ABSTRACT

Title of dissertation: Combining Static and Dynamic Typing in Ruby

Michael Furr
Doctor of Philosophy, 2009

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

Many popular scripting languages such as Ruby, Python, and Perl are dynamically typed. Dynamic typing provides many advantages such as terse, flexible code and the ability to use highly dynamic language constructs, such as an `eval` method that evaluates a string as program text. However these dynamic features have traditionally obstructed static analyses leaving the programmer without the benefits of static typing, including early error detection and the documentation provided by type annotations.

In this dissertation, we present Diamondback Ruby (DRuby), a tool that blends static and dynamic typing for Ruby. DRuby provides a type language that is rich enough to precisely type Ruby code, without unneeded complexity. DRuby uses static type inference to automatically discover type errors in Ruby programs and provides a type annotation language that serves as verified documentation of a method's behavior. When necessary, these annotations can be checked dynamically using runtime contracts. This allows statically and dynamically checked code to safely coexist, and any runtime errors are properly blamed on dynamic code. To

handle dynamic features such as eval, DRuby includes a novel dynamic analysis and transformation that gathers per-application profiles of dynamic feature usage via a program's test suite. Based on these profiles, DRuby transforms the program before applying its type inference algorithm, enforcing type safety for dynamic constructs. By leveraging a program's test suite, our technique gives the programmer an easy to understand trade-off: the more dynamic features covered by their tests, the more static checking is achieved.

We evaluated DRuby on a benchmark suite of sample Ruby programs. We found that our profile-guided analysis and type inference algorithms worked well, discovering several previously unknown type errors. Furthermore, our results give us insight into what kind of Ruby code programmers "want" to write but is not easily amenable to traditional static typing. This dissertation shows that it is possible to effectively integrate static typing into Ruby without losing the feel of a dynamic language.

Combining Static and Dynamic Typing in Ruby

by

Michael Furr

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
Doctor of Philosophy
2009

Advisory Committee:
Professor Jeffrey S. Foster, Chair/Advisor
Professor Michael Hicks
Professor Neil Spring
Professor Vibha Sazawal
Professor Bruce Jacob, Dean's Representative

Dedication

To my wife, Laura.

Table of Contents

List of Figures

Chapter 1

Introduction

Over the last decade a new wave of programming languages such as Perl [81], Python [79], and Ruby [74] have gained popularity. These languages are distinguished by their strong support for regular expressions and string manipulation, terse syntax, comprehensive libraries, and expressive language constructs. Combined, these features aim to make solving common programming tasks as easy as possible. In particular, these features ease the development of prototypes, where producing a implementation *quickly* is often more important than producing a program that is correct for every possible input. These languages are also dynamically typed, meaning that no program is ever prevented from being executed. If a type error occurs at runtime, such as evaluating **true** + 3, the program will be aborted with an error message. However, if this expression is not evaluated, no error will be emitted. This can be advantageous early in the development cycle. For example, a programmer may wish to run her program to observe the layout of a GUI component, ignoring the fact that clicking on a button would cause the program to crash.

In contrast, statically typed languages attempt to detect potential type errors

before the program is ever executed. In order to be decidable, static type systems must approximate which programs may cause a type error at runtime. For example, **if** (is_prime 10) **then true** + 3 **end** would be considered ill-typed by most type systems despite the fact that this program will never produce a type error at runtime as the primality test for 10 will always fail. Despite rejecting this program, one could argue that the type system is justified in its conservatism: if the true branch was ever taken (e.g., after performing some code refactoring) it would in fact produce a type error, suggesting that this code should be changed nonetheless to make the code more robust.

Ideally, we would like to have the best of both worlds: increased flexibility when developing new features, and additional static checks when the code base matures. In this dissertation, will explore adding an optional, static type system to the Ruby programming language. Our aim is to add a typing discipline that is simple for programmers to use, flexible enough to handle common idioms, that provides programmers with additional checking where they want it, and reverts to run-time checks where necessary.

There are several challenges in applying static typing to Ruby. First, Ruby is a dauntingly large language, with many constructs and complex control flow. Ruby aims to "feel natural to programmers" [69] by providing a rich syntax that is almost ambiguous, and a semantics that includes a significant amount of special-case, implicit behavior. While the resulting language is arguably easy to use, its complex syntax and semantics make it hard to write tools that work with Ruby source code.

Second, since Ruby is dynamically typed, it imposes few restrictions on the kind of code that developers can write, allowing programmers to use any idiom they choose. For example, program variables may be used with different types within the same scope.

Finally, like other scripting languages, Ruby includes a range of hard-to-analyze, highly dynamic constructs, such as an eval method that evaluates a string as program text. While these constructs allow terse and expressive code, they have traditionally obstructed static analysis.

1.1 Thesis

Despite these challenges, in this dissertation we will show:

> *Static typing can be effectively integrated into the Ruby language. A static type system for Ruby can be precise and does not require Ruby developers to significantly modify their programs to be accepted as well-typed. Furthermore, Ruby's dynamic features can be effectively profiled and approximated with statically analyzable constructs.*

As evidence in support of this thesis, we present Diamondback Ruby [1] (DRuby), an extension to Ruby that blends the benefits of static and dynamic typing. DRuby is focused on Ruby, but we expect our advances to apply to many other scripting languages as well. DRuby is available as a free download at `http://www.cs.umd.edu/projects/PL/druby`.

[1] The diamondback terrapin is the official mascot of the University of Maryland

1.2 Contributions

The remainder of this section will sketch our contributions, as presented in the rest of this dissertation.

1.2.1 RIL, a Ruby Analysis Framework

As we embarked on building DRuby, we quickly discovered that working with Ruby code was going to be quite challenging. Thus, in order to enable our other contributions, Chapter 2 presents RIL, a framework that simplifies analyzing Ruby code. The primary motivation for RIL is to provide a representation of Ruby source code that makes it easy to develop source code analysis and transformation tools. RIL includes an extensible GLR parser for Ruby, and an automatic translation into an easy-to-analyze intermediate form. This translation eliminates redundant language constructs, unravels the often subtle ordering among side effecting operations, and makes implicit interpreter operations explicit. RIL also includes several additional useful features, such as an instrumentation library for building dynamic analyses and a dataflow analysis engine.

Combined, these transformations and libraries make writing static analyses with RIL much easier than working directly with Ruby source code. RIL allowed us to build DRuby, but we believe it will be useful to others who wish to build analysis tools for Ruby.

1.2.2 Static Types for Ruby

In Chapter 3, we present DRuby's static type language. DRuby includes union and intersection types [56], nominal and structural object types [1], a self type [16], F-bounded polymorphism [57], tuple types for heterogeneous arrays, flow sensitive types for local variables, and optional and variable arguments in method types. We have proven DRuby's type system sound for a small Ruby-like calculus (Appendix B).

DRuby also includes a surface type annotation syntax. Annotations are required to give types to Ruby's core standard library, since it is written in C rather than Ruby, and for some types that DRuby cannot infer, such as higher order polymorphic types. Annotations are also useful for providing documentation in the form of type signatures to Ruby methods and classes. These annotations are verified using a mix of static and dynamic checks. DRuby's dynamically checked annotations improve upon Ruby's existing dynamic type system by tracking blame, i.e., statically typed code is never blamed for a run-time type error. We believe our annotation language is easy to understand, and indeed it resembles the "types" written down informally in the standard library documentation.

DRuby includes a type inference algorithm to statically discover type errors in Ruby programs, which can help programmers detect problems much earlier than dynamic typing. By providing type inference, DRuby helps maintain the lightweight feel of Ruby, since programmers need not write down extensive type annotations to gain the benefits of static typing.

1.2.3 Profiled-Guided Analysis of Dynamic Features

To handle the dynamic features of Ruby, DRuby includes a novel, profile-guided analysis (Chapter 4). Our approach uses run-time instrumentation to gather per-application profiles of dynamic feature usage from an application's test suite. Based on these profiles, we replace dynamic features with statically analyzable alternatives, adding instrumentation to safely handle cases when subsequent runs do not match the profile. Transformed code can then be analyzed using DRuby's static type inference algorithm to enforce type safety.

We prove that our transformation is *faithful*, meaning it does not change the behavior of a program under its profile, and that transformed programs that pass our type checker never go wrong at run time, except possibly from code that was instrumented with blame tracking (Appendix C).

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

Chapter 2

A Framework for Ruby Analysis

In this chapter, we describe the Ruby Intermediate Language (RIL), an intermediate language designed to make it easy to extend, analyze, and transform Ruby source code. As far as we are aware, RIL is the only Ruby front-end designed with these goals in mind. RIL provides four main advantages for working with Ruby code. First, RIL's parser is completely separated from the Ruby interpreter, and is defined using a Generalized LR (GLR) grammar [78], which makes it much easier to modify and extend. In particular, it was rather straightforward to extend our parser grammar to include type annotations, a key part of adding static types to Ruby (Section 2.2). Second, RIL translates many redundant syntactic forms into one common representation, reducing the burden on the analysis writer. For example, Ruby includes four different variants of `if-then-else` (standard, postfix, and standard and postfix variants with `unless`), and all four are represented in the same way in RIL. Third, RIL makes Ruby's (sometimes quite subtle) order of evaluation explicit by assigning intermediate results to temporary variables, making flow-sensitive analyses like dataflow analysis simpler to write. Finally, RIL makes explicit much of Ruby's implicit semantics, again reducing the burden on the analysis designer. For

example, RIL replaces empty Ruby method bodies by `return nil` to clearly indicate their behavior (Section 2.3).

In addition to the RIL data structure itself, our RIL implementation has a number of features that make working with RIL easier. RIL includes an implementation of the visitor pattern to simplify code traversals. The RIL pretty printer can output RIL as executable Ruby code, so that transformed RIL code can be directly run. To make it easy to build RIL data structures (a common requirement of transformations, which often inject bits of code into a program), RIL includes a partial reparsing module [25]. RIL also has a dataflow analysis engine, and extensive support for run-time instrumentation, which we use to profile highly dynamic features such as `eval` in Chapter 4.

RIL is written in OCaml [45], which we found to be a good choice due to its support for multi-paradigm programming. For example, its data type language and pattern matching features provide strong support for manipulating the RIL data structure, the late binding of its object system makes visitors easier to re-use, and RIL's dataflow and instrumentation libraries can be instantiated with new analyses using functors.

Along with DRuby, RIL has also been used to build DRails, a tool that brings static typing to Ruby on Rails applications [7]. In addition, several students in a graduate class at the University of Maryland used RIL for a course project. The students were able to build a working Ruby static analysis tool within a few weeks. These experiences lead us to believe that RIL is a useful and effective tool for analysis and transformation of Ruby source code. We hope that others will find

RIL as useful as we have, and that our discussion of RIL's design will be valuable to those working with other dynamic languages with similar features.

2.1 An Introduction to Ruby

We begin by introducing some of Ruby's key features. In Ruby, everything is an object, and all objects are instances of a particular class. For example, the literal 42 is an instance of Fixnum, **true** is an instance of TrueClass, and **nil** is an instance of NilClass. As in Java, the root of the class hierarchy is Object.

