ABSTRACT

Title of dissertation: Dynamic Inference of Static Types for Ruby

Jong-hoon (David) An, Master of Science, 2010

Dissertation directed by: Professor Jeffrey S. Foster and
Professor Michael Hicks and
Department of Computer Science

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 thesis, 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. 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. We found it to be lightweight and useful: Rubydust discovered 1 real type error, and all other inferred types were correct, and readable.

Dynamic Inference of Static Types for Ruby

by

Jong-hoon (David) An

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2010

Advisory Committee:
Professor Jeffrey S. Foster, Chair/Co-advisor
Professor Michael W. Hicks, Co-advisor
Professor Neil Spring

Dedication

To my grandma who is in heaven now

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Static Type Inference

Over the years, there have been several efforts to bring static type inference to object-oriented dynamic languages such as Ruby, Python, and Perl [10, 12, 3, 2, 17, 19, 7, 23, 4, 5, 26]. Static type inference has the potential to provide the benefits of static typing—well-typed programs don't go wrong [18]—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 this reverse engineering process tedious and error-prone. Moreover, dynamic languages are so-named because of features such as reflection and `eval`, which, while making some programming idioms convenient, impede precise yet sound static analysis. Combined, these challenges make developing static type inference for a dynamic language a major undertaking, and maintaining a static type inference system as a dynamic language evolves over time is a daunting prospect.

1.2 Dynamic Inference of Static Types

In this thesis, we introduce *constraint-based dynamic type inference*, a technique that uses information gathered from dynamic runs to infer static types. More precisely, at run-time we introduce type variables for each position, e.g., fields, method arguments, and return values, whose type we want to infer. 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 appropriate *subtyping constraints* on those variables. At the end of the run, we solve the constraints to find a valid typing, if one exists.

1.3 Rubydust

We have implemented this technique for Ruby, as a tool called Rubydust (where "dust" stands for dynamic unraveling of static types). A key advantage of Rubydust's dynamic type inference is that, 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. Thus, it is more precise than a standard static type system. For example, Rubydust supports flow-sensitive treatment of local variables, allowing them to be assigned values having different types. Rubydust is also path-sensitive since it only sees actual runs of the program, and thus correlated branches pose no special difficulty. In essence, by observing only actual executions, Rubydust avoids much of the conservatism of standard static type inference.

Even better, although Rubydust is based purely on dynamic runs, we still proved a soundness theorem in [15]. We formalized our algorithm on a core subset of Ruby (shown in this thesis), and we previously proved that if the training runs 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 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.

- Ruby effectively includes direct looping constructs, and hence method bodies could potentially contain an unbounded number of paths. However, based on our experimental benchmarks, we have found 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.

- 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 [12]. 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 [10]). 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 run-time 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 5 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. This fact is interesting because the type 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.

1.4 Thesis

We believe that Rubydust is a practical, effective method for inferring useful static types in Ruby. In support of this thesis, 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. (Chapter 2)

- We formalize our algorithm and explain how type constraints are obtained throughout the test run. (Chapter 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. (Chapter 4)
- We evaluate Rubydust on a small set of benchmarks and find it to be useful. (Chapter 5)

Chapter 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 [27].

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

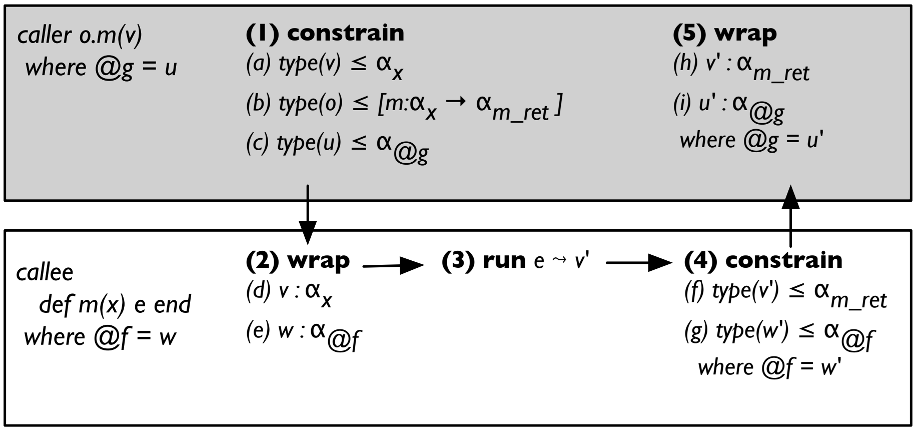


Figure 2.1: Dynamic instrumentation for a call o.m(v)

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 2.1 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, $\texttt{type}(x)$ 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 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 $\texttt{type}(o)$ 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 $\texttt{type}(v : \alpha_x)$, 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, except we are going 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.[1]

2.2 Complete Example

Now that we have seen the core algorithm, we can work through a complete example. Consider the code in Figure 2.2, which defines a class A with two methods foo and bar, and then calls foo on a fresh instance of A on line 21.

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 as the argument to typesig method on line 3, which indicates Numeric has a method + of type Numeric → Numeric.[2]

[1] Although in the figure and in our formalism we perform some operations in the caller, in Rubydust all operations are performed in the callee because it makes the implementation easier. See Chapter 4.

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

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

Figure 2.2: Basic method call and return example

As in the previous section, 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 21, 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 22 and 23) and wrap the actuals in the callee (line 9).

Next, on line 10, we use of 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

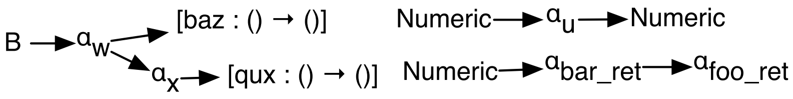
argument e_2.

arbitrary.)

On line 11, 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 2.1), 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) $\leq$ [+ : ...], 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 12), wrap the argument inside the callee (line 15), and generate constraints during execution of the body (line 16). At the return from bar, we constrain the return type (line 17) and wrap in the caller (yielding the wrapped value $7 : \alpha_{bar\_ret}$ on line 12). As that value is immediately returned by foo, we constrain the return type of foo with it (line 13) and wrap in the caller (line 24).

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:



(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

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

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

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

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

Consider class C in Figure 2.3. 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 27, 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 $\alpha_x \leq [\mathsf{baz} : () \to ()]$.

```
25 class C
26   def foo(x)
27     z = x;  z.baz();
28     z = 3; return z + 5;
29   end
30 end
31 class D
32   def bar(x)
33     @f = x
34   end
35   def baz()
36     y = 3 + @f
37   end
38   def qux()
39     @f = "foo"
40   end
41   def f()
42     bar("foo")
43   end
44 end
```

Figure 2.3: Example with local variables and fields

More interestingly, on line 28, 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. Each trace we can view 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;`". But 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 2.3, 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 $[\mathsf{bar} : \mathsf{Numeric} \rightarrow (), \mathsf{baz} : () \rightarrow (), \mathsf{qux} : () \rightarrow ()]$. But this result is clearly wrong, as the sequence qux(); baz(), which is "well-typed," 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.[3] Fortunately, we can still handle fields safely by applying the same principles we saw in Figure 2.1 for method arguments and returns. There, we needed two invariants:

[3] 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].

(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 2.1.

- 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 2.1.

Adding these extra constraints and wrapping operations solves the problem we saw above, in training the example in Figure 2.3 with the run `bar(1); baz(); qux()`. 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.

