Dynamic Inference of Static Types for Ruby

Jong-hoon (David) An[†] rockalizer@gmail.com
Avik Chaudhuri[‡] achaudhu@adobe.com
Jeffrey S. Foster[⋆] jfoster@cs.umd.edu
Michael Hicks[⋆] mwh@cs.umd.edu

[†]Epic Systems Corporation [‡]Advanced Technology Labs, Adobe Systems [⋆]University of Maryland, College Park

Abstract

There have been several efforts to bring static type inference to object-oriented dynamic languages such as Ruby, Python, and Perl. In our experience, however, such type inference systems are extremely difficult to develop, because dynamic languages are typically complex, poorly specified, and include features, such as `eval` and reflection, that are hard to analyze.

In this paper, we introduce *constraint-based dynamic type inference*, a technique that infers static types based on dynamic program executions. In our approach, we wrap each run-time value to associate it with a type variable, and the wrapper generates constraints on this type variable when the wrapped value is used. This technique avoids many of the often overly conservative approximations of static tools, as constraints are generated based on how values are used during actual program runs. Using wrappers is also easy to implement, since we need only write a constraint resolution algorithm and a transformation to introduce the wrappers. The best part is that we can eat our cake, too: our algorithm will infer *sound* types as long as it observes every path through each method body—note that the number of such paths may be dramatically smaller than the number of paths through the program as a whole.

We have developed Rubydust, an implementation of our algorithm for Ruby. Rubydust takes advantage of Ruby's dynamic features to implement wrappers as a language library. We applied Rubydust to a number of small programs and found it to be both easy to use and useful: Rubydust discovered 1 real type error, and all other inferred types were correct and readable.

Categories and Subject Descriptors F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program analysis; F.3.3 [*Logics and Meaning of Programs*]: Studies of Program Constructs—Type structure

General Terms Languages, Theory, Verification

Keywords Dynamic type inference, static types, Ruby, dynamic languages

1. Introduction

Over the years, there have been several efforts to bring static type inference to object-oriented dynamic languages such as Ruby, Python, and Perl [2–5, 7, 10, 12, 16, 18, 24, 27]. Static type inference has the potential to provide the benefits of static typing—well-typed programs don't go wrong [17]—without the annotation burden of pure type checking.

However, based on our own experience, developing a static type inference system for a dynamic language is extremely difficult. Most dynamic languages have poorly documented, complex syntax and semantics that must be carefully reverse-engineered before any static analysis is possible. Dynamic languages are usually "specified" only with a canonical implementation, and tend to have many obscure corner cases that make the reverse engineering process tedious and error-prone. Moreover, dynamic language programmers often employ programming idioms that impede precise yet sound static analysis. For example, programmers often give variables flow-sensitive types that differ along different paths, or add or remove methods from classes at run-time using dynamic features such as reflection and `eval`. Combined, these challenges make developing and maintaining a static type inference tool for a dynamic language a daunting prospect.

To address these problems, this paper introduces *constraint-based dynamic type inference*, a technique to infer static types using information gathered from dynamic runs. More precisely, at run time we introduce type variables for each position whose type we want to infer—specifically fields, method arguments, and return values. As values are passed to those positions, we *wrap* them in a proxy object that records the associated type variable. The user may also supply trusted type annotations for methods. When wrapped values are used as receivers or passed to type-annotated methods, we generate *subtyping constraints* on those variables. At the end of the run, we solve the constraints to find a valid typing, if one exists.

We have implemented this technique for Ruby, as a tool called Rubydust (where "dust" stands for dynamic unraveling of static types). Unlike standard static type systems, Rubydust only conflates type information at method boundaries (where type variables accumulate constraints from different calls), and not within a method. As such, Rubydust supports flow-sensitive treatment of local variables, allowing them to be assigned values having different types. Since Rubydust only sees actual runs of the program, it is naturally path-sensitive and supports use of many dynamic features. Finally, Rubydust can be implemented as a library by employing common introspection features to intercept method calls; there is no need to reverse-engineer any subtle parsing or elaboration rules and to build a separate analysis infrastructure. In sum, Rubydust is more precise than standard static type inference, and sidesteps many of the engineering difficulties of building a static analysis tool suite.

Although Rubydust is based purely on dynamic runs, we can still prove a soundness theorem. We formalized our algorithm on a core subset of Ruby, and we prove that if the training runs on which types are inferred cover every path in the control-flow graph (CFG) of every method of a class, then the inferred types for that class's fields and methods are sound for all possible runs. In our formalism, all looping occurs through recursion, and so the number of required

POPL'11, January 26–28, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0490-0/11/01…$10.00

paths is at most exponential in the size of the largest method body in a program. Notice that this can be dramatically smaller than the number of paths through the program as whole.

Clearly, in practice it is potentially an issue that we need test cases that cover all method paths for fully sound types. However, there are several factors that mitigate this potential drawback.

• Almost all software projects include test cases, and those test cases can be used for training. In fact, the Ruby community encourages test-driven development, which prescribes that tests be written before writing program code—thus tests will likely be available for Rubydust training right from the start.

• While loops within methods could theoretically yield an unbounded number of paths, in our experimental benchmarks we observed that most loop bodies use objects in a type-consistent manner within each path within the loop body. Hence, typically, observing all paths within a loop body (rather that observing all possible iterations of a loop) suffices to find correct types. We discuss this more in Section 4.

• Even incomplete tests may produce useful types. In particular, the inferred types will be sound for any execution that takes (within a method) paths that were covered in training. We could potentially add instrumentation to identify when the program executes a path not covered by training, and then blame the lack of coverage if an error arises as a result [10]. Types are also useful as documentation. Currently, the Ruby documentation includes informal type signatures for standard library methods, but those types could become out of sync with the code (we have found cases of this previously [12]). Using Rubydust, we could generate type annotations automatically from code using its test suite, and thus keep the type documentation in-sync with the tested program behaviors.

Our implementation of Rubydust is a Ruby library that takes advantage of Ruby's rich introspection features; no special tools or compilers are needed. Rubydust wraps each object o at runtime with a proxy object that associates o with a type variable α that corresponds to o's position in the program (a field, argument, or return-value). Method calls on o precipitate the generation of constraints; e.g., if the program invokes o.m(x) then we generate a constraint indicating that α must have a method m whose argument type is a supertype of the type of x. Rubydust also consumes trusted type annotations on methods; this is important for giving types to Ruby's built-in standard library, which is written in C rather than Ruby and hence is not subject to our type inference algorithm.

We evaluated Rubydust by applying it to several small programs, the largest of which was roughly 750 LOC, and used their accompanying test suites to infer types. We found one real type error, which is particularly interesting because the error was uncovered by solving constraints from a passing test run. All other programs were found to be type correct, with readable and correct types. The overhead of running Rubydust is currently quite high, but we believe it can be reduced with various optimizations that we intend to implement in the future. In general we found the performance acceptable and the tool itself quite easy to use.

In summary, our contributions are as follows:

- We introduce a novel algorithm to infer types at run-time by dynamically associating fields and method arguments and results with type variables, and generating subtyping constraints as those entities are used. (Section 2)
- We formalize our algorithm and prove that if training runs cover all syntactic paths through each method of a class, then the inferred type for that class is sound. (Section 3)
- We describe Rubydust, a practical implementation of our algorithm that uses Ruby's rich introspection features. Since Rubydust piggybacks on the standard Ruby interpreter, we can naturally handle all of Ruby's rather complex syntax and semantics without undue effort. (Section 4)
- We evaluate Rubydust on a small set of benchmarks and find it to be useful. (Section 5)

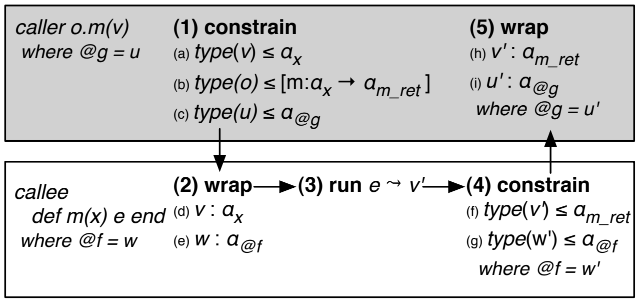

Figure 1. Dynamic instrumentation for a call o.m(v)

We believe that Rubydust is a practical, effective method for inferring useful static types in Ruby, and that the ideas of Rubydust can be applied to other dynamic languages.

2. Overview

Before presenting our constraint-based dynamic type inference algorithm formally, we describe the algorithm by example and illustrate some of its key features. Our examples below are written in Ruby, which is a dynamically typed, object-oriented language inspired by Smalltalk and Perl. In our discussion, we will try to point out any unusual syntax or language features we use; more complete information about Ruby can be found elsewhere [28].

2.1 Method call and return

In our algorithm, there are two kinds of classes: *annotated classes*, which have trusted type signatures, and *unannotated classes*, whose types we wish to infer. We assign a type variable to each field, method argument, and method return value in every unannotated class. At run time, values that occupy these positions are associated with the corresponding type variable. We call this association *wrapping* since we literally implement it by wrapping the value with some metadata. When a wrapped value is used, e.g., as a receiver of a method call, or as an argument to a method with a trusted type signature, we generate a subtyping constraint on the associated variable. At the end of the training runs, we solve the generated constraints to find solutions for those type variables (which yield field and method types for unannotated classes), or we report an error if no solution exists. Note that since all instances of a class share the same type variables, use of any instance contributes to inferring a single type for its class.

Before working through a full example, we consider the operation of our algorithm on a single call to an unannotated method. Figure 1 summarizes the five steps in analyzing a call o.m(v) to a method defined as **def** m(x) e **end**, where x is the formal argument and e is the method body. In this case, we create two type variables: α_x, to represent x's type, and α_{m_ret}, to represent m's return type.

In step (1), the caller looks up the (dynamic) class of the receiver to find the type of the called method. In this case, method m has type $\alpha_x \rightarrow \alpha_{m\_ret}$. The caller then generates two constraints. The constraint labeled (a) ensures the type of the actual argument is a subtype of the formal argument type. Here, $\mathtt{type}(x)$ is the type of an object, either its actual type, for an unwrapped object, or the type variable stored in the wrapper. In the constraint (b), the type $[m : \ldots]$ is the type of an object with a method m with the

```
1  # Numeric : [...+ : Numeric → Numeric...]
2
3  class A
4    # foo : $\alpha_w \times \alpha_u \to \alpha_{foo\_ret}$
5    def foo(w,u)        # w = $(b : \alpha_w)$, u = $(1 : \alpha_u)$
6      w.baz()           #                   $\alpha_w \leq$ [baz : () → ()]
7      y = 3 + u         # y = (4 : Numeric) $\alpha_u \leq$ Numeric
8      return bar(w)     # ret = $(7 : \alpha_{bar\_ret})$  $\alpha_w \leq \alpha_x$
9    end                 #                   $\alpha_{bar\_ret} \leq \alpha_{foo\_ret}$
10   # bar : $\alpha_x \to \alpha_{bar\_ret}$
11   def bar(x)          # x = $(b : \alpha_x)$
12     x.qux()           #                   $\alpha_x \leq$ [qux : () → ()]
13     return 7          #                   Numeric $\leq \alpha_{bar\_ret}$
14   end
15 end
16
17 A.new.foo(B.new,1)     # B.new returns a new object b
18                        # B $\leq \alpha_w$
19                        # Numeric $\leq \alpha_u$
20                        # ret = $(7 : \alpha_{foo\_ret})$
```

Figure 2. Basic method call and return example

given type. Hence by width-subtyping, constraint (b) specifies that o has at least an m method with the appropriate type. We generate this constraint to ensure o's static type type(o) is consistent with the type for m we found via dynamic lookup. For now, ignore the constraint (c) and the other constraints (e), (g), and (i) involving fields @f and @g; we will discuss these in Section 2.3.