There are several kinds of variables in Ruby, distinguished by a prefix: local variables `x` have no prefix, object instance variables (a.k.a. fields) are written `@x`, class variables (called `static` variables in Java) are written `@@x`, and global variables are written `$x`.[1]

Figure 2.1 contains some sample code that illustrates many features of Ruby. Local variables are not visible outside their defining scope are never declared, but rather come into existence when they are written to (line 1). It is an error to refer to a local variable before it is initialized (line 3). Since local variables are dynamically typed, their types may change as evaluation proceeds. In Figure 2.1, line 4 writes a string to b, effectively changing its type, so on line 5 it makes sense to invoke the length method on b.

Lines 7–19 define a new class Container with instance methods get and set, class

[1] As an aside, there are a few oddball variables like $1 that are prefixed as a global but are in fact local.

```
1  a = 42      # a in scope from here onward
2  b = a + 3  # equivalent to b = a.+(3)
3  c = d + 3  # error:  d undefined
4  b = "foo"  # b is now a String
5  b.length    # invoke method with no args
6
7  class Container    # implicitly extends Object
8    def get()        # method definition
9      @x             # field read; method evaluates to expr in body
10   end
11   def set(e)
12     @@last = @x   # class variable write
13     @x = e        # field write
14   end
15   def Container.last()    # class method
16     $gl = @@last
17     @@last
18   end
19 end
20 f = Container.new       # instance creation
21 f.set(3)                 # could also write as "f.set 3"
22 g = f.get                # g = 3
23 f.set(4)
24 h = Container.new.get   # returns nil
25 l = Container.last      # returns 3
26 $gl                     # returns 3
27
28 i = {``size'' => 3,
29       ``color'' => ``blue''} # hash literal
30 j = [1, 2, 3]                 # array literal
31 k = j.collect { |x| x + 1} # k is [2, 3, 4]
32
33 module My_enumerable  # module creation
34   def leq(x)
35     (self ⇔ x) ≤ 0     # note method "⇔" does not exist here
36   end
37 end
38
39 class Container
40   include My_enumerable          # mix in module
41   def ⇔(other)                   # define required method
42     @x ⇔ other.get
43   end
44 end
45
46 f ⇔ f            # returns 0
```

Figure 2.1: Sample Ruby code

method last, an instance variable @x, and a class variable @@last. Methods evaluate to the last statement in their body (line 9), and may also include explicit **return** statements. Instances are created by invoking the special new method (line 20). Method invocation syntax is standard (line 21), though parentheses are optional for the outermost method call in an expression (lines 20 and 22). Class methods must be called with the class as the receiver (line 25). Method names need not be alphanumeric; operations such as addition (line 2) are equivalent to method calls (in this case, equivalent to a.+(3)). Unlike local variables, fields are by default initialized to **nil** (line 24); the same is true with class and global variables. As usual, class variables are shared between all instances of a class (line 25).

To elaborate on Ruby's scope rules, class and instance variables are only visible within their defining class. For example, there is no syntax to access the @x or @@last fields of f from outside the class. Local variables are not visible inside nested classes or methods, e.g., it would be an error to refer to b inside of Container. Global variables are visible anywhere in the program (lines 16 and 26).

Like most scripting languages, Ruby provides special syntax for hash literals (line 28) and array literals (line 30) . Ruby also supports higher-order functions, called *code blocks*. Line 31 shows an invocation of the collect method, which produces a new array by applying the supplied code block to each element of the original array. The code block parameter list is given between vertical bars, and, unlike methods, code blocks may refer to the local variables that are in scope when they are created. Using standard syntax as shown, each method may take at most one code block as an argument, and a method invokes the code block using the

special syntax **yield**$(e_1, \ldots, e_n)$. Ruby does have support for passing code blocks as regular arguments, but the syntax is messier, discouraging its use.

Ruby supports single inheritance. The declaration syntax **class** Foo $<$ Bar indicates that Foo inherits from Bar. If no explicit inheritance relationship is specified (as with Container in Figure 2.1), then the declared class inherits from Object. Ruby also includes *modules* (a.k.a. *mixins*) [14] for supporting multiple inheritance. For example, lines 33–37 define a module My_enumerable, which defines an leq method in terms of another method $\Leftrightarrow$ that implements three-way comparison. On lines 39–44, we mix in the My_enumerable module (line 40) via include, and then define the needed $\Leftrightarrow$ method (lines 41–43). From line 46 onward, we can invoke Container.leq. Note also that on line 39 we *reopened* class Container and added a new mixin and method. This is just one way in which programmers can modify classes and methods dynamically. We will examine more of these features in detail in Chapter 4.

Additional information on the Ruby language is available elsewhere [74, 31].

2.2 Parsing Ruby

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

Thus, we opted to write a Ruby parser from scratch. The fundamental challenge in parsing Ruby stems from Ruby's goal of giving users the "freedom to choose" among many different ways of doing the same thing [80]. This philosophy extends to the surface syntax, making Ruby's grammar highly ambiguous from an LL/LR parsing standpoint. In fact, we are aware of no clean specification of Ruby's grammar.[2] Thus, our goal was to keep the grammar specification as understandable (and therefore as extensible) as possible while still correctly parsing all the potentially ambiguous cases. Meeting this goal turned out to be far harder than we originally anticipated, but we were ultimately able to develop a robust parser.

2.2.1 Language Ambiguities

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

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

```
1 def x()  return 4  end
2 def y()
3   if  false  then x = 1 end
```

[2]There is a pseudo-BNF formulation of the Ruby grammar in the on-line Ruby 1.4.6 language manual, but it is ambiguous and ignores the many exceptional cases [48].

```
4     x + 2       # error, x is nil, not a method call
5   end
```

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

As a second parsing challenge, consider the code:

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

Here we invoke the method f, passing a *code block* (higher-order method) as an argument. In this case the code block, delimited by do ... end, takes parameter x and returns $x + 1$. It turns out that code blocks can be used by several different constructs, and thus their use can introduce potential ambiguity. For example, the statement:

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

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

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

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

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

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

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

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

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

2.2.2 A GLR Ruby Parser

To meet these challenges and keep our grammar as clean as possible, we built our parser using the dypgen generalized LR (GLR) parser generator, which supports ambiguous grammars [54]. Our parser uses general BNF-style productions to describe the Ruby grammar, and without further change would produce several parse trees for conflicting cases like those described above. To indicate which tree to prefer, we use helper functions to prune invalid parse trees, and we use *merge* functions to combine multiple parse trees into a single, final output.

An excerpt from our parser is given in Figure 2.2. Line 9 delimits the OCaml functions defined in the preamble (lines 1–8), versus the parser productions (lines 10–

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

Figure 2.2: Example GLR Code

27). A dypgen production consists of a list of terminal or non-terminal symbols followed by a semantic action inside of {}'s. The value of a symbol may be bound using []'s for later reference. For example, the non-terminal stmt_list[ss] reduces to a list of statements that is bound to the identifier ss in the body of the action.

The production primary, defined on line 10, handles expressions that may appear nested within other expressions, like a subexpression block (line 11), a method call (line 13), or a for loop (line 14). On line 16, the action for this rule calls the helper function well_formed_do to prune ill-formed sub-trees. The well_formed_do function is defined in the preamble of the parser file, and is shown on lines 1–4. This function checks whether an expression ends with a method call that includes a code block and, if so, it raises the Dyp.Giveup exception to tell dypgen to abandon this parse tree. This rule has the effect of disambiguating the for...do..end example by only allowing the correct parse tree to be valid. Crucially, this rule does not require modifying the grammar for method calls, keeping that part of the grammar straightforward.

We use a similar technique to disambiguate parentheses for method calls. The command (line 18) and func (line 24) productions define the rules for method calls with and without parentheses respectively. Each includes a list of comma separated arguments using the call_args production, which may reduce to the primary production (line 10). As with the action of the for production, the command production calls the helper function well_formed_command to prune certain parse trees. This function (line 6) aborts the parse if the method call has a single argument that is a grouped sub-expression (stored in the E_Block constructor).

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

Figure 2.3: Disambiguation example

Figure 2.3 shows how our grammar would parse four variations of a Ruby method call. The first column shows the source syntax, and the last two columns show the result of using either func or command to parse the code. As the GLR parsing algorithm explores all possible parse trees, dypgen will attempt to use both productions to parse each method call. The first variation, f(x, y), fails to parse using the command production since the sub-expression production on line 11 must apply, but the pair x, y is not a valid (isolated) Ruby expression. Similarly, the second example does not parse using func since it would reduce f(x) to a method call, and be left with the pair (f(x)), y which is invalid for the same reason. The third example, f x, y would only be accepted by the command production as it contains no parentheses. Thus, our productions will always produce a single parse tree for method calls with at least 2 arguments. However, the last variation f(x) would be accepted by both productions, producing two slightly different parse trees: f(x) vs f((x)). To choose between these, the well_formed_command function rejects a function with a single E_Block argument (line 7), and thus only the func production will succeed.

By cleanly separating out the disambiguation rules in this way, the core productions are relatively easy to understand, and the parser is easier to maintain and

extend. For example, as we discovered more special parsing cases baked into the Ruby interpreter, we needed to modify only the disambiguation rules and could leave the productions alone. Similarly, adding type annotations to individual Ruby expressions required us to only change a single production and for us to add one OCaml function to the preamble. We believe that our GLR specification comes fairly close to serving as a standalone Ruby grammar: the production rules are quite similar to the pseudo-BNF used now [48], while the disambiguation rules describe the exceptional cases. Our parser currently consists of 75 productions and 513 lines of OCaml for disambiguation and helper functions.

Since Ruby has no official specification, we used three techniques to establish our parser is compatible with the existing "reference" parser, which is part of the Ruby 1.8 interpreter. First, we verified that our parser can successfully parse a large corpus of over 160k lines of Ruby code without any syntax errors. Second, we developed 458 hand-written unit tests that ensure Ruby syntax is correctly parsed into known ASTs. For example, we have tests to ensure that both `f x, y` and `f (x), y` are parsed as two-argument method calls. Finally, we parse and then unparse to disk the test suite shipped with the Ruby interpreter. This unparsed code is then executed to ensure the test suite still passes. This last technique is also used to verify that our pretty printing module and the semantic transformations described in Section 2.3 are correct (these modules each have their own unit test suites as well).

2.3 The Ruby Intermediate Language

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

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

We discuss each of these features in turn.

2.3.1 Eliminating Redundant Constructs

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

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

(1) **if** p **then** e **end** (3) e **if** p

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

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

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

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

RIL performs several other additional simplifications. Operators are replaced by the method calls they represent, e.g., x + 2 is translated into x.+(2); while and until are coalesced; logical operators such as and and or are expanded into sequences of conditions, similarly to CIL [50]; and negated forms (e.g., ! =) are translated into a positive form (e.g., ==) combined with a conditional.