Our implementation handles class variables similarly to instance variables.

```
class E
  def foo(x, p)
    if p then x.qux() else x.baz() end
  end
  def bar(p)
    if p then y = 3 else y = "hello" end
    if p then y + 6 else y.length end
  end
end
```

(a) Paths and path-sensitivity

```
class F
  def foo(x)
    return (if x then 0 else "hello" end)
  end
  def bar(y,z)
    return (if y then foo(z) else foo(!z) end)
  end
end
f = F.new
```

(b) Per-method path coverage

Figure 2.4: Additional examples

2.4 Path Observation, Path-Sensitivity, and Path Coverage

As we discussed earlier, our algorithm observes dynamic runs. Hence for code with branches, we need to observe all possible paths through the code to infer sound types. For example, we can infer types for the foo method in Figure 2.4(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 2.4(a).

In a call bar(**true**), line 50 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 51, 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. (Note that our system supports union types, so the type we would infer for bar's argument p would be String $\cup$ Numeric.)

Our soundness theorem in our technical report [15] 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 2.4(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:[4]

f.bar(true, true)	f.bar(false, true)
Boolean $\leq \alpha_y$	Boolean $\leq \alpha_y$
Boolean $\leq \alpha_z$	Boolean $\leq \alpha_z$
$\alpha_z \leq \alpha_x$	$\alpha_z \leq$ 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.

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:

```
def initialize (args)
```

[4] Note that using x and y in an **if** generates no constraints, as any object can be used in such a position (**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.

```
  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.

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 [12]. In the dynamic analysis we propose in this thesis, there is no need for a separate profiling pass, as 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.

Chapter 3

Formalism

In this section, we formally describe our dynamic type inference technique using a core Ruby-like source language. This is a variation of our formalism in [15], which was originally developed by the authors of the technical report to prove the soundness theorem. Here, we describe only the semantics to explain how Rubydust wraps values and generate type constraints at runtime.

3.1 Syntax

The syntax is shown in Figure 3.1. Expressions include nil, self, variables x, fields $@f$, variable assignments $x = e$, field assignments $@f = e$, object creations $A.\mathsf{new}$, method calls $e.m(e')$, sequences $e; e'$, and conditionals $\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e''$.

The form $\mathsf{def}\ m(x) = e$ 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. In our formal proof [15], we run

$$
\begin{array}{rrcl}
\text{expressions} & e & ::= & \mathsf{nil} \mid \mathsf{self} \mid x \mid @f \mid x = e \\
 & & \mid & @f = e \mid A.\mathsf{new} \mid e.m(e') \mid e; e' \\
 & & \mid & \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' \\
\text{methods} & d & ::= & \mathsf{def}\ m(x) = e \\
\text{classes} & c & ::= & \mathsf{class}\ A = d^{\star} \\
\text{programs} & \mathcal{P} & ::= & c^{\star} \rhd e
\end{array}
$$

$$
\begin{array}{rrcl}
\text{types} & \tau & ::= & A.@f \mid A.m \mid A.\overline{m} \mid \bot \mid \top \\
 & & \mid & A \mid [m : A.m \to A.\overline{m}] \mid \tau \cup \tau' \mid \tau \cap \tau'
\end{array}
$$

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

Figure 3.1: Syntax of source language

other tests to "monitor" the system—i.e., show that the inferred types are sound. In this 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 $\tau \cap \tau'$. The bottom type $\bot$ represents the empty type or NilClass in Ruby. The top type $\top$ is the universal type–i.e. any type is a subtype of $\top$. Note that Object is still a subtype of $\top$ because an object is requried to have a minimum set of methods in order to be an Object in Ruby. A union type $\tau \cup \tau'$ can either be a τ or τ', and an intersection type $\tau \cap \tau'$ must be both τ and τ'.

3.2 Training Semantics

In this section, we 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.

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

Figure 3.2: Auxiliary syntax

To define the semantics, we need some auxiliary syntax to describe internal data structures, shown in Figure 3.2. Let l denote heap locations. Values include 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. Finally, constraints Π include standard subtyping constraints $\tau \leq \tau'$.

The rules shown in Figure 3.3 derive big-step reduction judgments of the form $\mathcal{H}; \mathcal{E}; e \longrightarrow_{\mathcal{C}} \mathcal{H}'; \mathcal{E}'; \omega \mid \Pi$, meaning that given $\mathcal{C}$, expression e under heap $\mathcal{H}$ and environment $\mathcal{E}$ reduces to wrapped value ω, 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 $\texttt{val}(\omega) = v$, $\texttt{type}(\omega) = \tau$, 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 $\{\}$.

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 Chapter 2, we permit some flow-sensitivity for field types. Thus, (Field) and (Field=) are much like (Var) and (Var=), in that they generate no constraint nor perform any wrapping. In general having flow-sensitive types for fields would be unsound; we recover soundness by restricting such flow-sensitivity across method calls (shown later), and relying on the fact that fields of objects of a particular class cannot be accessed by methods of other classes. (This device is not new—similar ideas appear in implementations of object invariants and STM.) Fortunately, our approach also slightly improves the precision of field types.

(TSeq) is straightforward. For the conditional expression, we have two rules—(TCond-True) for the true branch and (TCond-False) for the false branch. Note that we assume looping in the formal language occurs only via recursive calls.

There are two rules (TCaller) and (TCallee) to capture the behavior of method calls. We split method call handling into two rules to closely reflect the dynamic instrumentation introduced in Chapter 2. (TCaller) performs the actions

(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 =)

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

(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 =)

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

(TSeq)

$$\frac{\mathcal{H};\mathcal{E};e \longrightarrow_{\mathcal{C}} \mathcal{H}';\mathcal{E}';\_ \mid \Pi \qquad \mathcal{H}';\mathcal{E}';e' \longrightarrow_{\mathcal{C}} \mathcal{H}'';\mathcal{E}'';\omega \mid \Pi'}{\mathcal{H};\mathcal{E};(e;e') \longrightarrow_{\mathcal{C}} \mathcal{H}'';\mathcal{E}'';\omega \mid \Pi,\Pi'}$$

(TCond-True)

