specially handles the case when these methods are called with string literals. For example, when DRuby sees const_set("X", 3), it will give the constant X the type Fixnum. These constructs are translated similarly to how the other dynamic features are treated, e.g., by inserting calls to safe_eval for unseen strings.

Ruby includes some dynamic features $\mathcal{P}$Ruby does not yet support. In particular, DRuby's type system treats certain low-level methods specially, but these methods could be redefined, effectively changing the semantics of the language. For instance, if a programmer changes the Module# append_features method, they can alter the semantics of module mixins. Other special methods include Class#new, Class#inherited, Module#method_added, and Module# included. $\mathcal{P}$Ruby also does not support applying dynamic constructs to per-object classes (eigen-classes) or calling dynamic features via the Method class. In addition to these features, $\mathcal{P}$Ruby currently does not support const_missing, which handles accesses to undefined constants, similarly to method_missing; we expect to add support for this in the future.

Currently, $\mathcal{P}$Ruby does not support nested dynamic constructs, e.g., eval'ing a string with eval inside it, or send'ing a message to the eval method. In these cases, $\mathcal{P}$Ruby will not recursively translate the nested construct. We believe these restrictions could be lifted with some engineering effort.

4.3 Implementing safe_eval

We implemented safe_eval$_\ell$ e, $[\![e]\!]_\ell$, and blame ℓ as two components: a small Ruby library with methods safe_eval(), wrap(), and blame(), and druby_eval, an external program for source-to-source translation.

The druby_eval program is written using RIL, and it implements the $\hookrightarrow_\ell$ translation as shown in Figure 5. For example, it translates method definitions to calls to blame(), and it inserts calls to wrap() where appropriate. There are a few additional issues when implementing $\hookrightarrow_\ell$ for the full Ruby language. First, we need not wrap the guard of if, because in Ruby, if accepts any object, not just booleans. Second, in addition to forbidding method definitions, we must also disallow calls to methods that may change the class hierarchy, such as undef_method. Lastly, we add calls to wrap() around any expressions that may escape the scope of safe_eval, such as values assigned to global variables and fields.

Given druby_eval, our library is fairly simple to implement. The safe_eval() method simply calls druby_eval to translate the string to be evaluated and then passes the result to Ruby's regular eval method. The blame() method aborts with an appropriate error. Lastly, the wrap() method uses a bit of low-level object manipulation (in fact, exactly the kind $\mathcal{P}$Ruby cannot analyze) to intercept method calls: Given an object, wrap() first renames the object's methods to have private names beginning with __druby, then calls undef_method to remove the original methods, and lastly adds a method_missing definition to intercept all calls to the (now removed) original methods. Our method_missing code checks to see if the called method did exist. If so, it delegates to the original method with wrapped arguments, also wrapping the method's return value. If not, it calls blame().

One nice feature of our implementation of wrap() is that because we do not change the identity of the wrapped object, we preserve physical equality, so that pointer comparisons work as expected. Our approach does not quite work for instances of Fixnum and Float, as they are internally represented as primitive values rather than via pointed-to objects. Instead, we wrap these objects by explicitly boxing them inside of an traditional object. We then extend the comparison methods for these classes to delegate to the values inside these objects when compared.

5. Profiling Effectiveness

We evaluated $\mathcal{P}$Ruby by running it on a suite of 13 programs downloaded from RubyForge. We included any dependencies directly used by the benchmarks, but not any optional components. Each benchmark in our suite uses at least some

Benchmark	LoC	Req	Eval	Snd	**Total**
ai4r-1.0	764	4/ 4	2/ 2	4/ 4	10/ 10
bacon-1.0.0	258	·	·	·	·
hashslice-1.0.4	78	·	·	·	·
hyde-0.0.4	115	2/ 2	1/11	1/ 2	4/ 15
isi-1.1.4	224	·	1/ 1	·	1/ 1
itcf-1.0.0	178	·	·	·	·
memoize-1.2.3	69	·	·	1/ 1	1/ 1
pit-0.0.6	166	2/ 2	·	·	2/ 2
sendq-0.0.1	88	·	·	·	·
StreetAddress-1.0.1	875	1/ 1	·	1/15	2/ 16
sudokusolver-1.4	188	2/ 2	1/ 1	·	3/ 3
text-highlight-1.0.2	262	·	2/48	·	2/ 48
use-1.2.1	193	·	·	·	·
Total	3,458	11/11	7/63	7/22	25/ 96

Req – dyn. require and load
Eval – eval and variants
Snd – send and __send__
n/m – n=occ, m=uniq strs
G/S – field and constant get/set; attr and its variants
MM – method_missing

(a) Per-benchmark results (no occ. of MM or G/S)