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

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

(a) Ruby code

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

(b) RIL Translation

Figure 2.4: Nested Assignment

2.3.2 Linearization

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

To illustrate some of the complexities of evaluation order in Ruby, consider the code in Figure 2.4(a). Here, the result of an exception handling block is stored into the variable result. If an analysis needs to know the value of the right-hand side and only has the AST to work with, it would need to descend into exception block and track the last expression on every branch, including the exception handlers.

Figure 2.4(b) shows the RIL translation of this fragment, which inlines an assignment to a temporary variable on every viable return path. Notice that the

value computed by the ensure clause (this construct is similar to finally in Java) is evaluated for its side effect only, and is not returned. Also notice that the translation has added an explicit nil assignment for the fall-through case for if. (This is an example of implicit behavior, discussed more in Section 2.3.3.) These sorts of details can be very tricky to get right, and it took a significant effort to find and implement these cases. RIL performs similar translations for ensuring that every path through a method body ends with a return statement and that every path through a block ends with a next statement[3].

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

Figure 2.5 demonstrates this difference. The first column lists two similar Ruby assignment statements whose only difference is that the lower one assigns to a tuple (the right-hand side must return a two-element array, which is then split

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

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

Figure 2.5: RIL Linearization Example

and assigned to the two parts of the tuple). The second column lists the method call order—notice that a is evaluated at a different time in the two statements. The third column gives the corresponding RIL code, which makes the evaluation order clear.

Finally, Ruby allows assignments to be chained together. For example, a = b = 3 assigns 3 to both a and b. However, when a method call is used in place of b, the call is evaluated purely for its side effect. For example:

```
1  class A
2    def x=(newx)
3      @x = newx
4      return 3
5    end
6  end
7  a = A.new
8  y = a.x = (2+3)
```

Interestingly, line 8 assigns the value 5 to y, not the return type of a.x =. Thus RIL translates line 8 into:

```
1    t1 = 2.+(3)
2    a.x=(t1)
3    y = t1
```

On line 1, we save the right most expression into a temporary variable so that it is

only evaluated once. Then line 2 calls the x = method with the temporary variable as its argument and discards the return value. Lastly, we store the result of line 1 in y. Again, these intricacies were hard to discover, and eliminating them makes RIL much easier to work with.

2.3.3 Materializing Implicit Constructs

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

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

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

the call on line 7 is the same as **super**(x,y), which is what RIL translates the call to. Without this transformation, every analysis would have to keep track of these

parameters itself, or worse, mistakenly model the call on line 7 as having no actual arguments.

One construct with subtle implicit semantics is rescue. In Figure 2.4(b), we saw this construct used with the syntax **rescue** C => x, which binds the exception to x if it is an instance of C (or a subclass of C). However, Ruby also includes a special abbreviated form **rescue** => x, in which the class name is omitted. The subtlety is that, contrary to what might be expected, a clause of this form does not match arbitrary exceptions, but instead only matches instances of StandardError, which is a superclass of many, but not all, exceptions. To make this behavior explicit, RIL requires every **rescue** clause to have an explicit class name, and inserts StandardError in this case.

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

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

This form may also be used inside of regular expression literals (/ab#{e}c/) or symbols (: "ab#{e}c"). Because this form may introduce side-effects, forms that

would otherwise be considered expressions must be translated to statements. To do this, we use the same transformation as above, and convert the final string into the intended type using Regexp.new or String.to_sym respectively.

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

2.4 Additional Libraries

RIL includes several other modules that can be used to simplify the implementation of a Ruby analysis.

2.4.1 Dataflow

To specify a dataflow analysis [5] in RIL, the user supplies a module that satisfies the following signature:

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

```
9     val meet : t list → t          (* meet operation *)
10  end
```

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

2.4.2 Pretty Printer

RIL includes a pair of pretty printing modules that can be used for emitting its data structures. The first module, CodePrinter outputs RIL's internal data structure as syntactically valid Ruby code, which can be executed by the standard Ruby interpreter. RIL also includes an ErrorPrinter module, which DRuby uses to emit code inside of error messages—since RIL introduces many temporary variables, the code produced by CodePrinter can be hard to understand. Thus, ErrorPrinter omits temporary variables (among other things), showing only the interesting part. For instance, if t1 is a temporary introduced by RIL, then ErrorPrinter shows the call t1 = f() as just f().

Each of these modules provide a set of functions for formatting RIL's datatypes (expressions, statements, etc...) using OCaml's standard Format library. To use these modules, the user provides a format string with %a tokens indicating where the formatted output should be inserted. The format string is then followed by a formatting function paired with the corresponding RIL value. For example:

```
1   open CodePrinter
```

```
2    Format.sprintf ''%a = %a'' format_lhs some_lhs format_expr some_expr
```

formats an assignment statement where format_lhs formats the left hand side some_lhs, and format_expr formats the right hand side some_expr.

RIL also provides a functional unparsing interface in the style of Danvy [23]. Like the modules above, we supply both a CodeUnparser and ErrorUnparser module to print syntactically valid Ruby code, and more user friendly strings respectively. Using this interface, the above example can instead be written:

```
1    open CodeUnparser
2    sprintf (lhs ++ s''='' ++ expr) some_lhs some_expr
```

Here, the format descriptor is built directly from unparsing combinators instead of using %a tokens followed by formatting functions. The expression (lhs ++ s''='' ++ expr), is a descriptor that accepts a left hand side and an expression, which are passed at the end (the s combinator simply inserts a string literal).

RIL also includes a partial reparsing module [25] that lets us mix concrete and abstract syntax. To use it, we call the reparse function instead of sprintf, passing in the necessary format string and arguments:

```
1    reparse ~env:localenv ''%a = %a'' format_lhs some_lhs format_expr some_expr
```

Also, recall from Section 2.2 that parsing in Ruby is highly context-dependent. Thus, on line 2 we pass localenv (which accompanies RIL's main data structure) as the optional argument env to ensure that the parser has the proper state to correctly parse this string in isolation.

2.4.3 Scope Resolution

Class names in Ruby are actually just constants bound to special class objects. In general, constants, which are distinguished by being capitalized, can hold any value, and may be nested inside of classes and modules to create namespaces. For example, the following code:

```
1 class A
2   class B
3     X = 1
4   end
5 end
```

defines the classes A and A::B, and the non-class constant A::B::X. RIL includes a module to resolve the namespaces of all constants statically (except for explicitly dynamic forms, like self :: A). Identifying the binding of a particular constant is actually rather tricky, because it involves a search in the lexical scope in which the constant was created as well as the superclasses and mixins of the enclosing class. Thus, RIL would rewrite the above example as:

```
1 class ::A
2   class ::A::B
3     ::A::B::X = 1
4   end
5 end
```

2.4.4 Visitor

RIL includes an implementation of the visitor pattern modeled after CIL [50]. A visitor object includes a (possibly inherited) method for each RIL syntactic variant (statement, expression, and so on), allowing a client analysis to only override

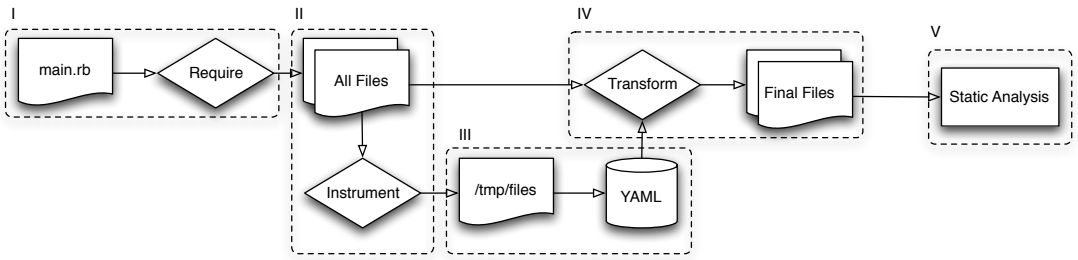

Figure 2.6: Dynamic Instrumentation Architecture

the variants they are interested in. Within each method, a client can then use OCaml's powerful pattern matching features to extract salient attributes of the relevant datatypes. RIL defines a few base classes that other analyses can inherit and extend, such as a visitor that visits every statement in a file, and a visitor that visits every statement in the current scope.

2.4.5 File Loading

Finally, RIL also includes a file loader module to streamline parsing and translation into its intermediate language. This module mimics the path lookup algorithm used by the Ruby interpreter, and can keeps track of which files were already loaded, returning each at most once. The file loader also allows clients to specify different actions based on whether the file is a Ruby source file or a compiled C object file. For instance, DRuby analyzes Ruby files directly, but must load a stub file filled with typing annotations when it encounters a C file.

2.4.6 Runtime Instrumentation

To support writing dynamic analyses, RIL also includes a runtime instrumentation and transformation library. This library allows a client to dynamically instrument and execute a Ruby program, and to use the results of this execution to transform the program before applying a static analysis. The architecture of this library is shown in Figure 2.6 and consists of five main stages. Note that these five stages represent the full capability of this library, and a client may use any subset of this functionality (e.g., to implement a pure dynamic analysis with no static counterpart).

First, stage I executes the target program (potentially using the program's test suite), but with a special Ruby file preloaded that redefines require, the method that loads another file. Our new version of require behaves as usual, except it also records all of the files that are loaded during the program's execution. This is because require has dynamic behavior: Ruby programs may dynamically construct file names to pass to require (and related constructs) or even redefine the semantics of the require method.

After we have discovered the set of application files, in stage II we instrument each file to implement the client's dynamic analysis. We then unparse the modified source files to disk using RIL's pretty printer and execute the resulting program in stage III. Here we must be very careful to preserve the execution environment of the process, e.g., the current working directory, the name of the executed script (stored in the Ruby global $0), and the name of the file (stored in __FILE__ in Ruby). For example, RIL inlines the string value of $0 to ensure the execution is unchanged

by running in a different directory. When the execution is complete, we serialize the data observed by the dynamic analysis to disk using YAML, a simple markup language supported by both Ruby and OCaml [88]. In stage IV, we read in the YAML data and use it to transform the original source code prior to applying a static analysis in stage V.

To use our dynamic analysis library, a client provides the instrumentation used in stage II, the transformation in stage IV, and the static analysis in stage V. We discuss each of these by example in Section 2.5.

2.5 Using RIL

In this section, we demonstrate RIL by developing a simple dynamic null pointer analysis that uses a source-to-source transformation to prevent methods from being called on nil. We then construct a dataflow analysis to improve the performance of the transformed code. Finally, we use our instrumentation library to further optimize functions that are verified with a test suite. Along the way, we illustrate some of the additional features our implementation provides to make it easier to work with RIL. In our implementation, RIL is represented as an OCaml data structure, and hence all our examples below are written in OCaml.