$$\frac{\mathcal{H};\mathcal{E};e \longrightarrow_{\mathcal{C}} \mathcal{H}';\mathcal{E}';\mathsf{true} : \mathsf{Boolean} \mid \Pi \qquad \mathcal{H}';\mathcal{E}';e' \longrightarrow_{\mathcal{C}} \mathcal{H}'';\mathcal{E}'';\omega' \mid \Pi'}{\mathcal{H};\mathcal{E};\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' \longrightarrow_{\mathcal{C}} \mathcal{H}'';\mathcal{E}'';\omega' \mid \Pi,\Pi'}$$

(TCond-False)

$$\frac{\mathcal{H};\mathcal{E};e \longrightarrow_{\mathcal{C}} \mathcal{H}';\mathcal{E}';\mathsf{false} : \mathsf{Boolean} \mid \Pi \qquad \mathcal{H}';\mathcal{E}';e'' \longrightarrow_{\mathcal{C}} \mathcal{H}'';\mathcal{E}'';\omega' \mid \Pi'}{\mathcal{H};\mathcal{E};\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' \longrightarrow_{\mathcal{C}} \mathcal{H}'';\mathcal{E}'';\omega' \mid \Pi,\Pi'}$$

(TCaller)

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

(TCallee)

$$\frac{\begin{array}{c}l = \mathcal{E}(\mathsf{self}) \qquad \mathcal{H}' = \texttt{wrap}_l(\mathcal{H}) \qquad \mathcal{E}' = \{x \mapsto \omega \bullet A.m\} \\ \mathcal{H}';\mathcal{E}';e \longrightarrow_{\mathcal{C}} \mathcal{H}'';\mathcal{E}'';\overline{\omega} \mid \Pi \qquad \overline{\tau} = \texttt{type}(\overline{\omega}) \qquad \overline{\Pi}' = \overline{\tau} \le A.\overline{m},\ \texttt{constrain}_l(\overline{\mathcal{H}})\end{array}}{\mathcal{H};\mathcal{E};\lambda(x)e \longrightarrow_{\mathcal{C}} \mathcal{H}'';\mathcal{E}'';\omega \mid \Pi,\overline{\Pi}'}$$

Figure 3.3: Training semantics

introduced in the caller part of Figure 2.1. 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. (TCallee) is then applied to evaluate the method body.

(TCallee) evaluates 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 ω'' is constrained to be a subtype of the result type $A.\overline{m}$ and returned as it is. Now (TCaller) finishes the job by wrapping the return value from the callee.

In addition, (TCaller) and (TCallee) involve 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

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

- $\mathtt{constrain}_l(\mathcal{H}) = \{\mathtt{type}(\omega) \leq A.@f \mid @f \mapsto \omega \in \mathcal{F}\}$

As discussed in Chapter 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)

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

We define:

$$\texttt{classmap}(c^\star) = \{A \mapsto \texttt{methodmap}(d^\star) \mid \textsf{class}\ A = d^\star \in c^\star\}$$

$$\texttt{methodmap}(d^\star) = \{m \mapsto \lambda(x)e \mid \textsf{def}\ m(x) = e \in d^\star\}$$

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 Chapter 4; however, our technique is agnostic to the choice of algorithm or even to the language of solved types.

Finally, as mentioned earlier, the formal proof for the soundness theorem can found in [15].

Chapter 4

Implementation

In this section we describe Rubydust, an implementation of our dynamic type inference algorithm for Ruby. Figure 4.1 shows the basic architecture of Rubydust, which comprises roughly 4,300 lines of code, and is written purely in Ruby. The shaded boxes indicate the different modules in Rubydust. We used *Rex* and *Racc* to generate the lexer and the parser, respectively, that scan and parse the type annotations. We exploit Ruby's powerful dynamic introspection features to implement Rubydust as a library, rather than requiring modifications to the interpreter. Since this does not require installations of many other modules than Rubydust itself and the Ruby Graph Library (RGL), we believe this tool is easy to setup and use for most Ruby programmers.

4.1 Rubydust Architecture

In this section, we discuss details of the instrumentation process, constraint resolution, and some limitations of our implementation.

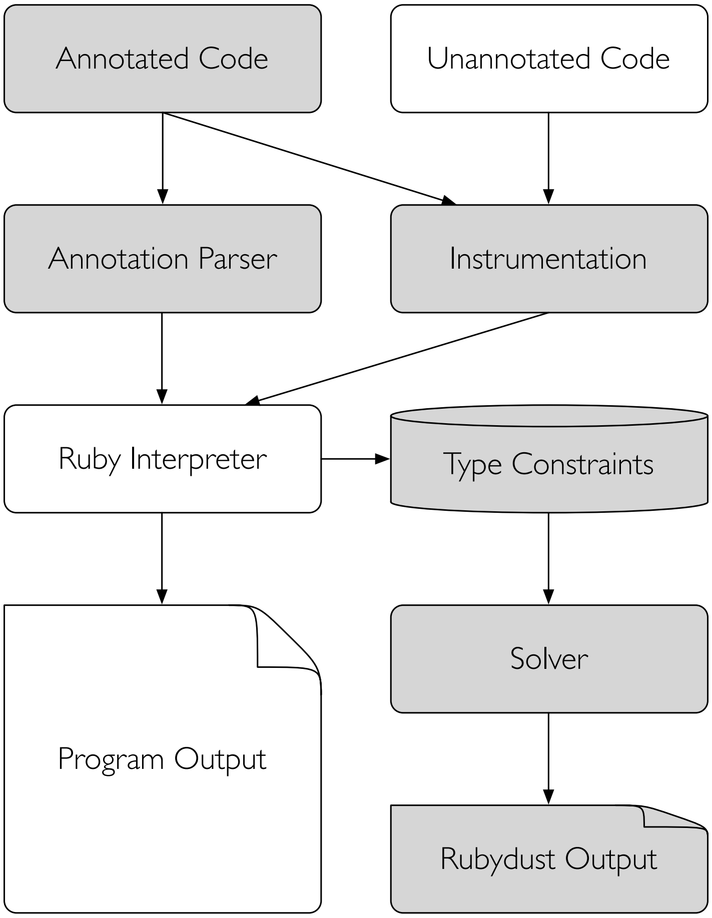


Figure 4.1: Rubydust architecture

As shown in Figure 4.1, Rubydust loads annotated code and unannotated code into memory. Because the Ruby core library is mostly written in C, we provide stubs for libary classes in a file named base_types.rb, which contains type annotations for the classes. This is a variation of the same file released with Diamondback Ruby (DRuby) and DRails [10, 11, 12, 3]. The original base_types.rb was converted to Rubydust format to support dynamic parsing of annotations. Currently, base_types.rb consists of about 800 lines of Ruby code and includes most of the classes in the core library, though not low-level such as IO, Thread, Exception, Proc, and Class to avoid passing wrapped values into their methods which may cause serious problems at runtime.

In addition to base_types.rb file, the user can always add more annotations as necessary. Rubydust's output includes annotations describing the inferred types, so one could use Rubydust to create annotations for one module and then use them when analyzing another module. As the annotated files are loaded, Rubydust parses individual annotations and stores the type information in class objects so that they can be retrieved later for inferring types.

Once the files are loaded, Rubydust dynamically patches them, so they generate constraints as according to the rules in Chapter 3. The instrumented code then is run via a set of *test cases*, which are not modified by Rubydust. Ruby executes the program and produces the typical output of the program as well as type constraints gathered during the run. The type constraints are then fed into the constraint solver, from which Rubydust generates type signatures or error messages.

4.1.1 Type Annotations

Here we describe our type annotation language as well as how annotations are parsed and stored in more detail. As introduced in [10], we support basic types such as nominal types and method types as well as several Ruby type idioms, which include intersection types, optional argument and variable length types, union types, the self type, structural types, and parametric polymorphism. We do not currently support first class methods and tuple types but plan to add these features in a future version of Rubydust. Furthermore, we support only types for method type signatures and class type parameter declarations in the current implementation. Future versions of Rubydust will support type annotations for fields, constants, and global variables.

Figure 4.2 shows the examples of types supported by Rubydust, taken from base_types.rb file. Each typesig method call has a single annotation entry in a string, which is parsed dynamically by Rubydust's type annotation parser when the method is invoked. The parser then stores the type information in the class object which is **self** at the call site. Note that in Ruby, class definitions are executed to create the class, and hence methods such as typesig can be invoked as classes are defined. In Ruby, everything is an object, and all objects are class instances. For example, line 73 declares a type for the method +, which concatenates a String argument with the receiver and returns a new String.

Rubydust supports *union* types, which allow programmers to mix different classes that share common methods. For example, String's include? method deter-