Lib Module	LoC	Req	Eval	Snd	G/S	MM	**Total**
archive-tar-minitar	539	·	·	·	2/ 32	·	2/ 32
date	1,938	·	3/ 33	·	·	·	3/ 33
digest	82	1/ 1	·	·	1/ 1	·	2/ 2
fileutils	950	·	4/101	·	·	·	4/101
hoe	502	1/ 2	·	1/ 2	·	·	2/ 4
net	2,217	·	1/ 8	·	·	·	1/ 8
openssl	637	·	3/ 2	·	·	·	3/ 20
optparse	964	·	·	2/ 4	·	·	2/ 4
ostruct	80	·	·	2/ 2	·	1/ 9	3/ 11
pathname	511	·	·	1/ 1	·	·	1/ 1
rake	1,995	2/19	3/136	·	·	·	5/155
rubyforge	500	·	1/ 2	·	·	·	1/ 2
rubygems	4,146	·	4/ 32	·	4/ 68	·	8/100
tempfile	134	·	·	1/ 2	·	1/ 2	2/ 4
term-ansicolor	78	·	1/ 28	·	·	·	1/ 28
testunit	1,293	·	·	1/63	·	·	1/ 63
Other	4,871	·	·	·	·	·	·
Total	21,437	4/22	20/360	8/74	7/101	2/11	41/568

(b) Library results (as covered by benchmarks)

Figure 6. Dynamic feature profiling data from benchmarks

of the dynamic language features handled by $\mathcal{P}$Ruby, either in the application itself or indirectly via external libraries. All of our benchmarks included test cases, which we used to drive the profiling runs for our experiments. Finally, many projects use the rake program to run their test suites. Rake normally invokes tests in forked subprocesses, but as this would make it more difficult to gather profiling information, we modified rake to invoke tests in the same process.

5.1 Dynamic Feature Usage

Figure 6 measures usage of the dynamic constructs we saw in our profiling runs. We give separate measurements for the benchmark code (part (a)) and the library modules used by the benchmarks (part (b)). We should note that our measurements are only for features seen during our profiling runs—the library modules in particular include other uses of dynamic features, but they were in code that was not called by our benchmarks.

For each benchmark or module, we list its lines of code (computed by SLOCCount (Wheeler 2008)) and a summary of the profiling data for its dynamic features, given in the form n/m, where n is the number of syntactic occurrences called at least once across all runs, and m is the number of unique strings used with that feature. For Req and G/S, we only count occurrences that are used with non-constant strings. Any library modules that did not execute any dynamic features are grouped together in the row labeled *Other* in Figure 6(b).

These results clearly show that dynamic features are used pervasively throughout our benchmark suite. All of the features handled by $\mathcal{P}$Ruby occur in some program, although method_missing is only encountered twice. Eight of the 13 benchmarks and more than 75% of the library module code use dynamic constructs. Perhaps surprisingly (given its power) eval is the most commonly used construct, occurring 27 times and used with 423 different strings—metaprogramming is indeed extremely common in Ruby. Over all benchmarks and all libraries, there were 66 syntactic occurrences of dynamic features that cumulatively were used with 664 unique strings. Given these large numbers, it is critical that any static analysis model these constructs to ensure soundness.

5.2 Categorizing Dynamic Features

The precision of DRuby's type inference algorithm depends on how much of the full range of dynamic feature usage is observed in our profiles. To measure this, we manually categorized each syntactic occurrence from Figure 6 based on how "dynamically" it is used. For example, eval "x + 2" is not dynamic at all since the eval will always evaluate the same string, whereas eval ($stdin.readline) is extremely dynamic, since it could evaluate any string.

Figure 7 summarizes our categorization. We found that all of the dynamic features in our profiles are used in a controlled manner—their use is either determined by the class they are called in, or by the local user's Ruby environment. In particular, we found no examples of truly dynamic code, e.g., eval'ing code supplied on the command line, suggesting that profiling can be used effectively in practice. We now discuss each category in detail.

Single The least dynamic use of a construct is to always invoke it with the same argument. Three uses of eval and seven uses of send can only be passed a single string. For instance, the *sudokusolver* benchmark includes the code

```
PROJECT = "SudokuSolver"
PROJECT_VERSION =
  eval("#{PROJECT}::VERSION")
```

which is equivalent to SudokuSolver::VERSION. As another example, the *ostruct* module contains the code

```
meta.send(:define_method, name) { @table[name] }
```

This code uses send to call the private method define_method from outside the class. The other uses of send in this category were similar.

Collection A slightly more expressive use of dynamic constructs is to apply them to a small, fixed set of arguments. One common idiom (18 occurrences) we observed was to apply a dynamic construct uniformly across a fixed collection of values. For example, the code in Fig. 1(d) iterates over an Array of string literals, evaling a method definition string from each literal. Thus, while multiple strings are passed to this occurrence of eval, the same strings will be used for every execution of the program. Additionally, any profile that executes this code will always see all possible strings for the construct.

Bounded We also found some dynamic constructs that are called several times via different paths (in contrast to being called within the same iteration over a collection), but the set of values used is still bounded. For example, consider the following code from the *pathname* module:

```
if RUBY_VERSON < "1.9"
  TO_PATH = :to_str
else TO_PATH = :to_path end
path = path.__send__(TO_PATH)
```

Here one of two strings is passed to send, depending on the library version.