2.5.1 Eliminating Nil Errors

We will develop an example Ruby-to-Ruby transformation written with RIL. Our transformation modifies method calls such that if the receiver object is nil then the

call is ignored rather than attempted. In essence this change makes Ruby programs *oblivious* [59] to method invocations on nil, which normally cause exceptions[4]. As an optimization, we will not transform a call if the receiver is self, since self can never be nil. Our example transformation may or may not be useful, but it works well to demonstrate the use of RIL. The full source code for our example can be found in Appendix A.

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

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

First, we use RIL's File_loader module to parse the given file (specified in the formal parameter fname), binding the result to stmt (lines 2–3). Next, we invoke new safeNil to create an instance of our transformation visitor, and pass that to visit_stmt to perform the transformation (line 4). This step performs the bulk of the work, and is discussed in detail next. Finally, we use the CodePrinter module to output the transformed RIL code as syntactically valid Ruby code, which can be directly executed (line 5).

The code for our safeNil visitor class is as follows:

```
1  class safeNil = object
2    inherit default_visitor as super
3    method visit_stmt node = match node.snode with
```

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

```
4     | MethodCall(_, {mc_target='ID_Self}) → SkipChildren
5     | MethodCall(_, {mc_target=#expr as targ}) →
6       ChangeTo (transform targ node)
7     | _ → super#visit_stmt node
8   end
```

The safeNil class inherits from default_visitor (line 2), which recursively visits every statement, but performs no actions. We then override the inherited visit_stmt method to get the behavior we want: method calls whose target is self are ignored, and we skip visiting the children (line 4). This is sensible because RIL method calls do not have any statements as sub-expressions, thanks to the linearization transformation mentioned in Section 2.3.2. Method calls with non-self receivers are transformed (lines 5–6). Any other statements are handled by the superclass visitor (line 7), which descends into any sub-statements or sub-expressions. For example, at an if statement, the visitor would traverse the true and false branches.

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

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

To build this code, we simply use RIL's reparsing function:

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

Here the string passed on line 3 describes the concrete syntax, just as above, with %a wherever we need "hole" in the string. We pass targ for the first hole, and node for the second. As mentioned in Section 2.4.2 we also include the option env parameter to ensure the string can be parsed in isolation. Note that one potential drawback

of reparsing is that reparse will complain at run-time if mistakenly given unparsable strings; constructing RIL data structures directly in OCaml would cause mistakes to be flagged at compile-time, but such direct construction is far more tedious.

2.5.2 Dataflow Analysis

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

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

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

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

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

We omit the definitions of top, eq, to_string, and meet, as they are uninteresting (they may be found in Appendix A). Instead, we will present only the transfer

function and a small helper function to demonstrate working with RIL's data structures. The function transfer takes as arguments the input dataflow facts map, a statement stmt, and returns the output dataflow facts:

```
1   let rec transfer map stmt = match stmt.snode with
2     | Assign(lhs, #literal) → update_lhs NonNil map lhs
3     | Assign(lhs, `ID_Var(`Var_Local, rvar)) →
4         update_lhs (StrMap.find rvar map) map lhs
5     | MethodCall(Some lhs,_) | Yield(Some lhs,_)
6     | Assign(lhs, _) → update_lhs MaybeNil map lhs
7     | _ → map
8
9   and update_lhs fact map lhs = match lhs with
10    | `ID_Var(`Var_Local, var) → update var fact map
11    | #identifier → map
12    | `Tuple lst → List.fold_left (update_lhs MaybeNil) map lst
13    | `Star (#lhs as l) → update_lhs NonNil map l
14
15  (* val update : string → fact → t → t *)
16 end
```

The first case we handle is assigning a literal (line 2). Since literals are never nil, line 2 uses the helper function update_lhs to mark the left-hand side of the assignment as non-nil.[5]

The function update_lhs has several cases, depending on the left-hand side. If it is a local variable, that variable's dataflow fact is updated in the map (line 10). If the left-hand side is any other identifier (such as a global variable), the update is ignored, since our analysis only applies to local variables. If the left-hand side is a tuple (i.e., a parallel assignment), then we recursively apply the same helper function but conservatively mark the tuple components as MaybeNil. The reason is that parallel assignment can be used even when a tuple on the left-hand side is

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

larger than the value on the right. For example x,y,z = 1,2 will store 1 in x, 2 in y and nil in z. In contrast, the star operator always returns an array (containing the remaining elements, if any), and hence variables marked with that operator will never be nil (line 13). For example, x,y,*z = 1,2 will set x and y to be 1 and 2, respectively, and will set z to be a 0-length array.

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

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

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

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

```
1     ... (* safeNil visitor *)
2   | Method(name,args,body) →
```

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

2.5.3 Dynamic Analysis

In order to reduce the overhead of our transformed code, we can use RIL's transformation library to gather information from a program's test suite. Currently, we use an intraprocedural dataflow analysis, and so method parameters are conservatively assumed to be MaybeNil by our analysis. While public methods may need to handle nil values (possibly by simply throwing an informative exception), private methods might have stronger assumptions, since they can only be called from within the current class. If we knew a private method is only invoked with non-nil values, our analysis could take this information into account. It may be possible to do this with a sufficiently advanced static analysis, but another option is to use a dynamic analysis to ensure this property holds which may be more light-weight. Technically, relying on a test suite to verify this property is unsound, as the test suite may be incomplete. Thus, this gives a programmer a choice, he can improve the performance of the transformation if he trusts his test suite to properly verify the specified methods.

Our strategy is as follows. First, we will instrument the program to ensure that

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

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

Instrumentation To record data from a profiled execution, RIL provides a small runtime library for collecting and storing application values to disk. Data is passed to this library by inserting explicit calls into the application using RIL's instrumentation pass (stage II in Figure 2.6). Thus, the first step in building a new dynamic analysis is to register a new collection class with our runtime library. This is done by providing a small Ruby class that defines how the data is to be collected at runtime:

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

Here, we define the class Dynnil, which inherits from the base class Interceptor that is part of RIL. Our class then defines two methods, a constructor on lines 3–5 that registers our module with the data collection object under the name "dynnil," and a class method extract, which collects the data of interest. In our case, it returns

true if all of the arguments to a method call are non-nil, and false otherwise. The RIL runtime library will instantiate this class automatically at runtime, providing an appropriate instance of collector to the constructor.

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

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

We begin our module, called NilProfile, by declaring the name of our Ruby collection on line 2 (matching the string used in the constructor above). Next, lines 4–12 define the function instrument, which inserts a call to the watch method, a method that Dynnil inherits from the Interceptor class. This method calls our overloaded extract method as a subroutine and stores the boolean result along with the given filename, line number, and method name in the YAML store. Like our example in Section 2.5.1, we construct a call to watch using our reparsing module (lines 6–9), and prepend it to the front of the method body using the seq function (line 11), which creates a sequence of statements. Finally, we construct a new method definition using the helper meth (line 12). The effect of this instrumentation is shown below:

```
1  def add(x,y) # before
2    x + y
```

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

Lastly, we invoke the instrument function by constructing a new visitor object that instruments a method on line 6 if should_instrument returns true and otherwise keeps the existing method definition (line 7).

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

Transformation After our application has been profiled, we can transform it using our safeNil visitor as defined in Section 2.5.2. The only modification we must make is that RIL's dynamic analysis library provides us with an additional lookup function, which can be used to query the YAML store. Thus, we will update the Method definition case in our visitor, by first checking the YAML store to see if the arguments are all non-nil, and if so, use stronger dataflow facts. Our NilProfile OCaml module continues:

```
13   module Domain = Yaml.YString
14   module CoDomain = Yaml.YBool
15
16   (* lookup : Domain.t → pos → CoDomain.t list *)
17   let really_nonnil lookup mname pos =
18     let uses = lookup mname pos in
19       if uses = [] then false
```

```
20          else not (List.mem false uses)
21
22      ... (* safeNil visitor *)
23    | Method(mname,args,body) →
24      let init_facts =
25        if really_nonnil lookup mname body.pos
26        then init_formals args NonNil
27        else init_formals args MaybeNil
28      in
29      let in_facts, _ = NilDataFlow.fixpoint body init_facts in
30        (* visit body *)
31  end
```

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

2.6 Related Work

Another project that provides access to the Ruby AST is ruby_parser [60]. This parser is written in Ruby and stores the AST as an S-expression. ruby_parser performs some syntactic simplifications, such as translating unless statements into

if statements, but does no semantic transformations such as linearizing effects or reifying implicit constructs. The authors of ruby_parser have also developed several tools to perform syntax analysis of Ruby programs [61]: *flay*, which detects structural similarities in source code; *heckle*, a mutation-based testing tool; and *flog*, which measures code complexity. We believe these tools could also be written using RIL, although most of RIL's features are tailored toward developing analyses that reason about the semantics of Ruby, not just its syntax.

Several integrated development environments [58, 51] have been developed for Ruby. These IDEs do some source code analysis to provide features such as code refactoring and method name completion. However, they are not specifically designed to allow users to develop their own source code analyses. Integrating analyses developed with RIL into an IDE would be an interesting direction for future work.

The Ruby developers recently released version 1.9 of the Ruby language, which includes a new bytecode-based virtual machine. The bytecode language retains some of Ruby's source level redundancy, including opcodes for both if and unless statements [89]. At the same time, opcodes in this language are lower level than RIL's statements, which may make it difficult to relate instructions back to their original source constructs. Since this bytecode formulation is quite new, it is not yet clear whether it would make it easier to write analyses and transformations, as was the goal of RIL.

While the Ruby language is defined by its C implementation, several other implementations exist, such as JRuby [41], IronRuby [39], and MacRuby [42]. These

projects aim to execute Ruby programs using different runtime environments, taking advantage of technologies present on a specific platform. For example, JRuby allows Ruby programs to execute on the Java Virtual Machine, and allows Ruby to call Java code and vice versa. While these projects necessarily include some analysis of the programs, they are not designed for use as an analysis writing platform.

Finally, RIL's design was influenced by the C Intermediate language [50], a project with similar goals for C. In particular, the author's prior experience using CIL's visitor class, and CIL's clean separation of side-effect expressions from statements, lead to a similar design in RIL.

Chapter 3

A Static Type System for Ruby

In this chapter, we present the core static type system used by DRuby. We begin with a design overview in Section 3.1, followed by an example-driven introduction to our type system in Section 3.2. In Section 3.3, we formalize and prove a type soundness result for a small Ruby-like calculus that highlights the interesting parts of our type system. We then describe our implementation in Section 3.4, followed by the results of applying our type system to a small benchmark suite in Section 3.5. Lastly, we describe some related work in Section 3.6.

3.1 Overview

Designing DRuby's type system has been a careful balancing act. On the one hand, we would like to statically discover all programs that produce a type error at run time. On the other hand, we should not falsely reject too many correct programs, lest programmers find static typing too restrictive. Thus to maximize flexibility, DRuby gives the programmer control over the amount of static checking it performs. By default, DRuby rejects any program that is potentially dynamically incorrect. However, a programmer may override this behavior by using annotations to force

DRuby to accept a potentially incorrect program.