```
class String
  typesig("'+' : (String) → String")
  typesig("include? : Numeric or String → Boolean")
  typesig(" '[]' : Numeric → Numeric")
  typesig(" '[]' : Range or Regexp or String → String")
  typesig(" '[]' : (Numeric or Regexp, Numeric) → String")
  typesig("chomp: (?String) → String")
  typesig("delete: (String,∗String) → String")
  ...
end

class Regexp
  typesig("'=∼' : [to_str : () → String] → Numeric")
  ...
end

typesig("class Array<t>")
class Array
  typesig(" '[]' : Range → Array<t>")
  typesig(" '[]' : (Fixnum, Fixnum) → Array<t> ")
  typesig(" '[]' : Fixnum → t ")
  ...
  typesig("assoc<self,u> ; self ≤ Array<Array<u>> : u → Array<u>")
end
```

Figure 4.2: Selected type annotations from base_types.rb

mines whether or not the given String or character (Numeric) is contained in the receiver, and the result is a Boolean. This type is shown in line 74.

Dually to union types, Rubydust also supports *intersection* types, which are used to describe overloaded methods, i.e., methods that have different behaviors depending on the number and types of their arguments. String's [] method, which is the index reader, has three possible cases. First, one can look up a character at a position n, in which case, the method takes a Numeric (the index n) and returns a Numeric (a corresponding character at the position n), as shown in line 75. Second, one can find a substring (String) within a range of indices (Range), a regular expression (Regexp), or another String (line 76). There are two other cases: 1) String's [] method takes a Regexp and a group number (Numeric) and returns the substring of the matched group. 2) Or, String's [] method takes takes two Numerics to indicate the beginning and the end of a range and returns a new String. Both of these cases can be represented in one type, as shown in line 77.

Rubydust also supports optional argument types and variable length argument types, which are actually shorthands for intersection types with different number of arguments. Line 78 gives a type for the chomp method, which removes the substring from the end of the string to the separator. The separator, by default, is set to a newline, but can be specified by the user. Thus, we use optional argument type for the separator type. Similarly, line 79 shows that delete method takes an indefinite number (≥ 1) of arguments, whose matches are deleted from the receiver.

Structural types $[m_0 : t_0, ..., m_n : t_n]$ describe an object having at least the methods m_i with types t_i. As shown in line 84, Regexp's to_str method takes an object that has at least one method named to_str, which returns a String. Structural types are important in Ruby because they essentially depict objects' requirements in a method more precisely than nominal types.

Lastly, Rubydust type system includes parametric polymorphism and the self type We support both class level and method level parametric polymorphism. The former can be declared by using a declaration style similar to Java, as shown in line 88. Here, the type parameter t is bound at the top of the class and can be used anywhere inside that class. For instance, index reader method [] of Array, as shown in lines 90-91, uses the type parameter t in the possible return types.

Methods may also be parametrically polymorphic in which the type variable is bound at the method level. The **self** type is actually an instance of method polymorphism. It is bound at the method level and can be used anywhere in that method's type. Furthermore, we support adding constraints on the type parameters using the standard subyping relation. For example, Array's rassoc method, which searches for subarrays using the key u in an Array of Arrays of type u, i.e. **self** should be of that type (line 94).

4.1.2 Parsing Type Annotations

As discussed earlier, Rubydust parses type annotations as typesig methods are invoked at runtime. The actual method annotations for classes are stored in the class object, and can thus be retrieved from the patched class by inspecting **self.class**.

Rubydust includes support for polymorphic class and method types. If a class

```
class BlankSlate
  # this removes all methods
  instance_methods.each {|m| undef_method m }
end

module Proxyness
   attr_accessor :__obj
   attr_reader :__tvar
   attr_reader :__owner
  def __is_proxy?(); return true end
  def __object(); @__obj.respond_to?(:__is_proxy?) ? @__obj.__object : @__obj end
   ...
end

class ProxyObject < BlankSlate
  include Proxyness                    # extends the above module
  def method_missing(mname, *args, &blk)
    @__obj.instance_variable_set(:@__dispatcher, self) # sets the __dispatcher field of @__obj to self
    retval = @__obj.send(:"#{mname}", *args, &blk)
    return retval
  end
  def self.create(obj, tvar, owner)
     ...
  end
end
```

Figure 4.3: ProxyObject class

has a polymorphic type signature, e.g., Array<t>, we instantiate its type parameters with fresh type variables whenever a method is invoked on an instance for the first time. We cannot instantiate the type parameters at the object instantiation site because 1) we cannot capture the creation of an array or hash literal (discussed next), and 2) we do not currently patch Object's new class method because almost every object in Ruby is created via this class method. To illustrate how we handle array or hash literals, consider the following code:

```
121 a = [1, 2]
122 a.each {|e| puts e}
123 a << 3
```

The array literal at line 121 involves no method calls, and therefore, cannot be captured at runtime. Instead, we generate type constraints for the elements by iterating all the elements at each method call since it is the first method call on the array object. Then, Rubydust sets a flag that indicates the elements have been inspected so that subsequent method calls will generate constraints only for the elements directly affected by the calls. For instance, $\ll$ method cosntrain the newly added element 3 to be a subtype of Numeric.

We store the instantiated parameters in the instance, so that we can substitute them in when we look up a method signature. If there are constraints on the type parameters in the annotation, they are also instantiated at this time and stored in the instance. For methods that are polymorphic, we instantiate their type parameters with fresh type variables at the call.

4.1.3 Instrumenting Unannotated Classes

This section describes in more detail how we dynamically instrument unannotated classes and how type constraints are generated by the instrumented code. Figure 4.3 shows a code snippet of the proxy class. The ProxyObject class is defined as a subclass of the BlankSlate class, an empty class where all methods are completely removed.[1] (Notice that it is not an Object in Ruby's stardard sense because it no longer has the minimum set of methods required as an Object.) Some internal operations for proxy are defined in Proxyness module which is included (or extended) by ProxyObject class. Proxyness is a separate module to make potential future extensions easier. These operations include a means to recognize itself as a ProxyObject (line 105) and to retrieve the actual object that is wrapped by the current proxy (line 106).

As shown in lines 102, 103, and 104, wrapped objects $v : \tau$ are implemented as instances of a class ProxyObject with three fields: the object that is being wrapped, its type, and the *owner* of the ProxyObject, which is the instance that was active when the ProxyObject was created. When a method is invoked on a Proxy, the object's method_missing method will be called (shown in line 112); in Ruby, if such a method is defined, it receives calls to any undefined methods. Here method_missing does a little work to memoize the current proxy as the dispatcher for the object (explained more below) and then redirects the call to the wrapped object. The former is explained further later.

[1] http://snippets.dzone.com/posts/show/1873

```
def self.add_method_missing(klass) # self.x means x is a class method
    ...
  klass.class_eval(:define_method, "__method_missing") do |mname, blk, *args|
    proxy = self.instance_variable_get(:@__dispatcher)
    # constrain caller's fields (i.e., fields of proxy.owner)
    # constrain and wrap callee's arguments
    # wrap callee's fields
    ret = send(:"__orig_#{mname}", *proxy_args, &blk) # dispatches the orig. method
    # constrain caller's fields
    # constrain and wrap callee's return
    # wrap caller's fields
    return ret # returning the wrapped return value
  end
end