Sometimes dynamic constructs are called in internal methods of classes or modules, as in the following example from the *net/https* library:

```
def self.ssl_context_accessor(name)
  HTTP.module_eval(<<-End, __FILE__, __LINE__ + 1)
    def #{name}() ... end       # defines get method
    def #{name}=(val) ... end   # defines set method
    end
  End
end
ssl_context_accessor :key
ssl_context_accessor : cert_store
```

This code defines method ssl_context_accessor, which given a symbol generates get and set methods based on that name. The body of the class then calls this method to add several such get/set methods. This particular method is only used in the class that defines it, and seems not to be intended for use elsewhere (nor is it used anywhere else in our benchmarks).

Features in this category are also essentially static, because their behavior is determined by the class they are contained in, and profiling, even in isolation, should be fully effective. Combining this with the previous two categories gives a total of 42 features used with 538 unique strings, which means around 2/3 of the total dynamic feature usage across all runs is essentially static.

Category	Req	Eval	Snd	G/S	MM	**Total**
Single	·	3/ 3	7/ 7	·	·	10/ 10
Collection	·	14/337	1/ 2	3/ 48	·	18/387
Bounded	·	7/ 69	4/20	3/ 52	·	14/141
File system	11/11	3/ 14	·	·	·	14/ 25
Open module	4/22	·	3/67	1/ 1	2/11	10/101
Total	15/33	27/423	15/96	7/101	2/11	66/664

n/m – n=occ, m=uniq strs

Figure 7. Categorization of profiled dynamic features

File System The next category covers those dynamic features whose use depends on the local file system. This includes most occurrences of Req, e.g., the code at the top of Figure 1(a), which loads a file who name is derived from __FILE__, the current file name. Another example is the following convoluted code from *rubyforge*:

```
config = File.read(__FILE__).split(/__END__/).last.gsub(
           /#\{(.*)\}/) { eval $1 }
```

This call reads the current file, removes any text that appears before __END__ (which signals the Ruby interpreter to stop reading), and then substitutes each string that matches the given pattern with the result of calling eval on that string. Despite its complexity, for any given installation of the library module, this code always evaluates the same set of strings.

The other cases of this category are similar to these two, and in all cases, the behavior of the dynamic constructs depends on the files installed in the user's Ruby environment.

Open module The last category covers cases in which dynamic features are called within a library module, but the library module itself does not determine the uses. For example, the *testunit* module uses send to invoke test methods that the module users specify. Similarly, the *rake* module loads client-specified Ruby files containing test cases. As another example, the *ostruct* module is used to create record-like objects, as shown in Figure 1(e).

These cases represent an interesting trade-off in profiling. If we profile the library modules in isolation, then we will not see all client usage of these 10 constructs (hence they are "open"). However, if we assume the user's Ruby environment is fixed, i.e., there are no new .rb files added at run time, then we can fully profile this code, and therefore we can perform full static typing checking on the code.

6. Type Inference

Finally, we used $\mathcal{P}$Ruby to perform type inference on each of the benchmarks, i.e. $\mathcal{P}$Ruby gathered the profiling data reported in Figure 6, transformed the code as outlined in Sections 3 and 4, and then applied DRuby's type inference algorithm on the resulting program.

When we first ran $\mathcal{P}$Ruby on our benchmarks, it produced hundreds of messages indicating potential type errors. As we began analyzing these results, we noted that most of the messages were false positives, meaning the code would actually execute type safely at run time. In fact, we found

Benchmark	Total LoC	Time (s)
ai4r-1.0	21,589	343
bacon-1.0.0	19,804	335
hashslice-1.0.4	20,694	307
hyde-0.0.4	21,012	345
isi-1.1.4	22,298	373
itcf-1.0.0	23,857	311
memoize-1.2.3	4,171	9
pit-0.0.6	24,345	340
sendq-0.0.1	20,913	320
StreetAddress-1.0.1	24,554	309
sudokusolver-1.4	21,027	388
text-highlight-1.0.2	2,039	2
use-1.2.1	20,796	323

Figure 8. Type inference results

that much of the offending code is *almost* statically typable with DRuby's type system. To measure how "close" the code is to being statically typable, we manually applied a number of refactorings and added type annotations so that the programs pass DRuby's type system, modulo several actual type errors we found.

The result gives us insight into what kind of Ruby code programmers "want" to write but is not easily amenable to standard static typing. (DRuby's type system combines a wide variety of features, but most of the features are well-known.) In the remainder of this section, we discuss the true type errors we found (Section 6.1), what refactorings were needed for static typing (Section 6.2), and what we learned about the way people write Ruby programs (Section 6.3). Overall, we found that most programs could be made statically typable, though in a few cases code seems truly dynamically typed.