In step (2) of analyzing the call, the callee wraps its arguments with the appropriate type variables immediately upon entry. In this case, we set x to be $v : \alpha_x$, which is our notation for the value v wrapped with type α_x.

Then in step (3), we execute the body of the method. Doing so will result in calls to other methods, which will undergo the same process. Moreover, as $v : \alpha_x$ may be used in some of these calls, we will generate constraints on type($v : \alpha_x$), i.e., α_x, that we saw in step (1). In particular, if $v : \alpha_x$ is used as a receiver, we will constrain α_x to have the called method; if $v : \alpha_x$ is used as an argument, we will constrain it to be a subtype of the target method's formal argument type.

At a high-level, steps (1) and (2) maintain two critical invariants:

- Prior to leaving method n to enter another method m, we generate constraints to capture the flow of values from n to m (Constraints (a) and (b)).
- Prior to entering a method m, all values that could affect the type of m are wrapped (Indicated by (d)).

Roughly speaking, constraining something with a type records how it was used in the past, and wrapping something with a type observes how it is used in the future.

Returning from methods should maintain the same invariants as above but in the reverse direction, from callee to caller. Thus, in step (4), we generate constraint (f) in the callee that the type of the returned value is a subtype of the return type, and in step (5), when we return to the caller we immediately wrap the returned value with the called method's return type variable.

2.2 Complete example

Now that we have seen the core algorithm, we can work through a complete example. Consider the code in Figure 2, which defines a class A with two methods foo and bar, and then calls foo on a fresh instance of A on line 17.

This code uses Ruby's Numeric class, which is one of the built-in classes for integers. Because Numeric is built-in, we make it an annotated class, and supply trusted type signatures for all of its methods. A portion of the signature is shown on line 1, which indicates Numeric has a method + of type Numeric→Numeric.[1]

As in the previous subsection, we introduce type variables for method arguments and returns, in this case α_w, α_u, and α_{foo_ret} for foo, and α_x and α_{bar_ret} for bar. Then we begin stepping through the calls. At the call on line 17, we pass in actual arguments b (the object created by the call to B.new on the same line) and 1. Thus we constrain the formal argument types in the caller (lines 18 and 19) and wrap the actuals in the callee (line 5).

Next, on line 6, we use a wrapped object, so we generate a constraint; here we require the associated type variable α_w contains a no-argument method baz. (For simplicity we show the return type as (), though as it is unused it could be arbitrary.)

On line 7, we call 3 + u. The receiver object is 3, which has actual class is Numeric, an annotated class. Thus, we do the normal handling in the caller (the shaded box in Figure 1), but omit the steps in the callee, since the annotation is trusted. Here, we generate a constraint $\alpha_u \leq$ Numeric between the actual and formal argument types. We also generate a constraint (not shown in the figure) type(3) $\leq [+ : \ldots]$, but as type(3) = Numeric, this constraint is immediately satisfiable; in general, we need to generate such constraints to correctly handle cases where the receiver is not a constant. Finally, we wrap the return value from the caller with its annotated type Numeric.

Next, we call method bar. As expected, we constrain the actuals and formals (line 8), wrap the argument inside the callee (line 11), and generate constraints during execution of the body (line 12). At the return from bar, we constrain the return type (line 13) and wrap in the caller (yielding the wrapped value $7 : \alpha_{bar\_ret}$ on line 8). As that value is immediately returned by foo, we constrain the return type of foo with it (line 9) and wrap in the caller (line 20).

After this run, we can solve the generated constraints to infer types. Drawing the constraints from the example as a directed graph, where an edge from x to y corresponds to the constraint $x \leq y$, we have:

$$\begin{tikzcd}
 & & {[baz : () \to ()]} & \text{Numeric} \arrow[r] & \alpha_u \arrow[r] & \text{Numeric} \\
B \arrow[r] & \alpha_w \arrow[ur] \arrow[dr] & & & & \\
 & & \alpha_x \arrow[r] & {[qux : () \to ()]} & \text{Numeric} \arrow[r] & \alpha_{bar\_ret} \arrow[r] & \alpha_{foo\_ret}
\end{tikzcd}$$

(Here we duplicated Numeric for clarity; in practice, it is typically represented by a single node in the graph.) As is standard, we wish to find the *least solution* (equivalent to a most general type) for each method. Since arguments are contravariant and returns are covariant, this corresponds to finding *upper bounds* (transitive successors in the constraint graph) for arguments and *lower bounds* (transitive predecessors) for returns. Intuitively, this is equivalent to inferring argument types based on how arguments are used within a method, and computing return types based on what types flow to return positions. For our example, the final solution is

$$\alpha_w = [\text{baz} : () \to (), \text{qux} : () \to ()] \qquad \alpha_x = [\text{qux} : () \to ()]$$

$$\alpha_u = \text{Numeric} \qquad \alpha_{bar\_ret} = \alpha_{foo\_ret} = \text{Numeric}$$

Notice that w must have bar and qux methods, but x only needs a qux method. For return types, both bar and foo always return Numeric.

2.3 Local variables and fields

In the previous example, our algorithm generated roughly the same constraints that a static type inference system might generate. How-

[1] In Ruby, the syntax $e_1 + e_2$ is shorthand for $e_1.+(e_2)$, i.e., calling the + method on e_1 with argument e_2.

```
21 class C
22   def foo(x)
23     z = x;  z.baz();
24     z = 3; return z + 5;
25   end
26 end
27 class D
28   def bar(x)  @f = x       end
29   def baz()   y = 3 + @f end
30   def qux()   @f = "foo"  end
31   def f() bar("foo") end
32 end
```

Figure 3. Example with local variables and fields

ever, because our algorithm observes only dynamic runs, in many cases it can be more precise than static type inference.

Consider class C in Figure 3. On entry to foo, we wrap the actual argument v as $v : \alpha_x$, where α_x is foo's formal argument type. At the assignment on line 23, we do nothing special—we allow the language interpreter to copy a reference to $v : \alpha_x$ into z. At the call to baz, we generate the expected constraint $\alpha_x \leq [\mathsf{baz} : () \to ()]$. More interestingly, on line 24, we reassign z to contain 3, and so at the call to z's $+$ method, we do not generate any constraints on α_x. Thus, our analysis is *flow-sensitive* with respect to local variables, meaning it respects the order of operations within a method.

To see why this treatment of local variable assignment is safe, it helps to think in terms of compiler optimization. We are essentially performing inference over a series of execution traces. We can view each trace as a straight-line program. Consider the execution trace of foo (which is the same as the body of the function, in this case). If we apply copy propagation (of x and 3) to the trace we get `z=x; x.baz(); z=3; return 3 + 5;` Since z is a local variable inaccessible outside of the scope of foo, it is dead at the end of the method, too, so we can apply dead code elimination to reduce the trace to "`x.baz(); return 3+5;`". The constraints we would generate from this trace are equivalent to those we would generate with our approach.

Instance fields in Ruby are not visible outside an object, but they are shared across all methods of an object. Thus, we need to treat them differently than locals. To see why, consider the class D in Figure 3, which uses the instance variable @f (all instance variables begin with @ in Ruby). Suppose that we treated fields the same way as local variables, i.e., we did nothing special at assignments to them. Now consider inferring types for D with the run `bar(1); baz(); qux()`. During the call bar(1), we would generate the constraint $\mathsf{Numeric} \leq \alpha_x$ (the type variable for x) and store $1 : \alpha_x$ in @f. Then during baz(), we would generate the constraint $\alpha_x \leq \mathsf{Numeric}$, and the call to qux() would generate no constraints. Thus, we could solve the constraints to get $\alpha_x = \mathsf{Numeric}$, and we would think this class has type $[\mathsf{bar} : \mathsf{Numeric} \to (), \mathsf{baz} : () \to (), \mathsf{qux} : () \to ()]$. But this result is clearly wrong, as the "well-typed" sequence `qux(); baz()` produces a type error.

To solve this problem, we need to introduce a type variable $\alpha_{@f}$ for the field, and then generate constraints and wrap values accordingly. It would be natural to do this at writes to fields, but that turns out to be impossible with a Ruby-only, dynamic solution, as there is no dynamic mechanism for intercepting field writes.[2] Fortunately, we can still handle fields safely by applying the same

[2] Recall we wish to avoid using static techniques, including program rewriting, because they require complex front-ends that understand program semantics. For example, even ordering of assignment operations in Ruby can be non-obvious in some cases [11].

```
33 class E
34   def foo(x, p)
35     if p then x.qux() else x.baz() end
36   end
37   def bar(p)
38     if p then y = 3 else y = "hello" end
39     if p then y + 6 else y.length end
40   end
41 end
```

(a) Paths and path-sensitivity

```
42 class F
43   def foo(x)
44     return (if x then 0 else "hello" end)
45   end
46   def bar(y,z)
47     return (if y then foo(z) else foo(!z) end)
48   end
49 end
50 f = F.new
```

(b) Per-method path coverage

Figure 4. Additional Examples

principles we saw in Figure 1 for method arguments and returns. There, we needed two invariants: (1) when we switch from method m to method n, we need to capture the flow of values from m to n, and (2) when we enter a method n, we need to wrap all values that could affect the type of n. Translating this idea to fields, we need to ensure:

- When we switch from m to n, we record all field *writes* performed by m, since they might be read by n. This is captured by constraints (c) and (g) in Figure 1.
- When we enter n, we need to wrap all fields n may use, so that subsequent field *reads* will see the wrapped values. This is captured by constraints (e) and (i) in Figure 1.

Adding these extra constraints and wrapping operations solves the problem we saw above. At the call bar(1), we generate the constraint $\mathsf{Numeric} \leq \alpha_x$, as before. However, at the end of bar, we now generate a constraint $\alpha_x \leq \alpha_{@f}$ to capture the write. At the beginning of baz, we wrap @f so that the body of baz will now generate the constraints $\alpha_{@f} \leq \mathsf{Numeric}$. Then qux generates the constraint $\mathsf{String} \leq \alpha_{@f}$. We can immediately see the constraints $\mathsf{String} \leq \alpha_{@f} \leq \mathsf{Numeric}$ are unsatisfiable, and hence we would correctly report a type error.

Notice that this design also allows a small amount of flow-sensitivity for fields: A field may temporarily contain a value of a different type, as long as it is overwritten before calling another method or returning. We do not expect that this yields much additional precision in practice, however.

Our implementation handles class variables similarly to instance variables. Note that we assume single-threaded execution—if accesses of such variables by other threads are interleaved, the inferred constraints may be unsound.