def self.patch_class(klass)
    ...
  add_method_missing(klass)
    ...
  klass.methods.each { |mname|
    klass.send(:class_eval, <<-EOS
      alias :"__orig_#{mname}" :"#{mname}"
      def #{mname}(*args, &blk)
        ret = __method_missing(:"#{mname}", blk, *args)
        return ret
      end
      EOS
    ) }
end
```

Figure 4.4: Code snippet of uannotated class instrumentation

To implement the wrapping (with ProxyObject) 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 a custom method_missing (named __method_missing to avoid name clashes) 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.

Figure 4.4 shows part of the code for patching a class. At line 141, we patch the class klass by calling add_method_missing method, which is defined in line 124. We used define_method to dynamically define methods, as shown in line 126. Note that it is possible to use a eval or one of its variants for defining the method __method_missing. However, it is tricky to obtain the current method name and to write code that is "debuggable." (Ruby does not provide a useful debugging information when such dynamic features are involved.) Fortunately, Ruby includes a feature for defining a method dynamically without losing debugging information via define_method. Since it is a private method, we had to use send, which allows one to bypass the access control set by the programmer. Notice that, because Ruby prohibits passing another block to a block, we converted any block argument into an explicit argument and passed it as the second argument blk to __method_missing method, as shown in line 126 and 147. (The first argument mname is a string and contains the name of the original method being called.)

During the invocation of __method_missing, we perform all of the constraint generation and wrapping on entry to and exit from a original method, according to

Figure 2.1. This is shown lines 128-134. 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 ProxyObject that most recently dispatched a call to this object; we set the field whenever a ProxyObject is used to invoke a wrapped-object method, as previously shown in line 113 of Figure 4.3. Also recall that each ProxyObject has an owner field, which was set to **self** at the time the proxy was created (line 117 of Figure 4.3). Since we wrap arguments and fields whenever we enter a method, this means all ProxyObjects 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 ProxyObject, and then finding the owner of that ProxyObject (line 127).

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_variable_get (:@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 similar features to break the normal access rules for fields (as we do!).

4.1.4 Instrumenting Annotated Classes

Similarly to 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 2.1.

We had to specially patch Array and Hash. For example, Array's map! replaces its contents with results of a mapping function that is passed to the method as a block argument. This means that the existing type variables no longer correspond to the values in that Array. Therefore, we replace the existing type variable with a fresh variable. In fact, it is typical that any method whose name ends with ! has a potential to change its type since it is Ruby's convention to name destructive methods with !. Currently, we manually inserted the code that replaces the type variable for such methods, but it is possible to detect such methods automatically, which we may implement in future.

We support intersection types for methods, as we introduced in Section 4.1.1. 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.)

Choosing the right "arm" for an intersection type is quite interesting. We basically look up the concrete types of the actual arguments and compare them to the formal argument types using Ruby's standard subclassing relation. In the case of polymorphic types, we compare the types *after* temporarily instantiating the type parameters with bound variables. Notice that we do not make use of the instantiated types for generating type constraints until we are sure that it is the chosen type. Structural types are the most interesting because they may involve nested types which seem to complicate resolving interesction types at first glance. However, we have found that, in every case we looked at, no two structural types should overlap and it is very unnatural to write a Ruby program that has a complicated intersection type in which structural types have same method names but with different types. Thus, we just rely on respond_to? check on the arguments to see if they have corresonding method names.

4.1.5 Runtime

At runtime, both annotated and unannotated code are instrumented and executed by Rubydust. This patched code perform operations that are annotated (in base_types.rb) which may lead to cycles in the process. To eliminate the infinite recursion, we inserted switches in the instrumentation so that operations from the patched code never come back into itself. To avoid too much overhead, we *unpatch* all instrumented code prior to constraint solving so it uses native Ruby code for basic operations which are much faster than their patched versions.

4.1.6 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 Chapter 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, which type variables replaced by their solutions, are consistent.

For example, let us consider the bar method in Figure 2.2. First, Rubydust finds a solution for the return type from the constraint, $\mathsf{Numeric} \leq \alpha_{bar\_ret}$, from which we obtain Numeric. Next, Rubydust solves the argument type x. Although there are two constraints involving α_x, only the constraint, $\alpha_x \leq [\mathsf{qux}\colon () \to ()]$, has the upper bound, from which we find the solution for the arugment x. Assuming that all constraints are satisfiable, the solution for the method bar is $([\mathsf{qux}\colon\ () \to ()]) \to \mathsf{Numeric}$.

Current Rubydust uses the Ruby Graph Library (RGL)[2] for computing the transitive closure. However, we realize that this is not the best solution for implementing a constraint solver because we lose contextual information as types are merged. Consequently, our error messages, in the presence of type inconsistencies,

[2] http://rgl.rubyforge.org

contain almost no debugging information. Improving this issue remains future work.

4.1.7 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 as annotated or unannotated; 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** as its methods are rarely invoked. Third, for soundness, we would need to treat global variables similarly to instance and class fields, we but do not currently support type annotations for them.

Fourth, Ruby includes looping constructs, and hence there are potentially an infinite number of paths through a method body with a loop. However, we manually inspected the code in our benchmarks (Chapter 5) and found that types are in fact invariant across loops. Thus, we can find sound types by exploring all paths through the loops just once. Note that looping constructs in Ruby actually take a code block—essentially a first-class method—as the loop body. If we could assign types to 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.

Finally, as we mentioned earlier, we do not support annotations on some low-level classes. 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.

4.2 Rubydust Framework

To run Rubydust, the user executes the command "rubydust *test*," where *test* is a file that includes a suite of tests for the program of interest. Currently, Rubydust only supports the standard Ruby unit test framework in a limited fashion because it involes a complicated code base which tends to cause conflicts with our runtime instrumentation of essential components in the core library. To avoid this problem, we provide our own testing framework which includes a minimum set of testing capabilities without introducing conflicts with the runtime instrumentation. Full support for the standard Ruby unit test framework remains as future work.

4.2.1 Inputs

Using the built-in testing framework is straightforward, as illustrated with the example in Figure 4.5. For classes with annotated types, the programmer adds a call to Rubydust's use_types method (shown in line 154). For each class whose types should be inferred, the programmer adds a call to Rubydust's infer_types method during the class definition (line 160).

```
153 class A
154   use_types()                    # A is an annotated class
155   typesig(”’+’ : (Numeric) → Numeric”)
156   def +(x); x * 2 end
157 end
158
159 class B
160   infer_types ()                 # B is an unannotated class
161   def foo(x) x.to_s end
162   def bar(y) y + 3 end
163 end
164
165 class TC
166   include Rubydust::RuntimeSystem::TestCase
167   def test_1 ()
168     b = B.new; b.foo(”S”); b.bar(A.new)
169   end
170   ...
171 end
```

Figure 4.5: Using Rubydust

```
172 class B
173   typesig(”foo : [to_s : () → String] → String”)
174   typesig(”bar : [+ : (Numeric) → Numeric] → Numeric”)
175 end
```

Figure 4.6: Type signatures generated by Rubydust

Rubydust's testing framework uses reflection to keep a record of test cases and run them without manual patching of the test code. The only requirement is to include Rubydust::RuntimeSystem::TestCase into the test classes instead of inheriting Ruby's Test::Unit::TestCase, as shown in line 166.

```
class TC
  include Rubydust::RuntimeSystem::TestCase
  def test_1
    d = D.new
    d.bar(1); d.baz(); d.qux(); d.f()
  end
end
```

Figure 4.7: Test case for class D

4.2.2 Output

The output for Figure 4.5 is shown in Figure 4.6. The types Rubydust generates may include both structural types and nominal types. For example, foo takes an object with a to_s method, which itself returns a String, and returns a String. Similarly, bar takes an object with a + method, which takes and returns a Numeric, and returns a Numeric.

In case of inconsistent type constraints, Rubydust generates an error message. For example, Figure 4.7 shows a possible test case for the class D from Figure 2.3. Notice that the given order makes the test case to go through; yet the code is not type safe if d.f is called before d.baz. Although the test case will finish without any runtime error, Rubydust complains that there is a type inconsistency as shown below.