To minimize the burden on the programmer, DRuby tries to be precise as possible. For example, we track the types of local variables flow-sensitively through the program, e.g., allowing a variable to first contain a String and then later an Array. On the other hand, some of Ruby's more dynamic features, such as metaprogramming with eval, are difficult to model statically and require more sophisticated analyses to be accepted as type correct. It turns out that modeling constructs like eval is critical to typing standard library code, and we will revisit this more in Chapter 4, where we present a novel technique for handling such constructs.

Between these two extremes lies some middle ground: DRuby might not be able to infer a static type for a method, but it may nonetheless have one. For example, DRuby supports but does not infer intersection types. To model these kinds of types, DRuby provides an annotation language that allows programmers to annotate code with types that are assumed correct at compile time and then are checked dynamically at run time. While the Ruby interpreter already safely aborts an execution after a run-time type error, our checked annotations localize errors and properly *blame* [28] the errant code. Annotations are required to give types to Ruby's core standard library, since it is written in C rather than Ruby. We believe our annotation language is easy to understand, and indeed it resembles the "types" written down informally in the standard library documentation.

DRuby's type language is designed with the features we found necessary to precisely type Ruby code: union and intersection types [56], object types (to complement nominal types) [1], F-bounded polymorphism [57], a self type, tuple types

for heterogeneous arrays, and optional and variable arguments in method types.

DRuby also includes a type inference algorithm to statically discover type errors in Ruby programs, which can help programmers detect problems much earlier than dynamic typing. By providing type inference, DRuby helps maintain the lightweight feel of Ruby, since programmers need not write down extensive type annotations to gain the benefits of static typing.

To evaluate our core type system, we have applied DRuby to a suite of small benchmark programs ranging from 29–1030 lines of code. DRuby found these 18 benchmarks to be largely amenable to static typing: it reported 5 potential errors and 16 warnings for questionable code compared to 16 false positives. Chapter 4 will present additional results for larger programs once we have developed the necessary techniques to hand Ruby's more dynamic features.

3.2 Static Types for Ruby

We begin our discussion of DRuby's type system with a series of examples. The design of DRuby was driven by experience—we included features expressive enough to type idioms common in our benchmark programs and the standard library APIs, but at the same time we tried to keep types easy for programmers to understand.

3.2.1 Basic Types and Type Annotations

In Ruby, everything is an object, and all objects are class instances. For example, 42 is an instance of Fixnum, **true** is an instance of TrueClass, and an instance of a

class defined with **class** A ... **end** is created with A.new. Thus, a basic DRuby type is simply a class name, e.g., Fixnum, TrueClass, A, etc. Classes can extend other classes, but a subclass need not be a subtype. The class Object is the root of the class hierarchy.

Although Ruby gives the illusion that built-in values such as 42 and **true** are objects, in fact they are implemented inside the Ruby interpreter in C code, and thus DRuby cannot infer the types of the methods defined for of these classes. However, we can declare their types using DRuby's type annotation language. In DRuby, type annotations are written before the corresponding class or method declaration and are prefixed with *##%*, and therefore appear as comments to the standard Ruby interpreter.

We developed a file `base_types.rb` that annotates the "core library," which contains classes and globals that are pre-loaded by the Ruby interpreter and are implemented purely in C. This file includes types for 1,112 methods in 190 classes and 19 modules, using 1,262 lines of type annotations in total. Our implementation analyzes `base_types.rb` before applying type inference to a program.

For example, here is part of the declaration of class String, with *##%* removed for clarity:

```
1  class String
2    "+" : (String) → String
3    insert : (Fixnum, String) → String
4    ...
5  end
```

The first declaration types the method + (non-alphanumeric method names appear in quotes), which concatenates a String argument with the receiver and returns a

new String. Similarly, the next line declares that insert takes a Fixnum (the index to insert at) and another String, and produces a new String as a result.

3.2.2 Intersection Types

Many methods in the standard library have different behaviors depending on the number and types of their arguments. For example, here is the type of String's include? method, which either takes a Fixnum representing a character and returns true if the object contains that character, or takes a String and performs a substring test:

```
1   include? : (Fixnum) → Boolean
2   include? : (String) → Boolean
```

The type of include? is an example of an *intersection type* [56]. A general intersection type has the form t **and** t', and a value of such a type has *both* type t and type t'. For example, if A and B are classes, an object of type A **and** B must be a common subtype of both A and B. In our annotation syntax for methods, the **and** keyword is omitted, only method types may appear in an intersection, and each conjunct of the intersection is listed on its own line.

Another example of intersection types is String's slice method, which returns either a character or a substring:

```
1   slice : (Fixnum) → Fixnum
2   slice : (Range) → String
3   slice : (Regexp) → String
4   slice : (String) → String
5   slice : (Fixnum, Fixnum) → String
6   slice : (Regexp, Fixnum) → String
```

Notice that this type has quite a few cases, and in fact, the Ruby standard library documentation for this function has essentially the same type list.[1] Intersection types serve a purpose similar to method overloading in Java, although they are resolved at run time via type introspection rather than at compile time via type checking. Annotation support for intersection types is critical for accurately modeling key parts of the core library—77 methods in `base_types.rb` use intersection types. Note that our inference system is unable to infer intersection types, as method bodies may perform ad-hoc type tests to differentiate various cases, and so these types can currently be created only via annotations.

3.2.3 Optional Arguments and Varargs

One particularly common use of intersection types is methods with optional arguments. For example, String's chomp method has the following type:

```
1  chomp : () → String
2  chomp : (String) → String
```

Calling chomp with an argument s removes s from the end of self, and calling chomp with no arguments removes the value of (global variable) $/ from self. Since optional arguments are so common, DRuby allows them to be concisely specified by prefixing an argument type with ?. The following type for chomp is equivalent to the intersection type above:

```
1  chomp : (?String) → String
```

[1] `http://ruby-doc.org/core/classes/String.html#M000858`

DRuby also supports varargs parameters, specified as $*t$, meaning zero or more parameters of type t (the corresponding formal argument would contain an Array of t's). For example, here is the type of delete, which removes any characters in the intersection of its (one or more) String arguments from self:

```
1   delete : (String, *String) → String
```

Notice this type is equivalent to an intersection of an unbounded number of types.

3.2.4 Union Types

Dually to intersection types, DRuby supports *union types*, which allow programmers to mix different classes that share common methods. For example, consider the following code:

```
1  class A; def f() end end
2  class B; def f() end end
3  x = (if ... then A.new else B.new)
4  x.f
```

Even though we cannot statically decide if x is an A or a B, this program is clearly well-typed at run time, since both classes have an f method. Notice that if we wanted to write a program like this in Java, we would need to create some interface I with method f, and have both A and B implement I. In contrast, DRuby supports union types of the form t **or** t', where t and t' are types (which can themselves be unions) [38]. For example, x above would have type A **or** B, and we can invoke any method on x that is common to A and B. We should emphasize the difference with intersection types here: A value of type A **and** B is both an A and a B, and so it has *both* A's and B's methods, rather than the union type, which has one set of methods

or the other set, and we do not know which.

Note that the type Boolean used in the type of include? above is equivalent to TrueClass **or** FalseClass (the classes of true and false, respectively). In practice we just treat Boolean as a pseudo-class, since distinguishing true from false statically is essentially useless—most uses of Booleans could yield either truth value.

3.2.5 Object Types

Thus far we have only discussed types constructed from class names and self. However, we need richer types for inference so that we can describe objects whose classes we do not yet know. For example, consider the following code snippet:

```
1  def f(x)  y = x.foo; z = x.bar;  end
```

If we wanted to stick to nominal typing only, we could try to find all classes that have methods foo and bar, and then give x a type that is the union of all such classes. However, this would be both extremely messy and non-modular, since changing the set of classes might require updating the type of f.

Instead, DRuby includes *object types* $[m_0 : t_0, \ldots, m_n : t_n]$, which describes an object in which each method m_i has type t_i. The parameter x above has type [foo : () → t, bar : () → u] for some t and u. As another example, `base_types.rb` gives the print method of Kernel the type

```
1    print : (*[to_s : () → String]) → NilClass
```

Thus print takes zero or more objects as arguments, each of which has a no-argument to_s method that produces a String. Object types are critical to describe code that

relies on the structure of a type (this structural view of types is commonly referred to as duck-typing[2]), rather than the type name. Forty-nine methods in `base_types.rb` include object types.

3.2.6 F-Bounded Polymorphism

To give precise types to container classes, we use *F-bounded polymorphism*, also called *generics* in Java. For example, here is part of the Array class type, which is parameterized by a *type variable* t, the type of the array contents:

```
1  class Array<t>
2    at : (Fixnum) → t
3    collect <u> : () {t → u} → Array<u>
4    ...
5  end
```

As usual, type variables bound at the top of a class can be used anywhere inside that class. For example, the at method takes an index and returns the element at that index. Methods may also be polymorphic. For example, for any type u, the collect (map) method takes a code block (higher-order function) from t to u and produces an array of u.

We also allow bounds to be placed on the quantified variables. For example,

```
1  class Hash<k,v>
2    fetch : k → v
3    store<v'> ; v' ≤ v : (k, v') → v'
4    ...
5  end
```

[2] *When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.*—James Whitcomb Riley [84]

here, the Hash class is parameterized by two type parameters, k for the type of keys and v for values. A value is extracted from the hash using fetch which takes a key and returns the corresponding value. The store method takes any type v′ that is a subtype of v and returns that same type. Thus, if hsh has type Hash < Fixnum, A > and B is a subtype of A, then we can safely call

```
1  hsh.store(1, B.new).b_method
```

which passes in B, and then calls the method b_method (defined only in B) on the return type. If we had instead used the type

```
1  store : (k, v) → v
```

then the above usage would not be well-typed, since DRuby would think the return type was A and the call to b_method would not be allowed. When a bound is omitted, it is assumed to be $\top$.

3.2.7 The self type

Consider the following code snippet:

```
1  class A; def me() self end end
2  class B < A; end
3  B.new.me
```

Here class A defines a method me that returns **self**. We could naively give me the type () → A, but observe that B inherits me, and the method invocation on the last line returns an instance of B, not of A. Thus our type language includes the special type, "self" which always refers to the current receiver that invoked the method. Therefore a more appropriate type for the method me would be () → self.

Internally, DRuby de-sugars the self type into a polymorpic type bounded by the current class. For example, DRuby would internally translate the type for me into me<t> ; t ≤ A : () → t.

3.2.8 Abstract Classes and Mixins

Self annotations may also be explicitly quantified in order to support abstract methods. For example:

```
1   class Base
2     def foo
3       bar()
4     end
5     def baz
6       return 3
7     end
8   end
```

Here, the Base class is abstract as it only defines the foo and baz methods, but not the bar method. Classes that inherit from Base must define the bar method in order allow calls to the foo method. To allow this flexibility, the foo method can be given an annotation that explicitly quantifies self under a different bound than Base, as dictated by its use:

```
1  foo<t> ; self ≤ [bar : () → t] : () → t
```

This annotation ensures that any call to foo must have as the receiver an object that responds to the bar method.