2.4 Path, path-sensitivity, and path coverage

As our algorithm observes dynamic runs, to infer sound types we need to observe all possible paths through branching code sequences. For example, we can infer types for the foo method in Figure 4(a) if we see an execution such as `foo(a, true); foo(b, false);`.

In this case, we will generate $\alpha_x \leq$ [qux : ...] from the first call, and $\alpha_x \leq$ [baz : ...] from the second.

One benefit of observing actual dynamic runs is that we never model unrealizable program executions. For example, consider the bar method in Figure 4(a). In a call bar(**true**), line 38 assigns a Numeric to y, and in a call bar(**false**), it assigns a String to y. Typical path-insensitive static type inference would conflate these possibilities and determine that y could be either a Numeric or String on line 39, and hence would signal a potential error for both the calls to + and to length. In contrast, in our approach we do not assign any type to local y, and we observe each path separately. Thus, we do not report a type error for this code.

Our soundness theorem (Section 3) holds if we observe all possible paths within each method body. To see why this is sufficient, rather than needing to observe all possible *program* paths, consider the code in Figure 4(b). Assuming bar is the entry point, there are four paths through this class, given by all possible truth value combinations for y and z. However, to observe all possible *types*, we only need to explore two paths. If we call f.bar(**true**, **true**) and f.bar(**false** , **true**), we will generate the following constraints:[3]

f.bar(true, true)	f.bar(false, true)
Boolean $\leq \alpha_y$	Boolean $\leq \alpha_y$
Boolean $\leq \alpha_z$	Boolean $\leq \alpha_z$
$\alpha_z \leq \alpha_x$	Boolean $\leq \alpha_x$
Numeric $\leq \alpha_{foo\_ret}$	String $\leq \alpha_{foo\_ret}$
$\alpha_{foo\_ret} \leq \alpha_{bar\_ret}$	$\alpha_{foo\_ret} \leq \alpha_{bar\_ret}$

Thus, we can deduce that bar may return a Numeric or a String. (Note that our system supports union types, so we can express this return type as String $\cup$ Numeric.)

The reason we only needed two paths is that type variables on method arguments and returns act as join points, summarizing the possible types of all paths within a method. In our example, both branches of the conditional in bar have the same type, α_{foo_ret}. Thus, the other possible calls, f.bar(**true**, **false**) and f.bar(**false** , **false**), do not affect what types bar could return.

2.5 Dynamic features

Another benefit of our dynamic type inference algorithm is that we can easily handle dynamic features that are very challenging for static type inference. For example, consider the following code, which occurred in one of our experimental benchmarks:

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

This constructor takes a hash args as an argument, and then for each key-value pair (k, v) uses reflective method invocation via send to call a method named after k with the argument v. Or, consider the following code:

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

For each element of the string array ATTRIBUTES, this code uses eval to define a new method named after the element.

[3] Note that an **if** generates no constraints on its guard as any object may be supplied (**false** and **nil** are false, and any other object is true). Also, here and in our implementation, we treat **true** and **false** as having type Boolean, though in Ruby they are actually instances of TrueClass and FalseClass , respectively.

$$\begin{array}{rrcl}
\text{expressions} & e & ::= & \mathsf{nil} \mid \mathsf{self} \mid x \mid @f \mid x = e \\
 & & \mid & @f = e \mid A.\mathsf{new} \mid e.m(e') \mid e; e' \\
 & & \mid & \mathsf{if}^{\ell}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' \\
\text{methods} & d & ::= & \mathsf{def}\ m(x) = e \\
\text{classes} & c & ::= & \mathsf{class}\ A = d^{\star} \\
\text{programs} & \mathcal{P} & ::= & c^{\star} \triangleright e \\
\\
\text{types} & \tau & ::= & A.@f \mid A.m \mid A.\overline{m} \mid \bot \mid \top \\
 & & \mid & A \mid [m : A.m \rightarrow A.\overline{m}] \mid \tau \cup \tau' \mid \tau \cap \tau'
\end{array}$$

$$\begin{array}{rcl@{\qquad}rcl}
x & \in & \text{local variables} & A & \in & \text{class names} \\
@f & \in & \text{field names} & \ell & \in & \text{labels} \\
m & \in & \text{method names} & & &
\end{array}$$

Figure 5. Syntax of source language

We encountered both of these code snippets in earlier work, in which we proposed using run-time profiling to gather concrete uses of send, eval, and other highly dynamic constructs, and then analyzing the profile data statically [10]. In the dynamic analysis we propose in the current paper, no separate profiling pass is needed: we simply let the language interpreter execute this code and observe the results during type inference. Method invocation via send is no harder than normal dynamic dispatch; we just do the usual constraint generation and wrapping, which, as mentioned earlier, is actually performed inside the callee in our implementation. Method creation via eval is also easy, since we add wrapping instrumentation by dynamically iterating through the defined methods of unannotated classes; it makes no difference how those methods were created, as long as it happens before instrumentation.

3. Formal development

In this section, we describe our dynamic type inference technique on a core Ruby-like source language.

The syntax is shown in Figure 5. Expressions include nil, self, variables x, fields $@f$, variable assignments $x = e$, field assignments $@f = e$, object creations A.new, method calls $e.m(e')$, sequences $e; e'$, and conditionals if$^{\ell}$ e then e' else e''. Conditionals carry labels ℓ to allow us to reason about path coverage. Conceptually, labels may be thought of as logical propositions—literals of the form ℓ and $\neg\ell$ represent a branch taken or not taken, respectively, at run time. We assume that all labels are distinct.

The form def $m(x) = e$ defines a method m with formal argument x and body e. Classes c are named collections of methods. A program consists of a set of classes and a single expression that serves as a *test*. Typically, we run a test on a collection of classes to "train" the system—i.e., infer types—and, in our proofs, run other tests to "monitor" the system—i.e., show that the inferred types are sound. In the formalism, we use only a single test e to train the system, but we can always represent a set of tests by sequencing them together into a single expression.

The syntax of types requires some explanation. Type variables are "tagged" to avoid generating and accounting for fresh variables. Thus, $A.@f$ is a type variable that denotes the type of the field $@f$ of objects of class A; similarly, $A.m$ and $A.\overline{m}$ are type variables that denote the argument and result types of the method m of class A. In addition, we have nominal types A for objects of class A, and structural types $[m : A.m \rightarrow A.\overline{m}]$ for objects with method m whose argument and result types can be viewed as $A.m$ and $A.\overline{m}$. Finally, we have the bottom type $\bot$, the top type $\top$, union types $\tau \cup \tau'$, and intersection types $\tau \cap \tau'$.

$$
\begin{array}{rcl}
\text{values } v & ::= & l \mid \mathsf{nil} \\
\text{wrapped values } \omega & ::= & v : \tau \\
\text{field maps } \mathcal{F} & ::= & (@f \mapsto \omega)^\star \\
\text{method maps } \mathcal{M} & ::= & (m \mapsto \lambda(x)e)^\star \\
\text{class maps } \mathcal{C} & ::= & (A \mapsto \mathcal{M})^\star \\
\text{heaps } \mathcal{H} & ::= & (l \mapsto A\langle\mathcal{F}\rangle)^\star \\
\text{environments } \mathcal{E} & ::= & (x \mapsto \omega)^\star, (\mathsf{self} \mapsto l : A)^? \\
\text{literals } p & ::= & (\ell \mid \neg\ell) \\
\text{paths } \phi & ::= & p^\star \\
\text{constraints } \Pi & ::= & (\tau \leq \tau')^\star, \phi^\star
\end{array}
$$

Figure 6. Auxiliary syntax

3.1 Training semantics

We now define a semantics for training. The semantics extends a standard semantics with some instrumentation. The instrumentation does not affect the run-time behavior of programs; it merely records run-time information that later allows us to infer types and argue about their soundness.

To define the semantics, we need some auxiliary syntax to describe internal data structures, shown in Figure 6. Let l denote heap locations. Values include heap locations and nil. Such values are *wrapped* with types for training. A field map associates field names with wrapped values. A method map associates method names with abstractions. A class map associates class names with method maps. A heap maps locations to objects $A\langle\mathcal{F}\rangle$, which denote objects of class A with field map $\mathcal{F}$. An environment maps variables to wrapped values and, optionally, self to a location wrapped with its run-time type. Formally, the run-time type of a value under a heap is defined by $\mathtt{runtype}_{\mathcal{H}}(l) = A$ if $\mathcal{H}(l) = A\langle\mathcal{F}\rangle$ and $\mathtt{runtype}_{\mathcal{H}}(\mathsf{nil}) = \bot$.

A path ϕ is a list of label literals ℓ or $\neg\ell$, denoted by p, that mark conditionals encountered in a run. Constraints Π include standard subtyping constraints $\tau \leq \tau'$ and coverage constraints ϕ, meaning that the path ϕ is traversed during some call in the run.

The rules shown in Figure 7 derive big-step reduction judgments of the form $\mathcal{H}; \mathcal{E}; e \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}'; \omega \mid \Pi; \phi$, meaning that given class map $\mathcal{C}$, expression e under heap $\mathcal{H}$ and environment $\mathcal{E}$ reduces to wrapped value ω, covering path ϕ, generating constraints Π, and returning heap $\mathcal{H}'$ and environment $\mathcal{E}'$. We define the following operations on wrapped values: if $\omega = v : \tau$ then $\mathtt{val}(\omega) = v$, $\mathtt{type}(\omega) = \tau$, and $\omega \bullet \tau' = v : \tau'$. In the rules, we use an underscore in any position where an arbitrary quantity is allowed, and we write empty set as $\{\}$ to avoid confusion with ϕ.

By (TNIL), the type assigned to nil is $\bot$, which means that nil may have any type. (TSELF) is straightforward. In (TNEW), the notation $A\langle \_ \mapsto \mathsf{nil} : \bot\rangle$ indicates an instance of A with all possible fields mapped to nil. (As in Ruby, fields need not be explicitly initialized before use, and are nil by default.) (TVAR) and (TVAR=) are standard, and generate no constraint nor perform any wrapping, as discussed in Section 2.3.

As explained in Section 2, we permit some flow-sensitivity for field types. Thus, (FIELD) and (FIELD=) are like (VAR) and (VAR=) in that they generate no constraint and perform no wrapping.

(TSEQ) is straightforward. By (TCOND), either ℓ or $\neg\ell$ is recorded in the current path, depending on the branch traversed for the conditional with label ℓ. Note that ℓ and $\neg\ell$ can never appear in the same path: looping in the formal language occurs only via recursive calls, and callee paths are not included in caller paths. That said, the conditional itself may be visited on many occasions during the training run, so that both ℓ and $\neg\ell$ may eventually appear in coverage constraints.