```
[ERROR] subtyping failed: String !<: Numeric
```

This is because, by calling d.qux first, a String can be flown into the field @f which is determined to be a Numeric from the training run. Our future work includes improving the error messages to be more user friendly by remembering the context in which the error has occured.

Chapter 5

Experiments

We ran Rubydust on five small programs obtained from RubyForge. We used a 2.5Ghz dual core processor with 4GB of memory running Mac OS X (Snow Leopard). Figure 5 tabulates our results. The column headers are defined at the bottom of the figure. The table lists the programs and shows the program size in lines of code (via SLOCcount), the number of test cases distributed with the benchmark, the method coverage and line coverage from the test cases (line coverage computed by rcov[1]), the number of manual changes made, and performance measures for each benchmark program. The only manual changes were inserting calls to infer_types . Testing code was excluded when calculating the lines of code, number of methods, and manual changes made. Rubydust found one type error, which we discuss below. As none of the test suites ensures complete coverage, we manually inspected the program code and confirmed that, for the methods that were covered, the inferred type annotations are correct.

[1] http://eigenclass.org/hiki/rcov

	LOC	TC	MCov	LCov(%)	P(#)	OT(s)	RT(s)	ST(s)
ministat-1.0.0	96	10	11 / 15	74.7	1	0.04	13.99	75.30
finitefield-0.1.0	103	9	12 / 12	98.0	1	0.00	2.48	0.87
Ascii85-1.0.0	105	7	2 / 2	95.2	1[†]	0.04	47.76	0.14
hebruby-2.0.2	178	19	20 / 26	80.8	1	0.04	27.20	29.59
StreetAddress-1.0.1	767	1	33 / 44	78.9	2	0.54	5.36	47.55

TC - test cases MCov - method coverage / total # of methods
LCov - line coverage P - manual edits
OT - original running time RT - Rubydust running time ST - constraint solving time

[†] Because the test suite was originally written in Spec, another Ruby testing framework that we do not support at this time, we manually translated it to a typical Ruby test.

Figure 5.1: Results

5.1 Performance

We split Rubydust's running time 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.

Another reason is that Rubydust currently wrap values and generate type constraints during invocations of annotated methods. Notice that, since the methods are annotated, there is no need for type inference within the calls except the blocks which may contain code from their callers. Unforunately, this is unavoidable at this point because we do not capture blocks, and therefore, cannot tell the tool to bypass the patched code and directly call the original code except for the block argument whose bindings are at the caller. Our initial investigation tells us that this is a major performance bottleneck.

Lastly, our handling of Array and Hash may slow down the performance dramatically. For example, every element in any newly created Array or Hash has to be iterated at some point in its lifetime in order for their types to be considered. Operations such as map or select introduces a new object out of the existing object; and therefore, they may cause the slowdown the most. This can be improved using dynamic lookup of type annotations and/or concrete types of the elements as these operations take place. 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. We expect this solving time would decrease dramatically if we exported the constraints to a solver written in a different language.

5.2 Inferred Types

We now describe the benchmark programs and show some example inferred type output by Rubydust. Basic notations are as follows. The bottom type $\perp$ is denoted by .! whereas the top type $\top$ is denoted by .? in our annotation language. It is understood that, if a return type has the bottom type .!, it is always the case that the corresponding method is not seen during the training run. (In fact, it would not make sense to return a bottom type unless it never returns to the caller.)

The top type .? usually indicates that we were unable to find any constraints on the type. If an argument has the top type, it may indicate that the method is not observed during the test run or anything is valid.

Some container classes such as Array may contain .? as the actual type argument. This occurs when Rubydust fails to find any constraints on the elements. This happens frequently when an Array literal is used to create an instance and no method is invoked on it. In future versions of Rubydust, we will enforce type constraint generation for such objects at method entries and exits which will eliminate .? types unless the container is empty.

It is typical to obtain structural types for argument types and nominal types for return types, at least in our experience with running Rubydust on the benchmark programs as well as on small Ruby scripts. Since it is natural that methods are invoked on the arguments, structural types are likely. However, return types depend on the concrete type of the objects being returned. Even if the return value is one of the arguments, because we generate a constraint for the argument (usually) involving a concrete type, we end up with that type. We believe this is still viable for most Ruby programmers who enjoy dynamic typing because structural types depict this dynamic behavior precisely.

Ministat

Ministat generates simple statistical information on numerical data sets. The complete types inferred by Rubydust are shown below:

```
class MiniStat::Data;
  typesig("mode :() → Numeric")
  typesig("mean : ([each : () → .?;size : () → Numeric;inject : (.?) → .?]) → Numeric")
  typesig("harmonic_mean :() → .!") # no test case for this method
  typesig("variance :() → Numeric")
  typesig("outliers :() → Array<.?>")
  typesig("std_dev :() → Numeric")
  typesig("partition :(Numeric, [each : () → Array<Numeric>]) → Hash<.?,.?>")
  typesig("median :
          ([sort! : () → Array<Numeric>; size : () → Numeric;
            '[]' : (Numeric) → Numeric; '==' : (Object) → Boolean]) → Numeric ")
  typesig("q1 :() → Numeric")
  typesig("iqr :() → Numeric")
  typesig("geometric_mean :() → .!") # no test case for this method
  typesig("to_s :() → .!")            # no test case for this method
  typesig("initialize :([collect : () → Array<.?>]) → Boolean")
  typesig("q3 :() → Numeric")
  typesig("data :() → .!")            # no test case for this method
end
```

Out of 15 total methods, four methods were not given a type because there were no test cases that covered them. Two of the 11 methods that are given types have structural types for the arguments. For example, the median 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 four 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

Finitefield, another mathematical library, provides basic operations on elements in a finite field. Rubydust successfully inferred types for all the methods, thanks to the test suite of the program which trained all methods. The complete result is given below.

```
class FiniteField ;
  typesig("reduce :(Numeric) → Numeric")
  typesig("binary_mul : (['&' : (Numeric) → Numeric; '≫' : (Numeric) → Numeric;
                          '==' : (Object) → Boolean], Numeric) → Numeric ")
  typesig("multiply :(Numeric, Numeric) → Numeric")
  typesig("polynomial :() → Numeric")
  typesig("multiplyWithoutReducing :(Numeric, Numeric) → Numeric")
  typesig("divide : (Numeric,
                      ['>' : (Numeric) → Boolean; '≪' : (Numeric) → Numeric;
                       '&' : (Numeric) → Numeric; '≫' : (Numeric) → Numeric;
                       '==' : (Object) → Boolean; '^' : (Numeric) → Numeric]) →
                     Numeric ")
  typesig(" binary_div :
          (['&' : (Numeric) → Numeric; '≫' : (Numeric) → Numeric;
            '==' : (Object) → Boolean; '^' : (Numeric) → Numeric],
           ['≪' : (Numeric) → Numeric; '≫' : (Numeric) → Numeric;
            '==' : (Object) → Boolean]) → Array<.?> ")
  typesig("subtract :(['^' : (Numeric) → Numeric], Numeric) → Numeric")
  typesig(" inverse :
          (['>' : (Numeric) → Boolean; '≪' : (Numeric) → Numeric;
            '&' : (Numeric) → Numeric; '≫' : (Numeric) → Numeric;
            '==' : (Object) → Boolean; '^' : (Numeric) → Numeric]) →
          Numeric ")
  typesig(" initialize :
          (Numeric,
           ['>' : (Numeric) → Boolean] and ['≪' : (Numeric) → Numeric] and
           ['≫' : (Numeric) → Numeric] and ['&' : (Numeric) → Numeric] and
           Object and ['^' : (Numeric) → Numeric]) → Numeric ")
  typesig("degree :
          (['≫' : (Numeric) → Numeric;'==' : (Object) → Boolean]) →
          Numeric ")
  typesig("add :(['^' : (Numeric) → Numeric], Numeric) → Numeric")
end
```

In this benchmark program, all methods are covered by the test cases. Several methods have structural types for the arguments, all of which resemble Numeric. For example, the inverse method requires an object that has &, $>$,$\ll$,$\gg$, $==$, and ˆ. As above, we can see exactly which methods are required of the argument.

Notice that one drawback of retaining structural types, as opposed to simplifying them all to nominal types, may arise if Rubydust cannot simplify two types in a user-friendly way. For example, the initialize method takes a Numeric argument and another argument that resembles a Numeric *and* a Object. Of course, this type may be thought of as Numeric, but we cannot definitely decide so because it is not required to be a Numeric. It is, however, required to be an Object and have the five methods. It is possible to ignore Object since most objects in Ruby are Object although it is not always the case (as with BlankSlate).

Although many structural types are similar, most arguments' types are not precisely identical. This is interesting because all have slightly different requirements as the arguments.

Ascii85

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 three 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 take an arbitrary non-integer, since its to_i method is invoked (which would not be necessary for an integer).

Hebruby

Hebruby is program that converts Hebrew dates to Julian dates and vice versa. The complete type annotations obtained by Rubydust are shown below.