However, Ruby has no restrictions on abstract classes, and so Base may be freely instantiated with new and any calls to baz will be type-safe (but not to foo). Ruby also includes support for *modules* (a.k.a *mixins* [14]) for implementing multiple

inheritance which are similar to abstract classes. For example, the following code defines a module Ordered, which includes an leq method that calls another method $\Leftrightarrow$ (three-way comparison). Class Container mixes in Ordered via include and defines the needed $\Leftrightarrow$ method.

```
1  module Ordered          # module/mixin creation
2    def leq(x)
3      (self ⇔ x) ≤ 0      # note "⇔" does not exist here
4  end end
5
6  class Container
7    include Ordered       # mix in module
8    def ⇔(other)          # define required method
9      @x ⇔ other.get
10 end end
```

Standard object-oriented type systems would check all the methods in a class together, but DRuby cannot do that because of abstract classes and mixins, e.g., when typing leq, the actual $\Leftrightarrow$ method referred to depends on where Ordered is included. Instead, whenever a method m invokes another method on self, DRuby adds the appropriate constraints to self, e.g., when typing leq, DRuby adds a constraint that self has a $\Leftrightarrow$ method. Then when m is called, we check those constraints against the actual receiver.

3.2.9 Tuple Types

The type Array<t>, that we saw in Section 3.2.6, describes *homogeneous* arrays in which each element has the same type. However, since Ruby is dynamically typed, it also allows programmers to create heterogeneous arrays, in which each element may have a different type. This is especially common for returning multiple values

from a function, and there is even special parallel assignment syntax for it. For example, the following code

```
1 def f()
2   return [1, true]
3 end
4 a, b = f
```

assigns 1 to a and true to b. If we were to type f's return value as a homogeneous Array, the best we could do is Array<Fixnum **or** Boolean>, with a corresponding loss of precision.

DRuby includes a special type Tuple<t_1,..., t_n> that represents an array whose element types are, left to right, t_1 through t_n. When we access an element of a Tuple using parallel assignment, we then know precisely the element type.

Of course, we may initially decide something is a Tuple, but then subsequently perform an operation that loses the individual element types, such as mutating a random element or appending an Array. In these cases, we apply a special subsumption rule that replaces the type Tuple<t_1, ..., t_n> with the type Array<t_1 **or** ... **or** t_n>.

3.2.10 First Class Methods

DRuby includes support for another special kind of array: method parameter lists. Ruby's syntax permits at most one code block (higher-order function) to be passed to a method. For example, we can map λx.x+1 over an array as follows:

```
1   [1, 2, 3].collect {|x| x + 1}   # returns [2, 3, 4]
```

If we want to pass multiple code blocks into a method or do other higher-order programming (e.g., store a code block in a data structure), we need to convert it to

a Proc object:

```
1   f = Proc.new {|x| x + 1}    # f is λx.x+1
2   f.call(3)                   # returns 4
```

A Proc object can be constructed from any code block and may be called with any number of arguments. To support this special behavior, in `base_types.rb`, we declare Proc as follows:

```
1  class Proc<^args,ret>
2     initialize : () {(^args) → ret} → Proc<^args,ret>
3     call : (^args) → ret
4  end
```

The Proc class is parameterized by a parameter list type ^args and a return type ret. The ^ character indicates the corresponding type variable ranges over parameter list types (e.g., optional and vararg arguments) instead of class or object types. The initialize method (the constructor called when Proc.new is invoked) takes a block with parameter types ^args and a return type ret and returns a corresponding Proc. The call method then has the same parameter and return types.

As another example use of ^, consider the Hash class:

```
1  class Hash<k, v>
2     initialize : () {(Hash<k, v>, k) → v} → Hash<k, v>
3     default_proc : () → Proc<^(Hash<k, v>, k),v>
4  end
```

The Hash class is parameterized by k and v, the types of the hash keys and values, respectively. When creating a hash table, the programmer may supply a default function that is called when an accessing a non-existent key. Thus the type of initialize includes a block that is passed the hash table (Hash<k,v>) and the missing key (k), and produces a value of type v. The programmer can later extract this

method using the default_proc method, which returns a Proc object with the same type as the block.

3.2.11 Types for Variables and Nil

DRuby tracks the types of local variables flow-sensitively, maintaining a per-program point mapping from locals to types, and combining types at join points with unions. This allows us to warn about accesses to locals prior to initialization, and to allow the types of local variables to vary from one statement to another. For example, in the following code

```
1  b = 42      # b is a Fixnum (no length method)
2  b = "foo"   # b is now a String (has a length method)
3  b.length    # only look for length in String, not Fixnum
```

we need flow-sensitivity for locals so to permit the call b.length.

We have to be careful about tracking local variables that may be captured by blocks. For example, in the code

```
1  x = 1
2  foo() { |y| x = y }  # pass in function λy.x=y
```

the value of x in the outer scope will be changed if foo invokes the code block. To keep our analysis simple, we track potentially captured variables flow-insensitively, meaning they have the same type throughout the program. We also model class, instance (fields), and global variables flow-insensitively.

Finally, as it is common in statically typed object-oriented languages, we treat nil as if it is an instance of any class. Not doing so would likely produce an excessive number of false alarms, or require a very sophisticated analysis.

3.2.12 Unsupported features

As we stated in the introduction, DRuby aims to be flexible enough to type common Ruby programming idioms without introducing needless complexity into its type system. Thus, there are a number of uses of Ruby that DRuby cannot type.

First, there are standard library methods with types DRuby cannot represent. For example, there is no finite intersection type that can describe Array.flatten, which converts an n-dimensional array to a one-dimensional array for any n.

Second, some features of Ruby are difficult for any static type system. In particular, Ruby allows classes and methods to be changed arbitrarily at run time (e.g., added to via class reopening or removed via the undef_method method). To keep DRuby practical, we assume that all classes and methods defined somewhere in the code base are available at all times and disallow any calls to undef_method.

Third, in Ruby, each object has a special *eigenclass* that can be modified without affecting other objects. For example, suppose x and y are both instances of Object. Then **def** x.foo() ... **end** adds method foo to the eigenclass of x but leaves y unchanged. Thus, after this declaration, we can invoke x.foo but not y.foo. DRuby is unable to model eigenclasses because it cannot always decide statically which object's type is being modified.

Finally, Ruby includes reflection (e.g., accessing fields by calling instance_variable_get) and dynamic evaluation (the eval method). Combined with the ability to change classes at run time, this gives programmers powerful metaprogramming tools. For example, our *text-highlight* benchmark includes the following:

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

This code iterates through ATTRIBUTES, an array of strings. For each element it creates a string code containing a new method definition, and then evaluates code. The result is certainly elegant—methods are generated dynamically based on the contents of an array. Handling this kind of code is difficult, and will be the focus of Chapter 4. For purposes of this chapter and evaluating our core static type system, we manually expanded calls to "eval" to use statically typed equivalents in our benchmarks (Section 3.5).

3.2.13 Cast Insertion

Dynamic annotations allow DRuby to interface with code it cannot statically verify, either because the code is written in C, or because it is too expressive for our type system. However, because DRuby trusts such annotations to be correct, improperly annotated code may cause run-time type errors, and these errors may be misleading. For example, consider the following code:

```
1   ##% evil : Fixnum → Fixnum
2   ##% evil : Float → Float
3   def evil(x) eval("#{x}.to_s()") end
4   def f() return(evil(2) * 3) end
5   f()-4
```

Here we have annotated the evil method on line 3 to return either a Fixnum or a Float based on the type of its parameter, and given this assumption, DRuby verifies that the remainder of the program is statically type safe. However, evil uses eval

to return the string "2." This does not cause a type error on line 4 because String has a method $*$: Fixnum→String, but then the resulting String is returned to line 5, where the Ruby interpreter finally raises a NoMethodError, since String does not have a $-$ method.

There are two problems here. First, DRuby has certified that line 5 will never cause a type error, and yet that is exactly what has occurred. Second, the error message gives little help in determining the source of the problem, since it accuses the return value of f of being incorrect, rather than the return value of evil.

We can solve this problem with higher-order contracts [28], which allow us to attribute the *blame* for this example correctly. To ensure that a method matches its annotation, we instrument each annotated method with run-time checks that inspect the parameters and return values of the method. DRuby ensures that "inputs" have the correct type, and the dynamic checks ensure the "outputs" match the static types provided to DRuby.

There is one catch, however: if a method has an intersection type, we may not know statically which parameter types were passed in, which might affect what result type to check for. Whether the result of evil should be a Fixnum or a Float depends on the input parameter type. Our instrumented code for this example (slightly simplified) looks like the following:

```
1  alias untyped_evil evil
2  def evil(arg)
3    sigs = [[Fixnum,Fixnum],[Float, Float]]
4    result = untyped_evil(arg)
5    fail () unless sigs.include? [arg.class, result.class]
6  end
```

On line 1, we save the old `evil` method under the name `untyped_evil` before defining a new version of `evil` on line 2. The annotated signature is then stored as a Ruby value in the variable `sigs` (line 3), and the original method is called on line 4. Finally, on line 5 we check to ensure that the combination of the argument class and return class are included in the list of possible signatures.

DRuby supports run-time checking for nominal and object types, union and intersection types, parameter lists with regular, optional, and vararg arguments, blocks, and polymorphic methods. We plan on expanding our support for annotations in future work, including runtime checks for polymorphic classes and annotations on individual expressions.

3.3 MiniRuby

We now present MiniRuby, a small Ruby-like calculus. The interesting parts of this language are implicitly defined local variables, imperatively updatable fields, tuples, parallel assignment, first-class Class names, and type inspection. At the type level, we have F-bounded polymorphism, union and intersection types, width and depth subtyping on objects, and structural subtyping. However, MiniRuby does deviate from Ruby in several ways. First, we require explicit definitions for fields, whereas fields in Ruby are created at their first assignment. Second, MiniRuby stratifies method definitions from expressions. In Ruby, everything is an expression, and methods may be defined at any point. Will we relax this restriction for DynRuby, our formalism in Chapter 4. Finally, we do not include inheritance. While this is

an important aspect of Ruby, it provides no technical insight, only complicating the lookup of fields and methods and thus we omit it from our calculus.

3.3.1 Source Language

The term language for MiniRuby is shown in Figure 3.1(a). Expressions e include identifiers, which are local variables x, the distinguished variable self, fields $@x$, and class names A (which are first class). We also include tuples inside of []s, and sequencing with semicolon. Assignment comes in three forms: to local variables, to fields, and a parallel assignment form. Objects are created from classes using $\mathsf{new}\ e$, which allocates a new object of type e. Notably, e is an arbitrary expression, not just a class name. Methods are invoked by using $e_0.m(e_1, \ldots, e_n)$, which sends the message m to the receiver object e_0 with arguments $e_1, \ldots, e_n$. Finally, we include a type discrimination form, $\mathsf{typecase}\ e_1\ \mathsf{when}\ (x : A)\ e_2\ \mathsf{else}\ e_3$. Here, if the runtime type of e_1 is A, then e_2 is evaluated with x bound to e_1. Otherwise, e_3 is evaluated.