6.1 Performance and Type Errors

Figure 8 shows the time it took $\mathcal{P}$Ruby to analyze our modified benchmarks. For each benchmark, we list the total lines of code analyzed (the benchmark, its test suite, and any libraries it uses), along with the analysis time. Times were the average of three runs on an AMD Athlon 4600 processor with 4GB of memory. These results show that $\mathcal{P}$Ruby's analysis takes only a few minutes, and we expect the time could be improved further with more engineering effort.

Figure 9 lists, for each benchmark or library module used by our benchmarks, its size, the number of refactorings and annotations we applied (discussed in detail in the next section), and the number of type errors we discovered. The last row, *Other*, gives the cumulative size of the benchmarks and library modules with no changes and no type errors.

$\mathcal{P}$Ruby identified eight type errors, each of which could cause a program crash. The two errors in the *pathname* module were due to code that was intended for the development branch of Ruby, but was included in the current stable version. In particular, *pathname* contains the code

```
def world_readable?() FileTest.world_readable?(@path) end
```

Module	LoC	Refactorings	Annots	Errors
archive-minitar	538	3	·	1
date	1,938	58	8	·
digest	82	1	·	·
fileutils	950	1	7	·
hoe	502	3	2	·
net	2,217	22	3	·
openssl	637	3	3	1
optparse	964	15	21	·
ostruct	80	1	·	·
pathname	511	21	1	2
pit-0.0.6	166	2	·	·
rake	1,995	17	7	·
rational	299	3	25	·
rbconfig	177	1	·	·
rubyforge	500		7	·
rubygems	4,146	44	47	4
sendq-0.0.1	88	1	·	·
shipit	341	4	·	·
tempfile	134	1	3	·
testunit	1,293	3	20	·
term-ansicolor	78	1		·
text-highlight-1.0.2	262	1	1	·
timeout	59	1	1	·
uri	1,867	15	20	·
webrick	435	4	1	·
Other	4,635	·	·	·
Total	24,895	226	177	8

Figure 9. Changes needed for static typing

However, the FileTest.world_readable? method is in the development version of Ruby but not in the stable branch that was used by our benchmarks. The second error in *pathname* is a similar case with the world_writable? method.

The type error in *archive-minitar* occurs in code that attempts to raise an exception but refers to a constant incorrectly. Thus, instead of throwing the intended error, the program instead raises a NameError exception.

The four type errors in *rubygems* were something of a surprise—this code is very widely used, with more than 1.6 million downloads on rubyforge.org, and so we thought any errors would have already been detected. Two type errors were simple typos in which the code incorrectly used the Policy class rather than the Policies constant. The third error occurred when code attempted to call the non-existent File.dir? method. Interestingly, this call was exercised by the *rubygems* test suite, but the test suite defines the missing method before the call. We are not quite sure why the test suite does this, but we contacted the developers and confirmed this is indeed an error in *rubygems*. The last type error occurred in the $=\sim$ method, which compares the @name field of two object instances. This field stores either a String or a Regexp, and so the body of the method must perform type tests to ensure the types are compatible. However, due to a logic error, one of the four possible type pairings is handled incorrectly, which could result in a run time type error.

Finally, the *openssl* module adds code to the Integer class that calls OpenSSL :: BN :: new(self). In this call, self has type Integer, but the constructor for the OpenSSL :: BN

class takes a string argument. Therefore, calling this code always triggers a run-time type error.

6.2 Changes for Static Typing

To enable our benchmarks and their libraries to type check (modulo the above errors), we applied 226 refactorings and added 177 type annotations. We can divide these into the following categories. For the moment, we refrain from evaluating whether these changes are reasonable to expect from the programmer, or whether they suggest possible improvements to $\mathcal{P}$Ruby; we discuss this issue in detail in Section 6.3.

Dynamic Type Tests (177 Annotations) Ruby programs often use a single expression to hold values with a range of types. Accordingly, DRuby supports union types (e.g., A or B) and intersection types (e.g., (Fixnum → Fixnum) **and** (Float → Float)). However, DRuby does not currently model run-time type tests specially. For example, if e has type A or B, then DRuby allows a program to call methods present in *both* A and B, but it does not support dynamically checking if e has (just) type A and then invoking a method that is in A but not in B.

To work around this limitation, we developed an annotation for conditional branches that allows programmers to indicate the result of a type test. For example, consider the following code:

```
1 case x
2 when Fixnum: ###% x : Fixnum
3   x + 3
4 when String:  ###% x : String
5   x.concat ''world''
6 end
```

Here, the case expression on line 1 tests the class of x against two possibilities. The annotations on lines 2 and 4 tell DRuby to treat x as having type Fixnum and String, respectively, on each branch. These annotations were extremely common—we added them to 135 branches in total.We also added 9 method annotations for intersection types and 33 method annotations for higher order polymorphic types. Polymorphic type signatures can be used by DRuby given annotations, but cannot currently be inferred. DRuby adds instrumentation to check all the above annotations dynamically at run time, to ensure they are correct.