(TNIL)

$$\mathcal{H}; \mathcal{E}; \mathsf{nil} \longrightarrow_{\mathcal{C}} \mathcal{H}; \mathcal{E}; \mathsf{nil} : \bot \mid \{\}; \{\}$$

(TSELF)

$$\frac{\mathcal{E}(\mathsf{self}) = l : A}{\mathcal{H}; \mathcal{E}; \mathsf{self} \longrightarrow_{\mathcal{C}} \mathcal{H}; \mathcal{E}; l : A \mid \{\}; \{\}}$$

(TNEW)

$$\frac{l \text{ fresh} \qquad \mathcal{H}' = \mathcal{H}\{l \mapsto A\langle \_ \mapsto \mathsf{nil} : \bot\rangle\}}{\mathcal{H}; \mathcal{E}; A.\mathsf{new} \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}; l : A \mid \{\}; \{\}}$$

(TVAR)

$$\frac{\mathcal{E}(x) = \omega}{\mathcal{H}; \mathcal{E}; x \longrightarrow_{\mathcal{C}} \mathcal{H}; \mathcal{E}; \omega \mid \{\}; \{\}}$$

(TVAR =)

$$\frac{\mathcal{H}; \mathcal{E}; e \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}'; \omega \mid \Pi; \phi \qquad \mathcal{E}'' = \mathcal{E}'\{x \mapsto \omega\}}{\mathcal{H}; \mathcal{E}; x = e \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}''; \omega \mid \Pi; \phi}$$

(TFIELD)

$$\frac{\mathcal{E}(\mathsf{self}) = \omega \qquad l = \mathtt{val}(\omega) \qquad \mathcal{H}(l) = \_\langle\mathcal{F}\rangle \qquad \mathcal{F}(@f) = \omega}{\mathcal{H}; \mathcal{E}; @f \longrightarrow_{\mathcal{C}} \mathcal{H}; \mathcal{E}; \omega \mid \{\}; \{\}}$$

(TFIELD =)

$$\frac{\begin{array}{c} \mathcal{H}; \mathcal{E}; e \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}'; \omega \mid \Pi; \phi \\ \mathcal{E}'(\mathsf{self}) = \omega' \qquad l = \mathtt{val}(\omega') \\ \mathcal{H}'(l) = A\langle\mathcal{F}\rangle \qquad \mathcal{H}'' = \mathcal{H}'\{l \mapsto A\langle\mathcal{F}\{@f \mapsto \omega\}\rangle\} \end{array}}{\mathcal{H}; \mathcal{E}; @f = e \longrightarrow_{\mathcal{C}} \mathcal{H}''; \mathcal{E}'; \omega \mid \Pi; \phi}$$

(TSEQ)

$$\frac{\begin{array}{c} \mathcal{H}; \mathcal{E}; e \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}'; \_ \mid \Pi; \phi \\ \mathcal{H}'; \mathcal{E}'; e' \longrightarrow_{\mathcal{C}} \mathcal{H}''; \mathcal{E}''; \omega \mid \Pi'; \phi' \end{array}}{\mathcal{H}; \mathcal{E}; (e; e') \longrightarrow_{\mathcal{C}} \mathcal{H}''; \mathcal{E}''; \omega \mid \Pi, \Pi'; \phi, \phi'}$$

(TCOND)

$$\frac{\begin{array}{c} \mathcal{H}; \mathcal{E}; e \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}'; \omega \mid \Pi; \phi \\ p = \ell, e_p = e' \text{ if } \mathtt{val}(\omega) \neq \mathsf{nil} \\ p = \neg\ell, e_p = e'' \text{ if } \mathtt{val}(\omega) = \mathsf{nil} \\ \mathcal{H}'; \mathcal{E}'; e_p \longrightarrow_{\mathcal{C}} \mathcal{H}''; \mathcal{E}''; \omega' \mid \Pi'; \phi' \end{array}}{\mathcal{H}; \mathcal{E}; \mathsf{if}^{\ell}\ e \ \mathsf{then}\ e' \ \mathsf{else}\ e'' \longrightarrow_{\mathcal{C}} \mathcal{H}''; \mathcal{E}''; \omega' \mid \Pi, \Pi'; \phi, p, \phi'}$$

(TCALL)

$$\frac{\begin{array}{c} \mathcal{H}; \mathcal{E}; e \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}'; \omega \mid \Pi; \phi \qquad \tau = \mathtt{type}(\omega) \\ \mathcal{H}'; \mathcal{E}'; e' \longrightarrow_{\mathcal{C}} \mathcal{H}''; \mathcal{E}''; \omega' \mid \Pi'; \phi' \qquad \tau' = \mathtt{type}(\omega') \\ l = \mathcal{E}''(\mathsf{self}) \qquad \bar{l} = \mathtt{val}(\omega') \\ \mathcal{H}''(\bar{l}) = A\langle\_\rangle \qquad \mathcal{C}(A) = \mathcal{M} \qquad \mathcal{M}(m) = \lambda(x)e'' \\ \Pi'' = \tau' \leq [m : A.m \rightarrow A.\overline{m}], \tau \leq A.m, \mathtt{constrain}_{l}(\mathcal{H}'') \\ \mathcal{H}''' = \mathtt{wrap}_{\bar{l}}(\mathcal{H}'') \qquad \mathcal{E}''' = \{\mathsf{self} \mapsto \omega' \bullet A, x \mapsto \omega \bullet A.m\} \\ \mathcal{H}'''; \mathcal{E}'''; e'' \longrightarrow_{\mathcal{C}} \overline{\mathcal{H}}; \_; \overline{\omega} \mid \overline{\Pi}; \overline{\phi} \qquad \overline{\tau} = \mathtt{type}(\overline{\omega}) \\ \overline{\Pi}' = \overline{\tau} \leq A.\overline{m}, \mathtt{constrain}_{\bar{l}}(\overline{\mathcal{H}}), \overline{\phi} \\ \overline{\mathcal{H}}' = \mathtt{wrap}_{l}(\overline{\mathcal{H}}) \qquad \overline{\omega}' = \overline{\omega} \bullet A.\overline{m} \end{array}}{\mathcal{H}; \mathcal{E}; e'.m(e) \longrightarrow_{\mathcal{C}} \overline{\mathcal{H}}'; \mathcal{E}''; \overline{\omega}' \mid \Pi, \Pi', \Pi'', \overline{\Pi}, \overline{\Pi}'; \phi, \phi'}$$

Figure 7. Training semantics

(TCALL) performs the actions introduced in Figure 1. First, the type of the receiver ω' is constrained to be a subtype of $[m : A.m \rightarrow A.\overline{m}]$, and the type of the argument ω is constrained to be a subtype of $A.m$, the argument type of the callee. We evaluate the body e'' with argument x mapped to $\omega \bullet A.m$, which is the argument wrapped with method argument's type variable. The type of the result $\overline{\omega}$ is constrained to be a subtype of the result type $A.\overline{m}$, and returned wrapped with that type. Furthermore (TCALL) relates the current path to the set of paths stored as coverage constraints. In particular, while ϕ, ϕ' records the current path traversed in the caller, the path $\overline{\phi}$ traversed by the callee is recorded as a coverage constraint in $\overline{\Pi}'$. In addition, (TCALL) involves wrapping and generation of subtyping constraints for fields of the caller and the callee objects. Let $\mathcal{H}(l) = A\langle\mathcal{F}\rangle$. We define

- $\texttt{wrap}_l(\mathcal{H}) = \mathcal{H}\{l \mapsto A\langle\{@f \mapsto \omega \bullet A.@f \mid @f \mapsto \omega \in \mathcal{F}\}\rangle\}$
- $\texttt{constrain}_l(\mathcal{H}) = \{\texttt{type}(\omega) \leq A.@f \mid @f \mapsto \omega \in \mathcal{F}\}$

As discussed in Section 2, we constrain the fields of the caller object and wrap the fields of the callee object before the method call, and symmetrically, constrain the fields of the callee object and wrap the fields of the caller object after the method call.

Finally, the following rule describes training with programs.

(TRAIN)
$$\frac{\mathcal{C} = \texttt{classmap}(c^\star) \qquad \{\}, \{\}, e \longrightarrow_{\mathcal{C}} \_; \_; \_ \mid \Pi; \_}{c^\star \rhd e \uparrow \texttt{solve}(\texttt{subtyping}(\Pi)); \texttt{coverage}(\Pi)}$$

where we define:

$$\begin{aligned}
\texttt{classmap}(c^\star) &= \{A \mapsto \texttt{methodmap}(d^\star) \mid \mathsf{class}\ A = d^\star \in c^\star\} \\
\texttt{methodmap}(d^\star) &= \{m \mapsto \lambda(x)e \mid \mathsf{def}\ m(x) = e \in d^\star\} \\
\texttt{coverage}(\Pi) &= \{\phi \mid \phi \in \Pi\}
\end{aligned}$$

We assume that $\texttt{solve}(\texttt{subtyping}(\Pi))$ externally solves the subtyping constraints in Π to obtain a mapping $\mathcal{T}$ from type variables to concrete types (possibly involving $\top$ and $\bot$, and unions and intersections of nominal types and structural types). We discuss the solving algorithm we use in our implementation in Section 4; however, our technique is agnostic to the choice of algorithm or even to the language of solved types.

The crucial soundness measure for the inferred types is $\Phi = \texttt{coverage}(\Pi)$, which collects all the coverage constraints in the run. The key soundness theorem states that any run whose paths are a subset of Φ must use types that are consistent with Φ. As a corollary, we show that if all possible paths are covered during inference, then all runs are type safe.

3.2 Monitoring semantics

To argue about the soundness of inferred types, next we define a monitoring semantics. This semantics slightly extends a standard semantics with monitoring of calls at the top level, and monitoring of conditionals. This affects the run-time behavior of programs—we enforce a contract between the run-time type of the argument and the inferred argument type for any call at the top level, and require that the monitoring run only traverses paths that have already been traversed by the training run. A top-level call is simply one that calls into any of the classes for which we have inferred types from outside; a call from within a typed class (either to its own method or to one of other typed class) is considered internal.

To define the monitoring semantics, we modify the syntax of expressions, field maps, class maps, and environments as follows.

$$\begin{array}{rcl}
\text{expressions } e & ::= & \ldots \mid \underline{e'.m(e)} \\
\text{class maps } \mathcal{C} & ::= & (A \mapsto \mathcal{M})^\star, \mathcal{T}, \Phi \\
\text{field maps } \mathcal{F} & ::= & (@f \mapsto v)^\star \\
\text{environments } \mathcal{E} & ::= & (x \mapsto v)^\star, (\mathsf{self} \mapsto l)^?
\end{array}$$

Expressions of the form $\underline{e'.m(e)}$ denote method calls at the top level. Class maps additionally carry the solution $\mathcal{T}$ of the subtyping constraints collected during training, and the coverage Φ of the training run. Field maps and environments no longer map to types.

The rules shown in Figure 8 derive big-step reduction judgments of the form $\mathcal{H}; \mathcal{E}; e \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}'; v \mid \phi'$, where ϕ is the prescribed path to be traversed and ϕ' is the suffix of ϕ that remains.