A program P is made up of a list of class definitions c followed by a "main" expression e. A class definition is annotated with a polytype and includes a series of definitions d. A definition is either a method definition $\mathsf{def}\ m(x_1, \ldots, x_n) : \eta = e$, which is annotated with a intersection type η (discussed below), or a field definition $@x = e$.

$$
\begin{array}{rcll}
e & ::= & id & \text{Identifiers} \\
 & | & [e_1, \ldots, e_n] & \text{Tuple} \\
 & | & e; e & \text{Sequencing} \\
 & | & lval = e & \text{Assignment} \\
 & | & \mathsf{new}\ e & \text{Object creation} \\
 & | & e.m(e, \ldots, e) & \text{Method invocation} \\
 & | & \mathsf{typecase}\ e\ \mathsf{when}\ (x : A)e\ \mathsf{else}\ e & \text{Type case} \\
 & & & \\
d & ::= & \mathsf{def}\ m(x_1, \ldots, x_n) : \eta = e & \text{Method definition} \\
 & | & @x = e & \text{Field definition} \\
c & ::= & \mathsf{class}\ A : \sigma = d^* & \text{Class definition} \\
P & ::= & c^*; e &
\end{array}
$$

$$
\begin{array}{rcl}
lval & ::= & x \mid @x \mid x_1, \ldots, x_n \\
id & ::= & \mathsf{self} \mid x \mid @x \mid A
\end{array}
$$

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

(a) Term Language

$$
\begin{array}{rcll}
\tau & ::= & \alpha \mid (\tau \times \cdots \times \tau) \to \tau \mid \tau \cup \tau & \text{Mono-Types} \\
 & | & (\tau \times \cdots \times \tau) \mid [F; M] \mid \tau\ \mathsf{class}\ \mid \top & \\
\sigma & ::= & \forall\ \alpha <: \tau\ .\sigma \mid \tau & \text{Poly-Types} \\
\eta & ::= & \sigma \mid \eta \cap \eta & \text{Intersection Types} \\
F & ::= & \emptyset \mid @x : \tau, F & \text{Field Sets} \\
M & ::= & \emptyset \mid m : \eta, M & \text{Method Sets} \\
 & & & \\
\Omega & : & \emptyset \mid \Omega, A : \sigma & \text{Heap Environment} \\
\Gamma & ::= & \emptyset \mid \Gamma, \alpha <: \tau & \text{Type Var Environment} \\
\Delta & ::= & \emptyset \mid \Delta, x : \tau & \text{Term Var Environment}
\end{array}
$$

(b) Type Language

Figure 3.1: MiniRuby

3.3.2 Type Checking

Our type language is shown in Figure 3.1(b). Types are stratified into monotypes τ, polytypes σ, and intersection types η. Monotypes include type variables α, uncurried function types $(\tau \times \cdots \times \tau) \rightarrow \tau$, union types $\tau \cup \tau$, and tuple types $\tau \times \cdots \times \tau$. Objects types $[F; M]$ include a field set F and a method set M. These sets are viewed as unordered collections, and we implicitly allow permutations among the elements in the set. Class types are written with postfix class, and the top type $\top$ is a supertype of all types.

Classes and methods may be given a bounded polymorphic type $\forall\alpha <: \tau.\sigma$. Here, α may be instantiated to any monontype τ' that is a subtype of τ (that is, our polymorphism is predicative). Intersection types (η) are restricted to methods, which are immutable in this calculus, thus keeping soundness [24]. Also, since we only allow intersection types on methods (not expressions in general), and methods are not curried, we have no need for the problematic distributive rule for intersection types [24].

We describe our type checking rules formally in Figure 3.2 using judgments in natural deduction style. The notation

$$\frac{\Im_1 \qquad \cdots \qquad \Im_n}{\Im}$$

means that the conclusion $\Im$ holds if all of the premises $\Im_i$ also hold. For a type system, our aim is to ensure that every expression in a program can be assigned a type and thus our assertions will be of the form $e : \tau$, meaning that expression e

$\boxed{\Omega; \Gamma; \Delta \vdash e : \tau; \Delta'}$

$$\frac{\Omega; \Gamma; \Delta \vdash e : \tau'; \Delta' \qquad \Gamma \vdash \tau' <: \tau}{\Omega; \Gamma; \Delta \vdash e : \tau; \Delta'} \ (\textsc{Subsumption}_\tau)$$

$$\frac{\Omega; \Gamma; \Delta_1 \vdash e_1 : \tau_1; \Delta_2 \quad \Omega; \Gamma; \Delta_2 \vdash e_2 : \tau_2; \Delta_3}{\Omega; \Gamma; \Delta_1 \vdash e_1; e_2 : \tau_2; \Delta_3} \ (\textsc{Seq}_\tau)$$

$$\frac{x \in dom(\Delta)}{\Omega; \Gamma; \Delta \vdash x : \Delta(x); \Delta} \ (\textsc{Var}_\tau)$$

$$\frac{\Omega; \Gamma; \Delta \vdash e : \tau; \Delta_1}{\Omega; \Gamma; \Delta \vdash x = e : \tau; \Delta_1[x \mapsto \tau]} \ (\textsc{Var Assign}_\tau)$$

$$\frac{\Omega; \Gamma; \Delta_i \vdash e_i : \tau_i; \Delta_{i+1} \qquad i \in 1..n}{\Omega; \Gamma; \Delta_1 \vdash [e_1, \ldots, e_n] : (\tau_1 \times \cdots \times \tau_n); \Delta_{n+1}} \ (\textsc{Tuple}_\tau)$$

$$\frac{\Omega; \Gamma; \Delta \vdash e : (\tau_1 \times \cdots \times \tau_n)}{\Omega; \Gamma; \Delta \vdash x_1, \ldots, x_n = e : (\tau_1 \times \cdots \times \tau_n); \Delta[x_i \mapsto \tau_i]} \ (\textsc{Tuple Assign}_\tau)$$

$$\frac{\Omega; \Gamma; \Delta \vdash \mathsf{self} : \tau_s; \Delta \quad \Gamma \vdash \tau_s <: [@x : \tau']}{\Omega; \Gamma; \Delta \vdash @x : \tau'; \Delta} \ (\textsc{Field}_\tau)$$

$$\frac{\Omega; \Gamma; \Delta \vdash e : \tau; \Delta' \quad \Omega; \Gamma; \Delta' \vdash \mathsf{self} : \tau_s; \Delta' \quad \Gamma \vdash \tau_s <: [@x : \tau]}{\Omega; \Gamma; \Delta \vdash @x = e : \tau; \Delta'} \ (\textsc{Field Assign}_\tau)$$

$$\frac{\mathsf{self} \in dom(\Delta) \quad \Delta(\mathsf{self}) = [F; M]}{\Omega; \Gamma; \Delta \vdash \mathsf{self} : [F; M]; \Delta} \ (\textsc{Self}_\tau)$$

$$\frac{\Omega(A) = \sigma \qquad \Gamma \vdash \sigma <: [F; M]\ \mathsf{class}}{\Omega; \Gamma; \Delta \vdash A : [F; M]\ \mathsf{class}\ ; \Delta} \ (\textsc{Class}_\tau)$$

$$\frac{\Omega; \Gamma; \Delta \vdash e : \tau\ \mathsf{class}\ ; \Delta' \quad \tau = [F; M]}{\Omega; \Gamma; \Delta \vdash \mathsf{new}\ e : \tau; \Delta'} \ (\textsc{New}_\tau)$$

$$\frac{\Omega; \Gamma; \Delta_i \vdash e_i : \tau_i; \Delta_{i+1} \qquad i \in 1..n \quad \Omega; \Gamma; \Delta_{n+1} \vdash e_0 : \tau_0; \Delta_{n+2} \quad \Gamma \vdash \tau_0 <: [m : (\tau_1 \times \cdots \times \tau_n) \to \tau]}{\Omega; \Gamma; \Delta_0 \vdash e_0.m(e_1, \ldots, e_n) : \tau; \Delta_{n+2}} \ (\textsc{Call}_\tau)$$

$$\frac{\Omega; \Gamma; \Delta \vdash e_1 : [F; M]; \Delta_1 \qquad \Omega; \Gamma; \Delta_1 \vdash A : \tau'\ \mathsf{class}\ ; \Delta_1 \quad \Omega; \Gamma; \Delta_1, x : \tau' \vdash e_2 : \tau; \Delta_2 \qquad \Omega; \Gamma; \Delta_1 \vdash e_3 : \tau; \Delta_3 \quad \Delta' = \Delta_2|_{dom(\Delta_1)} \uplus \Delta_3|_{dom(\Delta_1)}}{\Omega; \Gamma; \Delta \vdash \mathsf{typecase}\ e_1\ \mathsf{when}\ (x : A)\ e_2\ \mathsf{else}\ e_3 : \tau; \Delta'} \ (\textsc{TypeCase}_\tau)$$

Figure 3.2: Type Checking Rules for Expressions

has type τ. Our judgments make use of three environments: Ω maps class names to class types ($A : \sigma$); the environment Γ maps type variables to subtype bounds ($\alpha <: \tau$); and Δ maps term variables to types ($x : \tau$).

Our judgments are flow-sensitive, meaning the judgments have both an "in" environment and an "out" environment to allow the types variables to change throughout the body of the program. Specifically, our judgments have the form:

$$\Omega; \Gamma; \Delta \vdash e : \tau; \Delta'$$

meaning that in environments Ω, Γ, and Δ, expression e has type τ, and the evaluation of e produces environment Δ'.

We now discuss each type rule in turn. (Subsumption$_\tau$), (Seq$_\tau$), (Var$_\tau$), and (Tuple$_\tau$) are standard. For (Var Assign$_\tau$), we update the environment with x bound to the type of the right-hand side. For (Tuple Assign$_\tau$), we require the right-hand side to be a tuple of the correct width and then update each variable on the left-hand side in the resulting environment.

The rule (Field$_\tau$) types a field access. Fields may only be accessed through self, which is given some type τ_s. (Field$_\tau$) then constrains τ_s to have a field $@x$ with some type τ' which is the type of the expression. As we show later, we require field types to be invariant in our subtyping judgments. The use of subtyping here is for consistency with (Class$_\tau$) and (Call$_\tau$); note that fields always have a monotype. (Field Assign$_\tau$) is similar to (Field$_\tau$) updating the value of a field as long as the right-hand side expression has the appropriate type.

Rule (Self$_\tau$) simply looks up self, which must always be an object in Δ.

($\textsc{Class}_\tau$) looks up a class type in Ω, ensuring it has a class type. Note that this rule does not explicitly instantiate the type σ. Instead of instantiating such types in the type judgments for terms, we use a subtyping rule (($\textsc{Sub-App}_{<:}$) in Figure 3.4, discussed below) to perform the instantiation. Using a subtyping rule allows us to provide a uniform formalism for instantiating bounded polymorphic types and intersection types (which are simply finitely polymorphic types), similarly to other systems that combine intersection and polymorphic typing [40]. Thus, ($\textsc{Class}_\tau$) instantiates the type $\Omega(A)$, which may be a polytype, to a monotype by adding the subtyping constraint $\sigma <: \tau\ \mathsf{class}$.