Class Imprecision (81 Refactorings) In Ruby, classes are themselves objects that are instances of the Class class. Furthermore, "class methods" are actually methods bound inside of these instances. In many cases, we found programmers use Class instances returned from methods to invoke class methods. For example, consider the following code:

```
1   class A
2     def A.foo() ... end
3     def bar()
4       self.class.foo()   # calls A.foo()
5     end
6   end
```

Here the call on line 4 goes to the class method defined on line 2. However, the class method invoked on line 4 has type () → Class in DRuby, and since Class has no foo() method, DRuby rejects the call on line 4. To let examples like this type check, we changed self.class to use a different method call that dispatches to the current class. For example,

```
def bar()
  myclass().foo()
end
def myclass()
  A
end
```

Similarly, an instance can look up a constant dynamically in the current class using the syntax **self**.**class**::X, requiring an analogous transformation.

Block Argument Counts (24 Refactorings) In Ruby, higher-order methods can accept *code blocks* as arguments. However, the semantics of blocks is slightly different than regular methods. Surprisingly, Ruby does not require the formal parameter list of a block to exactly match the actual arguments: formal arguments not supplied by the caller are set to nil, and extra actual arguments are ignored.

DRuby, on the other hand, requires strict matching of the number of block arguments, since otherwise we could never discover mismatched argument counts for blocks. Thus we modified our benchmarks where necessary to make arguments lists match. We believe this is the right choice, because satisfying DRuby's requirement is a very minor change.

Non-Top Level Requires (21 Refactorings) $\mathcal{P}$Ruby uses profiling to decide which files are required during a run, and therefore which files should be included during type checking. However, some of our benchmarks had conditional calls to require that were never triggered in our test runs, but that we need for static typing. For instance, the URI module contains the following code:

```
1   if target.class == URI::HTTPS
2     require 'net/https'
3     http.verify_mode = OpenSSL::SSL::VERIFY_PEER
```

Here line 2 loads net/https if the conditional on line 1 is true. The method called on line 3 is added by a load-time eval inside of net/https. Thus, to successfully analyze this code, $\mathcal{P}$Ruby needs to not only analyze the source code of net/https, but it also must have its profile to know this method exists. However, the branch on line 1 was never taken in our benchmarks, and so this require was never executed and the eval was not included in the profile.

We refactored cases like this by moving the require statement outside of the method, so that it was always executed when the file is loaded.

Multiple Configurations (10 Refactorings) We encountered some code that behaves differently under different operating environments. For example,

```
if defined?(Win32)
  .... # win32 code
end
```

first checks if the constant Win32 is defined before using windows-specific methods and constants in the body of the if. As another example, consider this code from *rubygems*:

```
1  if RUBY_VERSION < '1.9' then
2    File.read file_name
3  else
4    File.read file_name, :encoding => 'UTF-8'
```

In versions prior to Ruby 1.9 (the current development version of Ruby), the read method only took a single parameter (line 2), whereas later versions accept a second parameter (line 4). When DRuby sees this code, it assumes both paths are possible and reports that read is called with the wrong number of arguments. To handle these type-conflicting cases, we commented out sections of code that were disabled by the platform configuration.

Heterogeneous Containers (12 Refactorings) DRuby supports homogeneous containers with types such as Array<T> and Hash<K,V>. Since arrays are sometimes used heterogeneously, DRuby also includes a special type Tuple<T_1, ..., T_n>, where the T_i are the tuple element types from left to right. Such a type is automatically coerced to Array<T_1 **or** ... **or** T_n> when one of its methods is invoked.

However, sometimes this automatic coercion causes type errors. For instance, the *optparse* module contains the following code:

```
1  def append(*args)
2    update(*args)
3    @list.push(args[0])
4  end
```

Here, calling the [] method on line 3 forces args to have a homogeneous array type, losing precision and causing a type error. We refactored this code to list the arguments to append explicitly, allowing DRuby to type check this method. We also encountered several other similar cases, as well as examples where instances of Hash were used heterogeneously.

Flow-insensitive Locals (11 Refactorings) DRuby treats local variables flow-sensitively, since their type may be updated throughout the body of a method. To be sound, we conservatively treat any local variables that appear inside of a block flow-insensitively (Furr et al. 2009c). However, this causes DRuby to report an error if a flow-insensitive local variable is assigned conflicting types at different program points. We eliminated these errors by introducing a fresh local variable at each conflicting assignment and renaming subsequent uses.

Other (65 Refactorings) We also needed a few other miscellaneous refactorings. In our benchmarks, there were 32 calls handled by method_missing that were never seen in our benchmark runs. Hence $\mathcal{P}$Ruby reported these calls as going to undefined methods. We fixed this by manually copying the method_missing bodies for each method name they were called with, simulating our translation rules. We could also have fixed this with additional test cases to expand our profiles, so that $\mathcal{P}$Ruby would add these methods automatically during its transformation.