(NIL)
$$\mathcal{H}; \mathcal{E}; \mathsf{nil} \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}; \mathcal{E}; \mathsf{nil} \mid \phi$$

(SELF)
$$\frac{\mathcal{E}(\mathsf{self}) = l}{\mathcal{H}; \mathcal{E}; \mathsf{self} \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}; \mathcal{E}; l \mid \phi}$$

(NEW)
$$\frac{l \text{ fresh} \qquad \mathcal{H}' = \mathcal{H}\{l \mapsto A\langle \_ \mapsto \mathsf{nil}\rangle\}}{\mathcal{H}; \mathcal{E}; A.\mathsf{new} \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}; l \mid \phi}$$

(VAR)
$$\frac{\mathcal{E}(x) = v}{\mathcal{H}; \mathcal{E}; x \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}; \mathcal{E}; v \mid \phi}$$

(VAR =)
$$\frac{\mathcal{H}; \mathcal{E}; e \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}'; v \mid \phi' \qquad \mathcal{E}'' = \mathcal{E}'\{x \mapsto v\}}{\mathcal{H}; \mathcal{E}; x = e \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}''; v \mid \phi'}$$

(FIELD)
$$\frac{\mathcal{E}(\mathsf{self}) = l \qquad \mathcal{H}(l) = \_\langle\mathcal{F}\rangle \qquad \mathcal{F}(@f) = v}{\mathcal{H}; \mathcal{E}; @f \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}; \mathcal{E}; v \mid \phi}$$

(FIELD =)
$$\frac{\begin{array}{c}\mathcal{H}; \mathcal{E}; e \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}'; v \mid \phi' \qquad \mathcal{E}'(\mathsf{self}) = l \\ \mathcal{H}'(l) = A\langle\mathcal{F}\rangle \qquad \mathcal{H}'' = \mathcal{H}'\{l \mapsto A\langle\mathcal{F}\{@f \mapsto v\}\rangle\}\end{array}}{\mathcal{H}; \mathcal{E}; @f = e \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}''; \mathcal{E}'; v \mid \phi'}$$

(SEQ)
$$\frac{\begin{array}{c}\mathcal{H}; \mathcal{E}; e \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}'; \_ \mid \phi' \\ \mathcal{H}'; \mathcal{E}'; e' \mid \phi' \longrightarrow_{\mathcal{C}} \mathcal{H}''; \mathcal{E}''; v \mid \phi''\end{array}}{\mathcal{H}; \mathcal{E}; (e; e') \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}''; \mathcal{E}''; v \mid \phi''}$$

(CALL)
$$\frac{\begin{array}{c}\mathcal{H}; \mathcal{E}; e \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}'; v \mid \phi' \\ \mathcal{H}'; \mathcal{E}'; e' \mid \phi' \longrightarrow_{\mathcal{C}} \mathcal{H}''; \mathcal{E}''; l \mid \phi'' \\ \mathcal{H}''(l) = A\langle\_\rangle \qquad \mathcal{C}(A) = \mathcal{M} \qquad \mathcal{M}(m) = \lambda(x)e'' \\ \overline{\phi} \in \mathcal{C} \qquad \mathcal{H}''; \{\mathsf{self} \mapsto l, x \mapsto v\}; e'' \mid \overline{\phi} \longrightarrow_{\mathcal{C}} \mathcal{H}'''; \_; v' \mid \_\end{array}}{\mathcal{H}; \mathcal{E}; e'.m(e) \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}'''; \mathcal{E}''; v' \mid \phi''}$$

(MCOND)
$$\frac{\begin{array}{c}\mathcal{H}; \mathcal{E}; e \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}'; v \mid p, \phi' \\ p = \ell, e_p = e' \text{ if } v \neq \mathsf{nil} \qquad p = \neg\ell, e_p = e'' \text{ if } v = \mathsf{nil} \\ \mathcal{H}'; \mathcal{E}'; e_p \mid \phi' \longrightarrow_{\mathcal{C}} \mathcal{H}''; \mathcal{E}''; v' \mid \phi''\end{array}}{\mathcal{H}; \mathcal{E}; \mathsf{if}^\ell\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}''; \mathcal{E}''; v' \mid \phi''}$$

(MCALL)
$$\frac{\begin{array}{c}\mathcal{H}; \mathcal{E}; e \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}'; v \mid \phi' \\ \mathcal{H}'; \mathcal{E}'; e' \mid \phi' \longrightarrow_{\mathcal{C}} \mathcal{H}''; \mathcal{E}''; l \mid \phi'' \\ \mathcal{H}''(l) = A\langle\_\rangle \qquad \mathcal{C}(A) = \mathcal{M} \\ \mathcal{M}(m) = \lambda(x)e'' \qquad \texttt{runtype}_{\mathcal{H}''}(v) \leq \mathcal{C}(A.m) \\ \overline{\phi} \in \mathcal{C} \qquad \mathcal{H}''; \{\mathsf{self} \mapsto l, x \mapsto v\}; e'' \mid \overline{\phi} \longrightarrow_{\mathcal{C}} \mathcal{H}'''; \_; v'' \mid \_\end{array}}{\mathcal{H}; \mathcal{E}; \underline{e'.m(e)} \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}'''; \mathcal{E}''; v'' \mid \phi''}$$

Figure 8. Monitoring semantics

(NIL), (SELF), (NEW), (VAR), (FIELD), (VAR=), (FIELD=), and (SEQ) are derived from the corresponding rules for training, simply by erasing types, paths, and constraints. (CALL) is similar, but in addition, it (non-deterministically) picks a path from the coverage of the training run, which is carried by $\mathcal{C}$, and prescribes that the call should traverse that path by including it in the environment under which that call is run.

By (MCOND), a particular branch p of a conditional may be traversed only if the path prescribes that branch, i.e., it begins with p. The branch is then run under a residual path ϕ'.

By (MCALL), a top level call requires that the run-time type of the argument v is a subtype of the argument type of the method, as prescribed by the solution carried by $\mathcal{C}$, which we write as $\mathcal{C}(A.m)$.

The following rule describes monitoring of programs.

(RUN)

$$\frac{\mathcal{C} = \texttt{classmap}(c^\star), \mathcal{T}, \Phi \qquad \{\}, \{\}, \texttt{monitor}(e) \mid \{\} \longrightarrow_{\mathcal{C}} \_; \_; \_ \mid \{\}}{c^\star \rhd e \downarrow}$$

where $\texttt{monitor}(e')$ is e' but with any method call underlined.

Of course, reduction may be stuck in several ways, so we do not expect to derive such a judgment for every program. In particular, we do not care when reduction is stuck due to calls on nil, violations of argument type contracts for top-level calls, and violation of coverage contracts for conditionals: these correspond to a failed null check, a type-incorrect program, and an inadequately monitored program, respectively. However, we do care about "method not found" errors, since they would reveal that our inferred types, even for paths we have monitored, are unsound. The following rule (along with other versions of the monitoring rules augmented with error propagation, not shown) derives error judgments of the form $\mathcal{H}; \mathcal{E}; e \mid \phi \not\downarrow_{\mathcal{C}}$.

(ERROR)

$$\frac{\begin{array}{c}\mathcal{H}; \mathcal{E}; e \mid \phi \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}'; v \mid \phi' \\ \mathcal{H}'; \mathcal{E}'; e' \mid \phi' \longrightarrow_{\mathcal{C}} \mathcal{H}''; \mathcal{E}''; l \mid \phi'' \\ \mathcal{H}''(l) = A\langle \_ \rangle \qquad \mathcal{C}(A) = \mathcal{M} \qquad m \notin \texttt{dom}(\mathcal{M})\end{array}}{\mathcal{H}; \mathcal{E}; e'.\underline{m}(e) \mid \phi \not\downarrow_{\mathcal{C}}}$$

Finally, the following rule defines program error judgments of the form $\mathcal{T}; \Phi \vdash c^\star \rhd e \not\downarrow$.

(MONITOR)

$$\frac{\mathcal{C} = \texttt{classmap}(c^\star), \mathcal{T}, \Phi \qquad \{\}, \{\}, \texttt{monitor}(e) \mid \{\} \not\downarrow_{\mathcal{C}}}{\mathcal{T}; \Phi \vdash c^\star \rhd e \not\downarrow}$$

3.3 Soundness

We prove the following key soundness theorem.

Theorem 3.1 (Soundness)**.** *Suppose that $c^\star \rhd e_1 \uparrow \mathcal{T}; \Phi$ for some e_1. Then there cannot be e_2 such that $\mathcal{T}; \Phi \vdash c^\star \rhd e_2 \not\downarrow$.*

Informally, if we infer types $\mathcal{T}$ with a test that covers paths Φ, then as long as we run other tests that only traverse paths in Φ and satisfy the type contracts in $\mathcal{T}$, we cannot have "method-not-found" errors. In particular, this implies the following corollary.

Corollary 3.2. *If we infer types with a test that covers all possible paths, then our types are always sound.*

We sketch our proof below. Our monitoring semantics constrains execution to follow only paths traversed in the training run—this guarantees that if some expression in the methods of $c^\star$ is visited in the monitoring run, then it must be visited by the training run. Our proof is by simulation of the executions of such expressions between these runs.

We define the following simulation relation between heaps and environments of the training run (marked by subscript $\cdot_1$) and those of the monitoring run (marked by subscript $\cdot_2$).

Definition 3.3 (Simulation)**.** *$\mathcal{H}_1; \mathcal{E}_1$ simulates $\mathcal{H}_2; \mathcal{E}_2$ under types $\mathcal{T}$, denoted by $\mathcal{H}_1; \mathcal{E}_1 \sim_{\mathcal{T}} \mathcal{H}_2; \mathcal{E}_2$, iff the following hold:*

- *$\mathcal{E}_1 \sim_{\mathcal{T}} \mathcal{E}_2$, i.e., $\mathcal{E}_1(x) = \omega_1$ if and only if $\mathcal{E}_2(x) = v_2$ such that $\texttt{runtype}_{\mathcal{H}_2}(v_2) \leq \mathcal{T}(\texttt{type}(\omega_1))$, and $\mathcal{E}_1(\mathsf{self}) = l_1 : A$ if and only if $\mathcal{E}_2(\mathsf{self}) = l_2 : A$.*
- *$\mathcal{H}_1(\mathcal{E}_1(\mathsf{self})) = A\langle \mathcal{F}_1 \rangle$ such that $\mathcal{F}_1(@f) = \omega_1$ if and only if $\mathcal{H}_2(\mathcal{E}_2(\mathsf{self})) = A\langle \mathcal{F}_2 \rangle$ such that $\mathcal{F}_2(@f) = v_2$ and $\texttt{runtype}_{\mathcal{H}_2}(v_2) \leq \mathcal{T}(\texttt{type}(\omega_1))$.*
- *whenever $\mathcal{H}_1(l_1) = A\langle \mathcal{F}_1 \rangle$ such that $\mathcal{F}_1(@f) = \omega_1$ and $l_1 \neq \mathcal{E}_1(\mathsf{self})$, we have $\texttt{type}(\omega_1) = A.@f$; and whenever $\mathcal{H}_2(l_2) = A\langle \mathcal{F}_2 \rangle$ such that $\mathcal{F}_2(@f) = v_2$ and $l_2 \neq \mathcal{E}_2(\mathsf{self})$, we have $\texttt{runtype}_{\mathcal{H}_2}(v_2) \leq \mathcal{T}(\texttt{type}(A.@f))$.*

Informally, let us say that a value v_2 in the monitoring run agrees with a type τ_1 associated in the training run if the run-time type of v_2 is a subtype of the solution of τ_1. Then, we say that a training state (heap and environment) simulates a monitoring state (heap and environment) if the values of variables in the monitoring environment agree with their types in the training environment; the self objects in both environments have the same class; the fields of objects in the monitoring heap agree with their types in the training heap; and for all objects other than self, the fields of those objects in the training heap are of the form $A.@f$. This last requirement essentially says that while self fields may have flow-sensitive types, all other fields must be "stabilized" with their flow-insensitive types. We define this notion of stability for training and monitoring heaps below.