```
class Hebruby::HebrewDate;
  typesig("hy :() → .!")
  typesig("day :() → Numeric")
  typesig("heb_month_name :() → String")
  typesig(" convert_from_julian :() → Numeric")
  typesig("month :() → Numeric")
  typesig("convert_from_hebrew :() → Numeric")
  typesig("hd= :(.?) → .!")
  typesig("year :() → Numeric")
  typesig("jd :() → Numeric")
  typesig("month_name :() → String")
  typesig("hm= :(.?) → .!")
  typesig("hd :() → .!")
  typesig(" initialize :(.?) → Numeric")
  typesig("hy= :(.?) → .!")
  typesig("hm :() → .!")
  typesig("heb_date :() → String")
  typesig("heb_year_name :() → String")
  typesig("heb_day_name :() → String")
end

class << Hebruby::HebrewDate; # meta class of Hebruby::HebrewDate
  typesig("month_days :
          (['+' : (Numeric) → Numeric; '−' : (Numeric) → Numeric;
            '*' : (Numeric) → Numeric], ['==' : (Object) → Boolean]) → Numeric ")
  typesig("heb_number :
          (['/' : (Numeric) → Numeric; '>' : (Numeric) → Boolean;
            '%' : (Numeric) → Numeric; '<' : (Numeric) → Boolean;
            '==' : (Object) → Boolean]) → String ")
  typesig(" days_in_prior_years :(['−' : (Numeric) → Numeric]) → Numeric")
  typesig("year_months :(['*' : (Numeric) → Numeric]) → Numeric")
  typesig("to_jd :
          (['+' : (Numeric) → Numeric; '−' : (Numeric) → Numeric;
            '*' : (Numeric) → Numeric], Numeric,
           ['+' : (Numeric) → Numeric; coerce : (Numeric) → Array<Numeric>]) →
          Numeric ")
  typesig("year_days :
          (['+' : (Numeric) → Numeric; '−' : (Numeric) → Numeric;
            '*' : (Numeric) → Numeric]) → Numeric ")
  typesig("jd_to_hebrew :
          (['+' : (Numeric) → Numeric; '>' : (Numeric) → Boolean;
            coerce : (Numeric) → Array<Numeric>; '−' : (Numeric) → Numeric;
            '>=' : (Numeric) → Boolean]) → Array<.?> ")
  typesig("leap? :(['*' : (Numeric) → Numeric]) → Boolean")
end
```

Hebruby::HebrewDate class has 12 out of 18 methods that are typed (covered by test cases), all of which have concrete types only. This is reasonable because the class represents a Hebrew date and most of the methods are access readers or writers for the subcomponents.

Its metaclass has eight methods (the former's class methods), all of which have arguments of structural types. For example, the leap method only requires an 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.

StreetAddress

Finally, *StreetAddress* is a tool that normalizes U.S. street address into different subcomponents. The complete type annotations generated by Rubydust are listed below.

```
class << StreetAddress::US; # meta class of StreetAddress :: US
  typesig (” parse_address  :( String ) → StreetAddress::US::Address” )
  typesig (”parse  :( String ) → StreetAddress::US::Address” )
  typesig (”  fips_state   :()  → .!” )
  typesig (” normalize_state  :  ([length  :  ()  → Numeric;upcase : () → String]) → String” )
  typesig (” normalize_address  :  (StreetAddress :: US::Address)
                                    → StreetAddress::US::Address ” )
  typesig (”  normalize_street_type   :
             ([ capitalize   :  ()  → String; eql? :  (Object)  → Boolean;
               downcase! :  ()  → String; hash :  ()  → Numeric]) → String ” )
  typesig (”  normalize_directional   :
             ([length  :  ()  → Numeric;upcase : () → String]) → String ” )
  typesig (”  parse_intersection   :( String )  → StreetAddress::US::Address” )
  typesig (”state_name  :()  → .!” )
end

class  StreetAddress :: US::Address;
  typesig (”  state_fips   :()  → .!” )
  typesig (” prefix=  :(String )  → String” )
  typesig (” prefix2=  :(.?)  → .!” )
  typesig (” city   :()  → String” )
  typesig (” street_type2   :()  → String” )
  typesig (” street=  :(String )  → String” )
  typesig (”postal_code   :()  → String” )
  typesig (” suffix2=  :(.?)  → .!” )
  typesig (” prefix2   :()  → .!” )
  typesig (” prefix   :()  → String” )
  typesig (” postal_code_ext=  :(.?)  → .!” )
  typesig (" street_type = :( String ) → String")
  typesig (" street  :() → String")
  typesig (" suffix2  :() → .!")
  typesig (" street2 = :( String ) → String")
  typesig (" state= :( String ) → String")
  typesig (" postal_code_ext  :() → .!")
  typesig (" unit= :( String ) → String")
  typesig (" street_type  :() → String")
  typesig (" suffix = :( String ) → String")
  typesig (" street2  :() → String")
  typesig ("state_name :() → .!")
  typesig (" state  :() → String")
  typesig ("  intersection ?  :() → Boolean")
  typesig ("  unit_prefix = :( String ) → String")
  typesig (" unit  :() → String")
  typesig (" suffix  :() → String")
  typesig ("number= :(String) → String")
  typesig ("  unit_prefix  :() → String")
  typesig (" to_s  :() → .!")
  typesig ("  initialize  :
          ([ keys  :  () → Array<Symbol>;
              '[]'  :  (.?) →
                     Object and [ capitalize  :  () → String; length :  () → Numeric;
                     downcase! :  () → String; eql? :  (Object) → Boolean;
                      capitalize !  :  () → String; hash :  () → Numeric;
                     upcase :  () → String; '==' : (Object) → Boolean;
                     gsub! :  (Regexp or String ) → String]
           ]) → Array<Symbol>
        ")
  typesig (" city= :( String ) → String")
  typesig (" street_type2 = :( String ) → String")
  typesig (" postal_code= :( String ) → String")
  typesig ("number :() → String")
end
```

Out of 44 methods, 33 methods are covered by the test suite and are given the type annotations. It is interesting that many class methods in StreetAddress :: US return an instance of StreetAddress :: US::Address class. This is one advantage of return types resolving into nominal types because, otherwise, it would list structural types that we are currently trying to infer. Notice that most methods in

StreetAddress :: US::Address have concrete types (mostly String) in their types because, as with the Hebruby::HebrewDate class in *Hebruby*, this class represents a US address and most of the methods access readers and writers for the address' subelements. It is, therefore, reasonable that they take or return Strings.

Chapter 6

Future Work

In this section we discuss other features that are currently not supported by Rubydust, but could improve the precision and performance of the tool. We also discuss additional experiments that could be carried out to measure the tool's precision and performance.

6.1 Other Features

We believe that there are other features that are useful (but not essential to our results) and reasonable to implement and incorporate into Rubydust.

6.1.1 Handling Blocks

As discussed earlier, we do not infer types for blocks because Ruby does not provide a straightforward mechanism to capture their arguments or return values. Our experience showed, however, that the types inferred are sound for the benchmark programs even though not handling blocks correctly may potentially be unsound. Regardless, it is desirable to infer types for blocks for subsequent analysis such as type checking and for documentation purposes, as well. Here we describe two possible solutions to handling blocks in Rubydust.

In Ruby, a block is passed in as an additional argument that can be either implicitly or explicitly declared in a method definition. A block can be invoked by a **yield** construct inside the callee's body or by invoking the call method of a Block object, to which Ruby converts the explicitly declared block argument. Unfortunately, we cannot intercept **yield** calls because it is not a method call, but rather is a language construct. One way to solve this issue is to syntactically replace all occurrences of **yield** with a method call. For example,

```
def foo(*args, &blk)
  ...
  yield
  ...
end
```

would be transformed into the following:

```
def foo(*args, &blk)
  ...
  __yield (*args) # it is invoking blk with the arguments, args.
  ...
end
```

Note that **yield** does not need explicit arguments for itself, in which case it takes the arguments from the callee (which is foo). Therefore, this must be taken into account when transforming the code, as shown above. Once the transformation is made, we can treat __yield as a typical method call and collect type information for the arguments and the return as usual. The drawback of this solution is that the transformed code must be written to disk or memory and loaded back to memory for an execution.

Another solution for the **yield** construct is to capture blocks at the callee's entry and exit. That is, instead of capturing the arguments and returns for the block, we would provide an instrumented block at the callee's entry. For example, at line 1, blk is wrapped with an instrumented block that would generate constraints for the arguments and the return and wrap them at the entry and exit, respectively.

Finally, we need to extend Rubydust to treat invocations of call specially for capturing the actual call to the high-order function, which we believe we could do by patching the Block, Proc, and Method classes.

6.1.2 Polymorphic Type Inference

Rubydust currently does not infer polymorphic or intersection types. Polymorphic type inference generally requires a more sophisticated algorithm and may decrease the performance of the tool. However, polymorphic types are essential part of Ruby types because they represent any container-like classes such as Set that many Ruby programmers write. Therefore, this feature may be supported in future.

Intersection types are similar to polymorphic types except that they are induced by type case expressions. We believe that type case expressions can be captured in a limited fashion although not perfectly. By doing so, we may precisely observe different type cases and infer intersection types accordingly. Alternatively, we can also introduce a construct that would indicate methods whose types are to be inferred as intersection types. Then, we can treat each method call as a separate set of constraints to solve, from which we gather different solutions into an intersection type for that method.

6.1.3 Dynamic Type Checker

Currently, Rubydust does not provide type checking for annotated code. Rubydust can find type errors *after* solving the type constraints, but as we saw earlier, constraint generation and solving has significant overhead. In a future version of Rubydust, we plan to include a type checker that compares the actual types of actual arguments and return against those of the annotation at each method; therefore, we may catch type errors as early as possible.

6.1.4 Capturing Dynamic Method Creation

In Ruby, it is quite useful to define methods at runtime using eval or define_method, as we did in our own code. Although Rubydust handles the top level evals, which execute at class definition time, it does not handle methods that are created after instrumentation. (Recall that Rubydust has a single point, prior to an entry point to a test run, where it patches all methods at once.) Fortunately, in Ruby, it is possible to establish a callback method for method creations and capture the newly created methods. In a future version of Rubydust, we plan to exploit this feature and instrument freshly introduced methods as well.

6.2 Array and Hash

There are several potential candidates for optimizations. Foremost, Array and Hash create overhead because, in order to generate type constraints for the type variables, all the elements must be inspected. This is especially a problem when a method call introduces another Array or Hash object from an existing one. This is because all the elements in the new object have to be iterated over. Consider the following code:

```
a = [...]
b = a.select {|e| e % 2 == 0}
 ...
c = b.to_ary
```

Because the result of the select method is a new Array, the elements of b must be iterated through at some point in the program. Likewise, c is a newly introduced Array whose elements will be iterated through at a future point. This obviously causes unwanted overhead because now we have to scan three Arrays of the same type. The overhead can be partially eliminated by utilizing the type annotations for the select and to_ary methods—i.e., they both return an Array of same type; and thus, we need not iterate over their contents to find out their types. Notice that this is a conservative approach because the elements in the new object are a subset of elements in the existing object.

6.3 Other Experiments

There are other experiments that we can perform in order to measure how Rubydust can be used in practice or for other purposes aside from static typing.

6.3.1 Ruby on Rails

Ruby on Rails is a popular web application framework written in Ruby. We have previously shown that Rails applications also suffer from typical Ruby type errors in addition to the errors specific to Rails. We also showed that many of the type errors can also be automatically detected if we transform the Rails code into pure Ruby code and apply DRuby on that code [3]. Using Rubydust, we believe we can do the same without parsing and transforming the code.

One of the difficulties with typing Rails programs is to not analyze the Rails framework because it is too big and uses highly dynamic features that cannot be analyzed easily. Rubydust already has this advantage because it will run, but bypass the analysis of classes that are not of our interest. Unlike Ruby programs, however, Rails frequently uses Hash to store structured information such as the sessions and database table rows. Thus, Rubydust needs to be extended to support more fine-grained typing for Hash, and possibly Tuple as well, to provide useful types to the user.

6.3.2 Fewer Test Cases

Although we used all supplied test cases in the benchmark programs, it is possible to write fewer test cases that would cover the same or more paths in the program. By doing this, we may show that Rubydust, in practice, often does not need an exaustive number of paths in order to infer correct types.

6.3.3 Beyond Types

Lastly, we believe that Rubydust's framework is not limited to the traditioanl type systems. It is possible, for example, to allow values to be observed instead of their types, to gather more precise information as in dependent type systems. We imagine this could be a interesting future direction.

Chapter 7

Related Work

There has been significant interest in the research community in bringing static type systems to dynamic languages. One recent focus area has been developing ways to mix typed and untyped code, e.g., quasi-static typing [25], contracts applied to types [28], gradual typing [24], and hybrid types [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 [10, 12, 3, 2, 17, 19], Python [7, 23, 4], and JavaScript [5, 26], 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 [10, 12, 3].

There has also been work on type inference for Scheme [29], 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 [21]. 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 2.2, 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 [30]. 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 [20]. 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 [15] resembles soundness for *Mix*, a static analysis system that mixes type checking and symbolic execution [16]. 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.

Chapter 8

Conclusion

In this thesis we presented a new technique, constraint-based dynamic type inference, that infers types based on dynamic executions of the program. 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 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.

Bibliography

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

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

[3] J. D. An, A. Chaudhuri, and J. S. Foster. Static Typing for Ruby on Rails. In *ASE*, 2009.

[4] D. Ancona, M. Ancona, A. Cuni, and N. Matsakis. RPython: Reconciling Dynamically and Statically Typed OO Languages. In *DLS*, 2007.

[5] C. Anderson, P. Giannini, and S. Drossopoulou. Towards Type Inference for JavaScript. In *ECOOP*, 2005.

[6] S. Artzi, A. Kiezun, D. Glasser, and M. D. Ernst. Combined static and dynamic mutability analysis. In *ASE*, 2007.

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

[8] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In *ICSE*, 2008.

[9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3), 2007.

[10] M. Furr, J. D. An, J. S. Foster, and M. Hicks. Static Type Inference for Ruby. In *OOPS Track, SAC*, 2009.

[11] M. Furr, J. D. An, J. S. Foster, and M. Hicks. The Ruby Intermediate Language. In *Dynamic Language Symposium*, 2009.

[12] M. Furr, J.-h. D. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA*, 2009.

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

[14] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In *ISSTA*, 2006.

[15] J. hoon (David) An, A. Chaudhuri, J. S. Foster, and M. Hicks. Dynamic inference of static types for ruby. Technical Report CS-TR-4965, University of Maryland, 2010.

[16] Y. P. Khoo, B.-Y. E. Chang, and J. S. Foster. Mixing type checking and symbolic execution. In *PLDI*, 2010.

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

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

[19] J. Morrison. Type Inference in Ruby. Google Summer of Code Project, 2006.

[20] J. W. Nimmer and M. D. Ernst. Invariant Inference for Static Checking: An Empirical Evaluation. In *FSE*, 2002.

[21] P. Rapicault, M. Blay-Fornarino, S. Ducasse, and A.-M. Dery. Dynamic type inference to support object-oriented reenginerring in Smalltalk. In *ECOOP Workshops*, 1998.

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

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

[24] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.

[25] S. Thatte. Quasi-static typing. In *POPL*, 1990.

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

[27] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2004.

[28] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *POPL*, 2008.

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

[30] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE*, 2006.