In some cases, DRuby infers union types for an object that actually has just one type. For example, *rubygems* includes a Package.open method that returns an instance of either TarInput or TarOutput, depending on whether a string argument is "r" or "w." DRuby treats the result as having either of these types, but as they have different methods, this causes a number of type errors. We fixed this problem by directly calling TarInput.open or TarOutput.open instead. A similar situation also occurred in the *uri* module.

We also refactored a few other oddball cases, such as a class that created its own include method (which DRuby would confuse with Module.include) and some complex array and method manipulation that could be simplified into typable code.

Untypable Code (12 Refactorings) Finally, some of the code we encountered could not reasonably be statically typed, even with refactorings and checked annotations. One example is the *optparse* class, which provides an API for command line parsing. Internally, *optparse* manipulates many different argument types, and because of the way the code is structured, DRuby heavily conflates types inside the module. We were able to perform limited refactoring inside of *optparse* to gain some static checking, but ultimately could only eliminate all static type errors by manually wrapping the code using the wrap() method from our safe_eval library (Section 4.3).

The other cases of untypable code were caused by uses of low-level methods that manipulate classes and modules directly in ways that DRuby does not support. For example, we found uses of remove_method, undef_method, and anonymous class creation. We also found uses of two modules that perform higher-level class manipulation: Singleton, which ensures only one instance of a class exists, and Delegate, which transparently forwards method calls to a delegate class. DRuby does not support code that uses these low-level features and will not detect any run-time errors from their misuse.

6.3 Discussion and Future Work

In our prior work on DRuby, we found that small benchmarks are mostly statically typable. We believe our current results with $\mathcal{P}$Ruby suggest that even large Ruby programs are mostly statically typable—on balance, most of our refactorings and type annotations indicate current limitations of

DRuby, and a few more suggest places where Ruby programmers could easily change their code to be typable (e.g., making argument counts for blocks consistent). Given the extreme flexibility of Ruby, we think this result is very encouraging, and it suggests that static typing could very well succeed in practice.

Our results suggest a number of future directions for $\mathcal{P}$Ruby. Dynamic type tests are clearly important to Ruby programmers but are not modeled by DRuby. Occurrence Typing (Tobin-Hochstadt and Felleisen 2008), previously proposed for Scheme, is one possible solution we plan to explore. One challenge we expect is that Ruby contains a multitude of ways to test the dynamic type of a value, and we need to strike the right balance between supporting common uses and producing an easy-to-use system. Similarly, improved handling of the Class type and a more precise analysis for flow-sensitive local variables would be beneficial. Combined, these changes could eliminate up to 76% of the annotations and 41% of the refactorings we introduced.

Other coding idioms may be difficult to support with DRuby's type inference algorithm, but could be handled with improvements to our profiling technique. For example, currently $\mathcal{P}$Ruby performs profiling, transformation, and type inference in one run (Section 4). If we could combine profiles from multiple runs, we could run additional tests to improve code coverage. For example, instead of hoisting require to the top-level of a file, a better solution may be to use additional test suites (such as those provided by a library maintainer), or for libraries to ship a profile database that could be used by library clients.

Along the same lines, commenting out code to handling multiple configurations will not work in practice. A better solution might be to annotate particular constants as *configuration variables* whose values are then profiled by $\mathcal{P}$Ruby. DRuby could then use these profiles to automatically prune irrelevant code sections.

Our results so far show that $\mathcal{P}$Ruby can be applied to existing code bases that were not written with static typing in mind. Ultimately, we believe that $\mathcal{P}$Ruby will be most useful to programmers while they are developing their code, so that potential errors can be caught early in the development life cycle. In the future, we plan not only to continue to improve $\mathcal{P}$Ruby technically, but also to directly study usability and utility of $\mathcal{P}$Ruby for software developers.

7. Threats to Validity

There are several potential threats to the validity of our results. Figures 6(a) and (b) only include dynamic constructs that were observed by our benchmark runs. As we mentioned earlier, there are also other dynamic constructs that are present in the code (particularly the library modules) but were not called via our test suites. However, additional profiling to try to exhibit these features would only bolster our claim that dynamic features are important to model. A more important consequence is that our categorization in Figure 7 may not generalize. It is possible that if we examined more constructs, we would find other categories or perhaps some features used in very dynamic ways. However, this would not affect our other results, and we believe we looked at enough occurrences (66 total) to gather useful information.

In Ruby, it is possible for code to "monkey-patch" arbitrary classes, changing their behavior. Monkey patching could invalidate our categorization from Section 5.2, e.g., by exposing a dynamic feature whose uses were previously bounded within a class. However, this would only affect our categorization and not $\mathcal{P}$Ruby, which can still easily profile and analyze the full, monkey-patched execution.

Similarly, Ruby's low-level object API could allow a programmer to subvert our analysis, as discussed at the end of Section 6.2. Because we cannot verify these unsafe features, they could potentially disable our run-time instrumentation, causing a Ruby script to fail. However, we hope that programmers who use unsafe features will treat them with appropriate caution.