Definition 3.4 (Training heap stability)**.** *$\mathcal{H}_1$ is training-stable iff, whenever $\mathcal{H}_1(l_1) = A_1\langle \mathcal{F}_1 \rangle$ such that $\mathcal{F}_1(@f) = \omega_1$, we have $\texttt{type}(\omega_1) = \texttt{type}(A_1.@f)$.*

Definition 3.5 (Monitoring heap stability)**.** *$\mathcal{H}_2$ is monitoring-stable iff, whenever $\mathcal{H}_2(l_2) = A_2\langle \mathcal{F}_2 \rangle$ such that $\mathcal{F}_2(@f) = v_2$, we have $\texttt{runtype}_{\mathcal{H}_2}(v_2) \leq \mathcal{T}(\texttt{type}(A_2.@f))$.*

Monitoring and training heaps can be stabilized by the operations of constraining and wrapping, respectively, as shown by the lemmas below. Recall that these operations happen upon entry and exit of method calls; thus, we can ensure that the flow-sensitivity of field types does not "leak" across method calls. This is crucial for our proof, as shown later.

Lemma 3.6 (Constraining)**.** *Suppose that $\mathcal{H}_1; \mathcal{E}_1 \sim_{\mathcal{T}} \mathcal{H}_2; \mathcal{E}_2$. Suppose that whenever $\tau \leq \tau' \in \texttt{constrain}_{\mathcal{E}_1(\mathsf{self})}(\mathcal{H}_1))$, we have $\mathcal{T}(\tau) \leq \mathcal{T}(\tau')$. Then $\mathcal{H}_2$ is monitoring-stable.*

Lemma 3.7 (Wrapping)**.** *Suppose that $\mathcal{H}_1; \mathcal{E}_1 \sim_{\mathcal{T}} \mathcal{H}_2; \mathcal{E}_2$. Then $\texttt{wrap}_{\mathcal{E}_1(\mathsf{self})}(\mathcal{H}_1)$ is training-stable.*

We now have a neat way of establishing that a training state simulates a monitoring state. If both heaps are stabilized (say by constraining and wrapping), and furthermore, if the environments themselves satisfy the requirements for simulation, then the states are related by simulation.

Lemma 3.8 (Proof technique for simulation)**.** *Suppose that $\mathcal{H}_1$ is training-stable; $\mathcal{H}_2$ is monitoring-stable; and $\mathcal{E}_1 \sim_{\mathcal{T}} \mathcal{E}_2$. Then $\mathcal{H}_1; \mathcal{E}_1 \sim_{\mathcal{T}} \mathcal{H}_2; \mathcal{E}_2$.*

This proof technique is used to show the following main lemma: if we run the same expression in both runs and the initial training heap simulates the initial monitoring heap, then we cannot have method-not-found errors in the monitoring run, the result of the monitoring run agrees with its type in the training run, and the final training heap simulates the final monitoring heap.

Lemma 3.9 (Preservation)**.** *Suppose that $\_ \rhd \_ \uparrow \mathcal{T}, \Phi$, in particular deriving $\mathcal{H}_1; \mathcal{E}_1; e \longrightarrow_{\mathcal{C}_1} \_; \mathcal{H}_1'; \mathcal{E}_1'; \_ : \tau_1 \mid \phi$. Let $\mathcal{C}_2 = \mathcal{C}_1, \mathcal{T}, \Phi$. Let $\mathcal{H}_2$ and $\mathcal{E}_2$ be such that $\mathcal{H}_1; \mathcal{E}_1 \sim_{\mathcal{T}} \mathcal{H}_2; \mathcal{E}_2$. Then:*

- *We cannot have $\mathcal{H}_2; \mathcal{E}_2; e \not\downarrow_{\mathcal{C}_2}$.*
- *If $\mathcal{H}_2; \mathcal{E}_2; e \longrightarrow_{\mathcal{C}_2} \mathcal{H}_2'; \mathcal{E}_2'; v_2$ with $\phi, \phi' \in \mathcal{E}_2$ and $\phi' \in \mathcal{E}_2'$ for some ϕ', then $\texttt{runtype}_{\mathcal{H}_2}(v_2) \leq \mathcal{C}_2(\tau_1)$ and $\mathcal{H}_1'; \mathcal{E}_1' \sim_{\mathcal{T}} \mathcal{H}_2'; \mathcal{E}_2'$.*

Proof. The proof is by induction on the monitoring run. The only difficult case is for method calls, which we sketch below. (Note that these method calls are internal, not top-level, since only environments for which self is defined can appear in the simulation.)

By the induction hypothesis we cannot have errors in the evaluation of the argument and the receiver; let us say they evaluate to v_2 and l_2. Furthermore, the simulation relation must hold for resulting states, say $\mathcal{H}_1; \mathcal{E}_1$ and $\mathcal{H}_2; \mathcal{E}_2$, and v_2 and l_2 must agree with the types τ_1 and τ_1' associated with the argument and receiver in the training run, respectively. Since τ_1' is a subtype of a structural type with method m, we cannot have a method-not-found error here.

The main complication now is that the method dispatched in the training run may not be the same as that in the monitoring run, although the latter method should be dispatched at some point in the training run. Thus, to apply our induction hypothesis for the method call, we need to show that the initial state for that method call in the monitoring run is simulated by the possibly unrelated initial state with which that method was called in the training run. However, at this point we can apply Lemmas 3.6, 3.7, and 3.8 to relate those states, since we constrain the initial state of the caller and wrap the initial state of the callee. Thus, now by induction hypothesis, the method expression evaluates to (say) v_2'', such that the simulation relation must hold for resulting states, say $\mathcal{H}_1''; \mathcal{E}_1''$ and $\mathcal{H}_2''; \mathcal{E}_2''$ and v_2'' must agree with the type τ_1'' associated with the result in the training run. Finally, as above, we apply Lemmas 3.6, 3.7, and 3.8 to show that the final state for the method call in the training run simulates the final state for the method call in the monitoring run, since we constrain the final state of the callee and wrap the final state of the caller. □

Now using Lemma 3.9, we can show that the following weaker state invariant is preserved at the top level of a monitoring run.

Definition 3.10. *At the top level,* $\texttt{invariant}(\mathcal{H}_2; \mathcal{E}_2)$ *holds iff:*

- *whenever* $\mathcal{H}_2(l) = A\langle\mathcal{F}\rangle$ *such that* $\mathcal{F}(@f) = v_2$*, we have* $\texttt{runtype}_{\mathcal{H}_2}(v_2) \leq \mathcal{C}_2(A.@f)$*;*
- $\mathcal{E}_2(\mathsf{self})$ *is undefined.*

Lemma 3.11. *Suppose that* $c^\star \rhd_{\_} \uparrow \mathcal{T}; \Phi$ *with* $\mathcal{C}_1 = \texttt{classmap}(c^\star)$*. Let* $\mathcal{C}_2 = \mathcal{C}_1, \mathcal{T}, \Phi$*. Let* $\mathcal{H}_2$ *and* $\mathcal{E}_2$ *be such that* $\texttt{invariant}(\mathcal{H}_2; \mathcal{E}_2)$*. Then:*

- *We cannot have* $\mathcal{H}_2; \mathcal{E}_2; \texttt{monitor}(e_0) \lightning_{\mathcal{C}_2}$*.*
- *If* $\mathcal{H}_2; \mathcal{E}_2; \texttt{monitor}(e_0) \longrightarrow_{\mathcal{C}_2} \mathcal{H}_2'; \mathcal{E}_2'; v_2$ *then we must have* $\texttt{invariant}(\mathcal{H}_2'; \mathcal{E}_2')$*.*

Theorem 3.1 follows by Lemma 3.11.

Perhaps the most interesting thing about this theorem is that it proves we do not need to train on all iterations of a loop (here formalized as recursive calls to a function), but rather just all paths within the loop body. Here is an example to provide some intuition as to why. Suppose a recursive method walks over an array and calls baz on each element, e.g.,

```
def foo(a, i)
  return if i == a.length
  a[i].baz
  foo(a, i+1)
end
```

Recall that a[i] is actually a method call to the [] method, i.e., a[i].baz is really a.[](i, baz). If α is the array's contents type variable (i.e., the array has type Array<α>), this call generates a constraint $\alpha \leq$ [baz : ...]. This constraint, generated from a single call to foo, affects *all* elements of a because when the array was created, we added constraints that each of its element types are compatible with α, e.g., $\tau_1 \leq \alpha$, $\tau_2 \leq \alpha$, etc. where the τ_i are the element types. (Thus the solution to α will contain $\tau_1 \cup \tau_2 \cup ...$)

```
65 class A
66   infer_types ()                # A is an unannotated class
67   def foo ... end
68   def bar ... end
69 end
70 class String                    # String is an annotated class
71   typesig (" insert : (Numeric, String) → String")
72   typesig ("+ : ([ to_s : ()→String]) → String")
73   ...
74 end
75 class ATest
76   include Rubydust::RuntimeSystem::TestCase
77   def test_1 ... end   # test cases called by Rubydust
78   def test_2 ... end
79 end
```

Figure 9. Using Rubydust

We can also prove the following "completeness" theorem.

Theorem 3.12 (Completeness)**.** *If there are* N *methods and a maximum of* K *labels in each method, and each label is reachable, then a program of size* $O(N \cdot 2^K)$ *is sufficient to ensure that the inferred types are sound over all runs.*

Proof. For any method m, there can be at most 2^K paths with the labels in m; and there can be at most N such methods. □

In practice, the completeness bound $N \cdot 2^K$ is likely quite loose. Of course, a tighter bound could be obtained by considering the actual number of labels K in each method. Furthermore, a method with K labels may induce far less paths than 2^K—indeed, we are estimating the number of nodes in a branching tree of height K, which may even be K if the tree is skewed as a list, as is the case for switch statements.

4. Implementation

In this section we describe Rubydust, an implementation of our dynamic type inference algorithm for Ruby. Rubydust comprises roughly 4,500 lines of code, and is written purely in standard Ruby.

Using Rubydust is straightforward, as illustrated with the example in Figure 9. To use a program with Rubydust, the programmer simply runs the program as `ruby rubydust.rb` ⟨*program*⟩. The Ruby script `rubydust.rb` in turn sets up the Ruby environment before running the target program ⟨*program*⟩. For each class whose types should be inferred, the programmer adds a call to Rubydust's infer_types method during the class definition (line 66). Note that in Ruby, class definitions are executed to create the class, and hence methods can be invoked as classes are defined.

For classes with annotated types, the programmer calls Rubydust's typesig method with a string for each type to be declared. For example, on line 71, we declare that String :: insert takes a Numeric and String and returns a String. As another example, on line 72, we declare that String::+ takes a argument that has a to_s method and returns a String. We support the full type annotation language from DRuby [12], including intersection types and generics (discussed below).

Finally, the programmer also needs to define several test cases, which are run when the program is executed by Rubydust. After Rubydust runs the test cases, it solves the generated constraints and outputs the inferred types (examples in Section 5).

Next, we briefly discuss details of the instrumentation process, constraint resolution, and some limitations of our implementation.

Instrumenting Unannotated Classes Wrapped objects $v : \tau$ are implemented as instances of a class Proxy that has three fields: the object that is being wrapped, its type, and the *owner* of the Proxy, which is the instance that was active when the Proxy was created. When a method is invoked on a Proxy, the object's method_missing method will be called; in Ruby, if such a method is defined, it receives calls to any undefined methods. Here method_missing does a little work and redirects the call to the wrapped object.

To implement the wrapping (with Proxy) and constraint generation operations, we use Ruby introspection to *patch* the unannotated class. In particular, we rename the current methods of each unannotated class and then add method_missing to perform work before and after delegating to the now-renamed method. We need to patch classes to bootstrap our algorithm, as the program code we're tracking creates ordinary Ruby objects whose method invocations we need to intercept.

On entry to and exit from a patched method, we perform all of the constraint generation and wrapping, according to Figure 1. Note that we perform both the caller and the callee's actions in the callee's method_missing. This is convenient, because it allows us rely on Ruby's built-in dynamic dispatch to find the right callee method, whereas if we did work in the caller, we would need to reimplement Ruby's dynamic dispatch algorithm. Moreover, it means we can naturally handle dispatches via send, which performs reflective method invocation.

Since we are working in the callee, we need to do a little extra work to access the caller object. Inside of each patched class, we add an extra field *dispatcher* that points to the Proxy object that most recently dispatched a call to this object; we set the field whenever a Proxy is used to invoke a wrapped-object method. Also recall that each Proxy has an owner field, which was set to **self** at the time the proxy was created. Since we wrap arguments and fields whenever we enter a method, this means all Proxy objects accessible from the current method are always owned by **self**. Thus, on entry to a callee, we can find the caller object by immediately getting its dispatching Proxy, and then finding the owner of that Proxy.

Finally, notice that the above discussion suggests we sometimes need to access the fields of an object from a different object. This is disallowed when trying to read and write fields normally, but there is an escape hatch: we can access field @f of o from anywhere by calling o. instance_eval ("@f"). In our formalism, we assumed fields were only accessible from within the enclosing object; thus, we may be unsound for Ruby code that uses instance_eval to break the normal access rules for fields (as we do!).

Instrumenting Annotated Classes As with unannotated classes, we patch annotated classes to intercept calls to them, and we perform constraint generation and wrapping for the caller side only, as in Figure 1. We also fully unwrap any arguments to the patched method before delegating to the original method. We do this to support annotations on Ruby's core library methods, which are actually implemented as native code and expect regular, rather than Proxy-wrapped, objects. The actual method annotations for classes are stored in the class object, and can thus be retrieved from the patched method by inspecting **self** . **class**. For some methods, the proxy forwards intercepted calls to the original method, unwrapping all the arguments. For example, we forward calls to eql? and hash so wrapped objects will be treated correctly within collections.

Rubydust includes support for polymorphic class and method types. If a class has a polymorphic type signature, e.g., A<t>, we instantiate its type parameters with fresh type variables whenever an instance is created. We store the instantiated parameters in the instance, so that we can substitute them in when we look up a method signature. For methods that are polymorphic, we instantiate their type parameters with fresh type variables at the call.

Lastly, we also support intersection types for methods, which are common in the Ruby core library [12]. If we invoke o.m(x), and o.m has signature $(A \rightarrow B) \cap (C \rightarrow D)$, we use the run-time type of x to determine which branch of the intersection applies. (Recall we trust type annotations, so if the branches overlap, then either branch is valid if both apply.)

Constraint solving and type reconstruction We train a program by running it under a test suite and generating subtyping constraints, which are stored in globals at run time. At the end, we check the consistency of the subtyping constraints and solve them to reconstruct types for unannotated methods. The type language for reconstruction is simple, as outlined in Section 3; we do not try to reconstruct polymorphic or intersection types for methods. Consequently, the algorithms we use are fairly standard.

We begin by computing the transitive closure of the subtyping constraints to put them in a *solved form*. Then, we can essentially read off the solution for each type variable. First, we set method return type variables to be the union of their (immediate) lower bounds. Then, we set method argument type variables to be the intersection of their (immediate) upper bounds. These steps correspond to finding the least type for each method. Then we set the remaining type variables to be either the union of their lower-bounds or intersection of their upper-bounds, depending on which is available. Finally, we check that our solved constraints, in which type variables are replaced by their solutions, are consistent.

Limitations There are several limitations of our current implementation, beyond what has been mentioned so far. First, for practicality, we allow calls to methods whose classes are neither marked with infer_types () nor provided with annotations; we do nothing special to support this case, and it is up to the programmer to ensure the resulting program behavior will be reasonable. Second, we do not wrap **false** and **nil**, because those two values are treated as false by conditionals, whereas wrapped versions of them would be true. Thus we may miss some typing constraints. However, this is unlikely to be a problem, because the methods of **false** and **nil** are rarely invoked. For consistency, we also do not wrap **true**. Third, Rubydust has no way to intercept the creation of Array and Hash literals, and thus initially Rubydust treats such a literal as having elements of type $\top$. The first time a method is invoked on an Array or Hash literal, Rubydust iterates through the container elements to infer a more precise type. Thus if an Array or Hash literal is returned before one of its methods is invoked, Rubydust will use the less precise $\top$ for the element type for the constraint on the return variable. (This limitation is an oversight we will address soon.)

Fourth, for soundness, we would need to treat global variables similarly to instance and class fields, we but do not current implement this. Fifth, Ruby includes looping constructs, and hence there are potentially an infinite number of paths through a method body with a loop. However, as the foo example at the end of Section 3 illustrates, if types of the loop-carried state are invariant, the number of loop iterations is immaterial as long as all internal paths are covered. We manually inspected the code in our benchmarks (Section 5) and found that types are in fact invariant across loops. Note that looping constructs in Ruby actually take a code block—essentially a first-class method—as the loop body. If we could assign type variables to all the inputs and outputs of such blocks, we could eliminate the potential unsoundness at loops. However, Ruby does not currently provide any mechanism to intercept code block creation or to patch the behavior of a code block.

Sixth, we currently do not support annotations on some low-level classes, such as IO, Thread, Exception, Proc, and Class. Also, if methods are defined during the execution of a test case, Rubydust will not currently instrument them. We expect to add handling of these cases in the future.

	LOC	TC	E(#)	LCov	MCov	P(#)	PCov	OT(s)	RT(s)	Solving(s)
ministat-1.0.0	96	10	7	75%	11 / 15	19	84%	0.00	11.19	57.11
finitefield-0.1.0	103	9	6	98%	12 / 12	14	93%	0.00	1.74	1.28
Ascii85-1.0.0	105	7	3	95%	2 / 2	67	28%	0.01	6.81	0.17
a-star	134	1	5	100%	20 / 24	41	62%	0.04	114.81	37.46
hebruby-2.0.2	178	19	8	81%	20 / 26	36	91%	0.01	19.97	19.08
style-0.0.2	237	12	1	75%	17 / 32	88	28%	0.01	8.46	0.28
Rubyk	258	1	4	69%	15 / 20	37	68%	0.00	7.33	0.56
StreetAddress-1.0.1	772	1	10	79%	33 / 44	23	88%	0.02	4.45	24.58

TC - test cases E - manual edits LCov - line coverage MCov - method coverage / total # of methods
P - paths PCov - path coverage OT - original running time RT - Rubydust running time

Figure 10. Results

Finally, in Rubydust, infer_types () is a class-level annotation—either all methods in a class have inferred types, or none do. However, it is fairly common for Ruby programs to patch existing classes or inherit from annotated classes. In these cases, we would like to infer method signatures for just the newly added methods; we plan to add support for doing so in the future.

5. Experiments

We ran Rubydust on eight small programs obtained from RubyForge and Ruby Quiz [23]. We ran on a Mac Pro with two 2.4Ghz quad core processors with 8GB of memory running Mac OS X version 10.6.5. Figure 10 tabulates our results. The column headers are defined at the bottom of the figure. The first group of columns shows the program size in terms of lines of code (via SLOCcount [30]), the number of test cases distributed with the benchmark, and the number of manual edits made, which includes rewriting Array and Hash literals, and inserting calls to infer_types . For the *ministat* benchmark, two of the edits ensure the test setup code is called only once across all tests; without this change, the constraint solver does not complete in a reasonable amount of time. For all benchmarks, when calculating the lines of code, number of methods, and manual edits made we excluded testing code. For *style*, we also did not count about 2,700 lines of the source file that occur after Ruby's __END__ tag, which tells the interpreter to ignore anything below that line. In this case, those lines contain static data that the program loads at run-time by reading its own source file.

The next group of columns gives the line coverage from the test cases (computed by rcov [21]), the method coverage, the sum of the number paths in covered methods, and the percentage of these paths covered by the test cases. As rcov does not compute path coverage, we determined the last two metrics by hand—for each method, we inserted print statements on every branch of the program to record which branches were taken at run time. We then manually analyzed this information to determine path coverage. When considering paths, we tried to disregard infeasible paths (e.g., conditionals with overlapping guards) and paths that only flagged errors. The path coverage is generally high, with the exception of *Ascii85* and *style*, which have long methods with sequences of conditionals that create an exponential number of paths.

We manually inspected the source code of the benchmark programs, and we determined that for the methods that were covered, the type annotations inferred by Rubydust are correct. It is interesting that the annotations are correct even for the two programs with low path coverage; this suggests that for these programs, reasonable line coverage is sufficient to infer sound types. Rubydust also found one type error, which we discuss below.

Performance The last group of columns in Figure 10 reports Rubydust's running time, which we split into the time to instrument and run the instrumented program, and the time to solve the generated constraints. As we can see, the overhead of running under Rubydust, even excluding solving time, is quite high compared to running the original, uninstrumented program. Part of the reason is that we have not optimized our implementation, and our heavy use of wrappers likely impedes fast paths in the Ruby interpreter. For example, values of primitive types like numbers are not really implemented as objects, but we wrap them with objects. Nevertheless, we believe that some overhead is acceptable because inference need not be performed every time the program is run.

The solving time is high, but that is likely because our solver is written in Ruby, which is known to be slow (this is being addressed in newer versions of Ruby). We expect this solving time would decrease dramatically if we exported the constraints to a solver written in a different language.

Inferred Types We now describe the benchmark programs and show some example inferred types, as output by Rubydust.

Ministat generates simple statistical information on numerical data sets. As an example inferred type, consider the median method, which computes a median from a list of numbers:

```
median: ([ sort !:  () → Array<Numeric>;
           size :  () → Numeric;
           ' [] ' :  (Numeric) → Numeric])
         → Numeric
```

The method takes an object that has sort !, size , and [] methods, and returns a Numeric. Thus, one possible argument would be an Array of Numeric. However, this method could be used with other arguments that have those three methods—indeed, because Ruby is dynamically typed, programmers are rarely required to pass in objects of exactly a particular type, as long as the passed-in objects have the right methods (this is referred to as "duck typing" in the Ruby community).