8. Related Work

There are several threads of related work. $\mathcal{P}$Ruby is an extension to DRuby (Furr et al. 2009c), which implements static type inference for Ruby. The key contribution of $\mathcal{P}$Ruby is our sound handling of highly dynamic language constructs. Our prior work on DRuby avoided these features by sticking to small examples, using programmer annotations for library APIs, and eliminating dynamic constructs with manual transformation. However, as we saw in Section 5, highly dynamic features are pervasive throughout Ruby, and so this approach is ultimately untenable. Kristensen (2007) has also developed a type inference system for Ruby based on the cartesian product algorithm. This system does not handle any of Ruby's dynamic features, making it unsound in the presence of these constructs.

In addition to DRuby, researchers have proposed a number of other type systems for dynamic languages including Scheme (Cartwright and Fagan 1991; Tobin-Hochstadt and Felleisen 2008), Smalltalk (Graver and Johnson 1990; Strongtalk; Wuyts 2007), Javascript (Thiemann 2005; Hansen 2007; Anderson et al. 2005), and Python (Salib 2004; Aycock 2000; Cannon 2005), though these Python type systems are aimed at performance optimization rather than at the user level. To our knowledge, none of these systems handles send, eval, or similar dynamic features.

One exception is RPython (Ancona et al. 2007), a system that inspired our work on $\mathcal{P}$Ruby. RPython translates Python programs to type safe back-ends such as the JVM. In RPython, programs may include an initial bootstrapping phase that uses arbitrary language features, including highly dynamic ones. RPython executes the bootstrapping phase using the standard Python interpreter, and then produces a type safe output program based on the interpreter state. The key

differences between RPython and $\mathcal{P}$Ruby are that $\mathcal{P}$Ruby supports dynamic feature use at arbitrary execution points; that we include a formalization and proof of correctness; that we provide some information about profile coverage with test runs; and, perhaps foremost, that $\mathcal{P}$Ruby operates on Ruby rather than Python.

Another approach to typing languages with dynamic features is to use the type Dynamic (Abadi et al. 1991). Extensions of this idea include quasi-static typing (Thatte 1990), gradual type systems (Siek and Taha 2006, 2007; Herman et al. 2007), and hybrid types (Gronski et al. 2006). However, we believe these approaches cannot handle cases where dynamic code might have side effects that interact with (what we would like to be) statically typed code. For example, recall the code from Figure 1(d), which uses eval to define methods. Since these definitions are available everywhere, they can potentially influence any part of the program, and it is unclear how to allow some static and some dynamic typing in this context. In contrast, $\mathcal{P}$Ruby explicitly supports constructs that would *look* dynamic to a standard type system, but *act* essentially statically, because they have only a few dynamic behaviors that can be seen with profiling; for code that is truly dynamic, $\mathcal{P}$Ruby reverts to full dynamic checking.

Several researchers have proposed using purely static approaches to eliminating dynamic language constructs. Livshits et al. (2005) use a static points-to analysis to resolve reflective method calls in Java by tracking string values. Christensen et al. (2003) propose a general string analysis they use to resolve reflection and check the syntax of SQL queries, among other applications. Gould et al. (2004) also propose a static string analysis to check database queries, and several proposed systems use partial evaluation to resolve reflection and other dynamic constructs (Braux and Noyé 2000; Thiemann 1996). The main disadvantage of all of these approaches is that they rely purely on static analysis. Indeed, Sawin and Rountev (2007) observe that pure static analysis of strings is unable to resolve many dynamic class loading sites in Java. They propose solving this problem using a semi-static analysis, where partial information is gathered dynamically and then static analysis computes the rest. In $\mathcal{P}$Ruby, we opted to use a pure dynamic analysis to track highly dynamic features, to keep $\mathcal{P}$Ruby as simple and predictable as possible.

Chugh et al. (2009) present a hybrid approach to information flow in Javascript that computes as much of the flow graph as possible statically, and performs only residual checks at run time when new code becomes available. In Ruby, we found that the effects of dynamic features must be available during static analysis, to ensure that all defined methods are known to the type checker. Our runtime instrumentation for blame tracking is similar to a proposed system for tracking NULL values in C (Bond et al. 2007). One difference is that we must check for and allow type-correct methods at runtime, whereas NULL supports no operations.

Finally, there is an extensive body of work on performing static analysis for optimization of Java. A major challenge is handling both dynamic class loading and reflection. Jax (Tip et al. 1999) uses programmer specifications to ensure safe modeling of reflective calls. Sreedhar et al. (2000) describe a technique for ahead-of-time optimization of parts of a Java program that are guaranteed unaffected by dynamic class loading. Pechtchanski and Sarkar (2001) present a Java optimization system that reanalyzes code on seeing any dynamic events that would invalidate prior analysis. Hirzel et al. (2004) develop an online pointer analysis that tracks reflective method calls and can analyze classes as they are dynamically loaded. All of these systems are concerned with optimizing a program, whereas in contrast, $\mathcal{P}$Ruby extracts run-time profiling information to guide compile-time (user-level) type inference.