Finitefield, another mathematical library, provides basic operations on elements in a finite field. As an example type, consider the inverse method, which inverts a matrix:

```
inverse :  ([ '>': (Numeric) → Boolean;
              '≪': (Numeric) → Numeric;
              '≫': (Numeric) → Numeric;
              '&':  (Numeric) → Numeric;
              '^' :  (Numeric) → Numeric]) → Numeric ")
```

As above, we can see exactly which methods are required of the argument; one concrete class that has all these methods is Numeric.

Ascii85 encodes and decodes data following Adobe's binary-to-text Ascii85 format. There are only two methods in this program, both of which are covered by the seven test cases. Rubydust issues an error during the constraint solving, complaining that Boolean is not a subtype of [to_i : () → Numeric]. The offending parts of the code are shown below.

```
module Ascii85
  def self .encode(str ,  wrap_lines =80)
     ...  if  (! wrap_lines ) then  ...  return end
```

```
    ... wrap_lines . to_i
  end ...
end
```

The author of the library uses wrap_lines as an optional argument, with a default value of 80. In one test case, the author passes in **false**, hence wrap_lines may be a Boolean. But as Boolean does not have a to_i method, invoking wrap_lines . to_i is a type error. For example, passing **true** as the second argument will cause the program to crash. It is unclear whether the author intends to allow **true** as a second argument, but clearly wrap_lines can potentially be an arbitrary non-integer, since its to_i method is invoked (which would not be necessary for an integer).

A-star is a solution for the A* search problem. It includes a class Tile to represent a position on a two-dimensional Map. Rubydust infers that two of the methods to Tile have types:

```
y: () → Numeric
x: () → Numeric
```

The Map class uses Tile, which we can see from the inferred types for two of Map's methods:

```
adjacent: ([y: () → Numeric; x: () → Numeric]) → Array<Tile>
find_route : () →
  [push: ([y: () → Numeric; x: () → Numeric]) → Array<Tile>;
   include ?: ([y: () → Numeric; x: () → Numeric]) → Boolean;
   pop: () → [y: () → Numeric; x : () → Numeric];
   to_ary : () → Array<Tile>]
```

The adjacent method computes a set of adjacent locations (represented as an Array<Tile>) given an input Tile. The find_route method returns an array of Tile, which is an instance of the slightly more precise structural type Rubydust infers. (Here the structural type is actually the type of a field returned by find_route .)

Hebruby is a program that converts Hebrew dates to Julian dates and vice versa. One method type we inferred is

```
leap ?: (['*' : (Numeric) → Numeric]) → Boolean
```

This method determines whether the year is a leap year. This signature is interesting because the argument only needs a single method, $*$, which returns a Numeric. This is the only requirement because subsequent operations are on the return value of $*$, rather than on the original method argument.

Rubyk is a Rubik's Cube solver; as this program did not come with a test suite, we constructed a test case ourselves (one of the four edits we made to the program). Some of the inferred types are shown below.

```
visualize :() → Array<Array<String>>
initialize :() → Array<Array<String>>
```

We can see that the visualize and initialize methods return a representation of the cube faces as an Array<Array<String>>.

Style is a "surface-level analysis tool" for English. The program has several methods that determine characteristics of English text. Some sample inferred types are:

```
pronoun? :([downcase : () → String]) → Boolean
passive? :([each : () → Array<String>]) → Boolean
preposition ? :([downcase : () → String]) → Boolean
abbreviation ? :([downcase : () → String]) → Boolean
nominalization ? :([downcase : () → String]) → Boolean
interrogative_pronoun ? :([downcase : () → String]) → Boolean
normalize :([downcase : () → String]) → String
```

Here, most of the methods first downcase their argument, and then use the resulting String. The passive method takes an array of Strings, representing a sentence, and iterates through the array to determine whether the sentence is in the passive voice.

Finally, *StreetAddress* is a tool that normalizes U.S. street address into different subcomponents. As an example, the parse class method takes a String and returns an instance of the class, which is depicted in the following output:

```
class << StreetAddress::US; # define a class method
  parse: (String) → StreetAddress::US::Address
end
```

Notes We encountered all of Rubydust's limitations in these benchmark programs—there were tests that called out to unannotated and uninferred classes; that use **false** and **nil**; that use global variables; and that use unannotated low-level classes like IO. However, as already mentioned, the types Rubydust inferred were correct, and so these limitations did not affect the results.

We also encountered the use of reflective method invocation via send described in Section 2.5. We did not encounter uses of other dynamic features for classes whose types are inferred (though they do occur internally in the standard library).

6. Related work

There has been significant interest in the research community in bringing static type systems to dynamic languages. Much recent research has developed ways to mix typed and untyped code, e.g., via quasi-static typing [26], occurrence typing [29], gradual typing [25], and hybrid typing [13]. In these systems, types are supplied by the user. In contrast, our work focuses on type inference, which is complementary: we could potentially use our dynamic type inference algorithm to infer type annotations for future checking.

Several researchers have explored type inference for object-oriented dynamic languages, including Ruby [2, 3, 10, 12, 16, 18], Python [4, 7, 24], and JavaScript [5, 27], among others. As discussed in the introduction, these languages are complex and have subtle semantics typically only defined by the language implementation. This makes it a major challenge to implement and maintain a static type inference system for these languages. We experienced this firsthand in our development of DRuby, a static type inference system for Ruby [3, 10, 12].

There has also been work on type inference for Scheme [31], a dynamic language with a very compact syntax and semantics; however, these inference systems do not support objects.

Dynamic type inference has been explored previously in several contexts. Rapicault et al. describe a dynamic type inference algorithm for Smalltalk that takes advantage of introspection features of that language [20]. However, their algorithm is not very clearly explained, and seems to infer types for variables based on what types are stored in that variable. In contrast, we infer more general types based on usage constraints. For example, back in Figure 2, we discovered argument x must have a qux method, whereas we believe the approach of Rapicault et al would instead infer x has type B, which is correct, but less general.

Guo et al. dynamically infer abstract types in x86 binaries and Java bytecode [14]. Artzi et al. propose a combined static and dynamic mutability inference algorithm [6]. In both of these systems, the inferred types have no structure—in the former system, abstract types are essentially tags that group together values that are related by the program, and in the latter system, parameters and fields are either mutable or not. In contrast, our goal is to infer more standard structural or nominal types.

In addition to inferring types, dynamic analysis has been proposed to discover many other program properties. To cite three examples, Daikon discovers likely program invariants from dynamic runs [9]; DySy uses symbolic execution to infer Daikon-like invariants [8]; and Perracotta discovers temporal properties of programs [32]. In these systems, there is no notion of sufficient coverage to guarantee sound results. In contrast, we showed we can soundly infer types by covering all paths through each method.

There are several dynamic inference systems that, while they have no theorems about sufficient coverage, do use a subsequent checking phase to test whether the inferred information is sound. Rose et al. [22] and Agarwal and Stoller [1] dynamically infer types that protect against races. After inference the program is annotated and passed to a type checker to verify that the types are sound. Similarly, Nimmer and Ernst use Daikon to infer invariants that are then checked by ESC/Java [19]. We could follow a similar approach to these systems and apply DRuby to our inferred types (when coverage is known to be incomplete); we leave this as future work.

Finally, our soundness theorem resembles soundness for *Mix*, a static analysis system that mixes type checking and symbolic execution [15]. In *Mix*, blocks are introduced to designate which code should be analyzed with symbolic execution, and which should be analyzed with type checking. At a high-level, we could model our dynamic inference algorithm in *Mix* by analyzing method bodies with symbolic execution, and method calls and field reads and writes with type checking. However, there are several important differences: We use concrete test runs, where *Mix* uses symbolic execution; we operate on an object-oriented language, where *Mix* applies to a conventional imperative language; and we can model the heap more precisely than *Mix*, because in our formal language, fields are only accessible from within their containing objects.

7. Conclusion

In this paper we presented a new technique, constraint-based dynamic type inference, that infers types based on dynamic executions of the program. We have proved that this technique infers sound types as long as all possible paths through each method are traversed during inference. We have developed Rubydust, an implementation of our technique for Ruby, and have applied it to a number of small Ruby programs to find a real error and to accurately infer types in other cases. We expect that further engineering of our tool will improve its performance. We also leave the inference of more advanced types, including polymorphic and intersection types, to future work.

Acknowledgments

We would like to thank Mark T. Daly for his work on the type annotation parser. Most of this work was performed while the first and second authors were at the University of Maryland, College Park. This work was supported in part by DARPA ODOD.HR00110810-073, NSF CCF-0346982, CCF-0541036, and CCF-0915978.

References

[1] Rahul Agarwal and Scott D. Stoller. Type Inference for Parameterized Race-Free Java. In *VMCAI*, 2004.

[2] O. Agesen, J. Palsberg, and M.I. Schwartzbach. Type Inference of SELF. *ECOOP*, 1993.

[3] Jong-hoon (David) An, Avik Chaudhuri, and Jeffrey S. Foster. Static Typing for Ruby on Rails. In *ASE*, 2009.

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

[5] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards Type Inference for JavaScript. In *ECOOP*, 2005.

[6] Shay Artzi, Adam Kiezun, David Glasser, and Michael D. Ernst. Combined static and dynamic mutability analysis. In *ASE*, 2007.

[7] John Aycock. Aggressive Type Inference. In *International Python Conference*, 2000.

[8] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In *ICSE*, 2008.

[9] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3), 2007.

[10] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA*, 2009.

[11] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. The Ruby Intermediate Language. In *Dynamic Language Symposium*, 2009.

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

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

[14] Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. Dynamic inference of abstract types. In *ISSTA*, 2006.

[15] Yit Phang Khoo, Bor-Yuh Evan Chang, and Jeffrey S. Foster. Mixing type checking and symbolic execution. In *PLDI*, 2010.

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

[17] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17, 1978.

[18] Jason Morrison. Type Inference in Ruby. Google Summer of Code Project, 2006.

[19] Jeremy W. Nimmer and Michael D. Ernst. Invariant Inference for Static Checking: An Empirical Evaluation. In *FSE*, 2002.

[20] Pascal Rapicault, Mireille Blay-Fornarino, Stéphane Ducasse, and Anne-Marie Dery. Dynamic type inference to support object-oriented reenginerring in Smalltalk. In *ECOOP Workshops*, 1998.

[21] rcov. rcov: code coverage for ruby, 2010. `http://eigenclass.org/hiki/rcov`.

[22] James Rose, Nikhil Swamy, and Michael Hicks. Dynamic inference of polymorphic lock types. *Sci. Comput. Program.*, 58(3), 2005.

[23] Ruby Quiz, 2010. `http://www.rubyquiz.com`.

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

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

[26] Satish Thatte. Quasi-static typing. In *POPL*, 1990.

[27] Peter Thiemann. Towards a type system for analyzing javascript programs. In *ESOP*, 2005.

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

[29] Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *POPL*, 2008.

[30] David A. Wheeler. Sloccount, August 2004. `http://www.dwheeler.com/sloccount/`.

[31] A.K. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM TOPLAS*, 19(1), 1997.

[32] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE*, 2006.