9. Conclusion

We have presented $\mathcal{P}$Ruby, a profile-guided type inference system for Ruby. $\mathcal{P}$Ruby is built on top of DRuby, which performs purely static type inference on Ruby. $\mathcal{P}$Ruby works by first instrumenting source programs to gather profiles that record how dynamic constructs are used by the program. These profiles then guide a transformation phase that replaces dynamic constructs with static constructs specialized to the values seen at run time. We have proven that our technique is sound for TinyRuby, a small Ruby-like calculus with dynamic features. We evaluated $\mathcal{P}$Ruby on a suite of Ruby programs, and we found that use of dynamic features is pervasive throughout our benchmarks, but that, nevertheless, most uses of these features are essentially static, and hence can be profiled. We also discovered a number of type errors in our benchmarks and found that, modulo these errors, our benchmarks can be made mostly typable by applying a number of refactorings. We believe our results show that using profiles to enhance static analysis is a promising technique for analyzing programs written in highly dynamic scripting languages.

Acknowledgments

We wish to thank Michael Hicks and the anonymous reviewers for their helpful comments on this paper. This research was supported in part by DARPA ODOD.HR00110810073.

References

M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM TOPLAS*, 13(2):237–268, 1991.

Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas Matsakis. RPython: Reconciling Dynamically and Statically Typed OO Languages. In *DLS*, 2007.

Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards Type Inference for JavaScript. In *ECOOP*, pages 428–452, 2005.

John Aycock. Aggressive Type Inference. In *Proceedings of the 8th International Python Conference*, pages 11–20, 2000.

M.D. Bond, N. Nethercote, S.W. Kent, S.Z. Guyer, and K.S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *Proceedings of the 2007 OOPSLA conference*, pages 405–422. ACM New York, NY, USA, 2007.

M. Braux and J. Noyé. Towards partially evaluating reflection in Java. In *PEPM*, pages 2–11, 2000.

Brett Cannon. Localized Type Inference of Atomic Types in Python. Master's thesis, California Polytechnic State University, San Luis Obispo, 2005.

Robert Cartwright and Mike Fagan. Soft typing. In *PLDI*, pages 278–292, 1991.

Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise Analysis of String Expressions. In *SAS*, pages 1–18, 2003.

Ravi Chugh, Jeff Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, June 2009. To appear.

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

Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. Technical Report CS-TR-4935, University of Maryland, 2009a. `http://www.cs.umd.edu/projects/PL/druby`.

Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. The Ruby Intermediate Language. In *Dynamic Language Symposium*, Orlando, Florida, October 2009b.

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

Carl Gould, Zhendong Su, and Premkumar Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *ICSE*, pages 645–654, 2004.

Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In *PLDI*, pages 136–150, 1990.

J. Gronski, K. Knowles, A. Tomb, S.N. Freund, and C. Flanagan. Sage: Hybrid Checking for Flexible Specifications. *Scheme and Functional Programming*, 2006.

Lars T Hansen. Evolutionary Programming and Gradual Typing in ECMAScript 4 (Tutorial), November 2007.

D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Trends in Functional Programming*, 2007.

M. Hirzel, A. Diwan, and M. Hind. Pointer Analysis in the Presence of Dynamic Class Loading. In *ECOOP*, 2004.

Kristian Kristensen. Ecstatic – Type Inference for Ruby Using the Cartesian Product Algorithm. Master's thesis, Aalborg University, 2007.

Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection Analysis for Java. In *ASPLS*, 2005.

I. Pechtchanski and V. Sarkar. Dynamic optimistic interprocedural analysis: a framework and an application. In *OOPSLA*, pages 195–210, 2001.

Michael Salib. Starkiller: A Static Type Inferencer and Compiler for Python. Master's thesis, MIT, 2004.

Jason Sawin and Atanas Rountev. Improved static resolution of dynamic class loading in Java. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 143–154, 2007.

Jeremy Siek and Walid Taha. Gradual typing for objects. In *ECOOP*, pages 2–27, 2007.

Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.

V.C. Sreedhar, M. Burke, and J.D. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *PLDI*, pages 196–207, 2000.

Strongtalk. Strongtalk, 2008. `http://www.strongtalk.org/`.

Satish Thatte. Quasi-static typing. In *POPL*, pages 367–381, 1990.

Peter Thiemann. Towards partial evaluation of full scheme. In *Reflection 96*, pages 95–106, 1996.

Peter Thiemann. Towards a type system for analyzing javascript programs. In *ESOP*, pages 408–422, 2005.

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

F. Tip, C. Laffra, P.F. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. In *OOPSLA*, pages 292–305, 1999.

Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *POPL*, pages 395–406, 2008.

David A. Wheeler. Sloccount, 2008. `http://www.dwheeler.com/sloccount/`.

Roel Wuyts. RoelTyper, May 2007. `http://decomp.ulb.ac.be/roelwuyts/smalltalk/roeltyper/`.