($\textsc{New}_\tau$) creates a new instance for the class type $\tau\ \mathsf{class}$. Rule ($\textsc{Call}_\tau$) handles method calls. First, each argument is typed left-to-right, followed by the receiver e_0. We then use the subtype constraint to extract and instantiate the type of the method itself, which may have an intersection or polymorphic type.

Finally, ($\textsc{TypeCase}_\tau$) provides conditional branching in our calculus. First, we type the guard e_1, which must have an object type. We then type the "match" branch e_2 under the assumption that x is an instance of the matched class A. However, the "else" branch is typed with no additional assumptions. Both branches must be have the same type τ, which is the type of the entire expression. Since each branch may update, or introduce new term variables, we use the $\uplus$ operator to combine the types of the variables in each branch. Formally:

Definition 1

$$\begin{aligned}
\Delta_1 \uplus \Delta_2 &= \{x \mapsto \tau_1 \uplus \tau_2 \mid x \in dom(\Delta_1) \cap dom(\Delta_2) \wedge \tau_1 = \Delta_1(x) \wedge \tau_2 = \Delta_2(x)\} \\
\tau_1 \uplus \tau_2 &= \tau_1 \ \textit{if}\ \tau_1 = \tau_2 \\
\tau_1 \uplus \tau_2 &= \tau_1 \cup \tau_2 \ \textit{if}\ \tau_1 \neq \tau_2
\end{aligned}$$

This operator makes variables defined inside of a branch local to that branch and combines variables that have been updated with different types by taking their union. Therefore, only when a variable exists prior to a `typecase` will it be in scope afterwards as well.

Our type checking rules for definitions and programs are given in Figure 3.3. We explain the rules in Figure 3.3 bottom-up. Recall that programs P are made up of a series of class definitions followed by a "main" expression. Thus our judgments for P have the form $\vdash P$ since all of the environments are explicitly constructed by our typing judgments for definitions. In ($\textsc{Program}_\tau$), we initialize the initial class environment Ω_0 to the empty set, and then incrementally build the class environment by visiting each class definition in turn. Thus, each Ω_j contains the definitions of the previous $j-1$ classes. The special token *cur_class* is used to store the name and type of the current class in Ω. Each class definition is given its own scope, and thus $\emptyset$ is used for the initial Δ environment. Finally, we type the "main" expressions e using the final class environment and an empty Δ environment.

Class definitions are typed by either ($\textsc{Poly Class}_\tau$) or ($\textsc{Mono Class}_\tau$). The former recursively types the class definition by peeling away a $\forall$ quantifier and

$\boxed{\Omega;\Gamma;\Delta \vdash_m m}$

$$\frac{\tau = (\tau_1 \times \cdots \times \tau_n) \to \tau_r \qquad \Omega;\Gamma;\Delta[x_i \mapsto \tau_i] \vdash e : \tau_r;\Delta'}{\Omega;\Gamma;\Delta \vdash_m \mathsf{def}\ m(x_1,\ldots,x_n) : \tau = e}\ (\textsc{Mono Method}_\tau)$$

$$\frac{\sigma = \forall\alpha <: \tau.\sigma' \qquad \alpha \notin FV(\Gamma) \qquad \Omega;\Gamma[\alpha <: \tau];\Delta \vdash_m \mathsf{def}\ m(x_1,\ldots,x_n) : \sigma' = e}{\Omega;\Gamma;\Delta \vdash_m \mathsf{def}\ m(x_1,\ldots,x_n) : \sigma = e}\ (\textsc{Poly Method}_\tau)$$

$$\frac{\eta = \eta_1 \cap \eta_2 \qquad \Omega;\Gamma;\Delta \vdash_m \mathsf{def}\ m(x_1,\ldots,x_n) : \eta_1 = e \qquad \Omega;\Gamma;\Delta \vdash_m \mathsf{def}\ m(x_1,\ldots,x_n) : \eta_2 = e}{\Omega;\Gamma;\Delta \vdash_m \mathsf{def}\ m(x_1,\ldots,x_n) : \eta = e}\ (\textsc{Inter Method}_\tau)$$

$\boxed{\Omega;\Gamma;\Delta \vdash d}$

$$\frac{\Omega;\Gamma;\emptyset \vdash e : \tau;\Delta' \qquad \Gamma \vdash \Delta(\mathsf{self}) <: [@x : \tau]}{\Omega;\Gamma;\Delta \vdash @x = e}\ (\textsc{Field Decl}_\tau)$$

$$\frac{\Gamma \vdash \Delta(\mathsf{self}) <: [m : \eta] \qquad \Omega[cur\_class] = (A,\sigma) \qquad \Omega[A \mapsto \sigma];\Gamma;\Delta \vdash_m \mathsf{def}\ m(x_1,\ldots,x_n) : \eta = e}{\Omega;\Gamma;\Delta \vdash \mathsf{def}\ m(x_1,\ldots,x_n) : \eta = e}\ (\textsc{Method Decl}_\tau)$$

$\boxed{\Omega;\Gamma;\Delta \vdash c}$

$$\frac{\sigma = \forall\alpha <: \tau.\sigma' \qquad \alpha \notin FV(\Gamma) \qquad \Omega;\Gamma[\alpha <: \tau];\Delta \vdash \mathsf{class}\ A : \sigma' = d_1,\ldots,d_n}{\Omega;\Gamma;\Delta \vdash \mathsf{class}\ A : \sigma = d_1,\ldots,d_n}\ (\textsc{Poly Class}_\tau)$$

$$\frac{\Omega;\Gamma;[\mathsf{self} \mapsto \tau] \vdash d_i \qquad i \in 1..n \qquad \mathrm{labels}(\tau) = \{\mathrm{label}(d_1),\ldots,\mathrm{label}(d_n)\}}{\Omega;\Gamma;\Delta \vdash \mathsf{class}\ A : \tau = d_1,\ldots,d_n}\ (\textsc{Mono Class}_\tau)$$

$\boxed{\vdash P}$

$$\frac{\Omega_1 = \emptyset \qquad c_i = (\mathsf{class}\ A_i : \sigma_i = e_i) \qquad \Omega_j = \Omega_{j-1}[A_{j-1} \mapsto \sigma_{j-1}] \qquad j \in 2..n \qquad \Omega_i[cur\_class \mapsto (A_i,\sigma_i)];\emptyset;\emptyset \vdash c_i \qquad \Omega_{n+1};\emptyset;\emptyset \vdash e : \tau;\Delta \qquad i \in 1..n}{\vdash c_1 \cdots c_n, ; e}\ (\textsc{Program}_\tau)$$

Figure 3.3: Type Checking Rules for Definitions

adding the corresponding subtyping assumption to Γ. Once any quantifiers have been instantiated in Γ, class bodies are typed using (Mono Class$_\tau$). Here, we create an initial local environment containing only self, mapping to the current monomorphic class type, and type each element of the class body. The last hypothesis in (Mono Class$_\tau$) ensures that the class definition is complete. That is, it requires that the body of the class define all of the fields and methods (termed "labels") present in τ and that no label is defined twice.

Rule (Field Decl$_\tau$) types field declarations. The first hypothesis checks the value of the field with an empty environment so that the right hand side expression may not reference self as the class is still being defined. The second hypothesis then ensures that the type of the expression matches the expected type for the field in self.

Method declarations are similar: we ensure that the current class has a compatible type for m by constraining self. Unlike fields, however, we allow method bodies to reference the current class. Therefore, we type add the current class to Ω and type the method declaration with Δ instead of $\emptyset$. The method definition is then typed using an auxiliary judgment $\vdash_m$ using (Inter Method$_\tau$), (Poly Method$_\tau$), or (Mono Method$_\tau$) depending on the form of η.

If m has the intersection type $\eta = \eta_1 \cap \eta_2$, then m must have both type η_1 and type η_2. Thus (Inter Method$_\tau$) recursively types the method under each case. If m is a polymorphic type $\forall\alpha <: \tau.\tau'$, then similar to (Poly Class$_\tau$), we add the subtyping assumption to Γ and recurse to ensure m is well-typed under τ'.

Finally, we type m with a monotype using (Mono Method$_\tau$). We first ensure

that the type τ is in fact a function type with the correct arity. We then type the body e adding the formal arguments to Δ, which already contains self from (Mono Class$_\tau$).

Our subtyping rules are given in Figure 3.4. Rules (Refl$_{<:}$), (Trans$_{<:}$), and (Top$_{<:}$) are standard. (Var$_{<:}$) simply checks that the variable is bound in Γ. (Fun$_{<:}$) includes the standard contravariance for function parameters and covariance for function return types. Tuples are covariant in their elements as they are immutable. When comparing two bounded polymorphic types, we use the "kernel" rule [19], which requires the subtype bounds to be equal to avoid undecidability.

If a union type appears on the left of <:, (UnionL$_{<:}$) requires that each component must be a subtype of the type on the right-hand side. Howver, if the union appears on the right, (UnionR$_{<:}$) requires only one type to match. Conversely, (InterL$_{<:}$) only requires one type to match when an intersection occurs on the left, and (InterR$_{<:}$) requires both components to match.

The rule (Object$_{<:}$) gives the subtyping rule for objects. Here, the object on the left may include additional fields and methods not found on the right (notice the width-subtyping via $n \geq p$ and $r \geq q$). Any fields that common to both objects must have the same type (field types are *invariant*). However, methods are immutable, and we therefore allow method types to be subtypes (i.e., they use depth subtyping).

Finally, one can view the (InterL$_{<:}$) rule for intersection types as an elimination form for intersection types. We include a similar rule that instantiates $\forall$ quantifiers with rule (App$_{<:}$). Intuitively, this rule says that a generalization is a

$\boxed{\Gamma \vdash \eta <: \eta}$

$$\frac{}{\Gamma \vdash \eta <: \eta}\ (\textsc{Refl}_{<:}) \qquad \frac{\Gamma \vdash \eta_1 <: \eta_2 \quad \Gamma \vdash \eta_2 <: \eta_3}{\Gamma \vdash \eta_1 <: \eta_3}\ (\textsc{Trans}_{<:}) \qquad \frac{}{\Gamma \vdash \eta <: \top}\ (\textsc{Top}_{<:}) \qquad \frac{\alpha <: \tau \in \Gamma}{\Gamma \vdash \alpha <: \tau}\ (\textsc{Var}_{<:})$$

$$\frac{\Gamma \vdash \tau_i' <: \tau_i \qquad i \in 1..n \qquad \Gamma \vdash \tau_{n+1} <: \tau_{n+1}'}{\Gamma \vdash (\tau_1 \times \cdots \times \tau_n) \to \tau_{n+1} <: (\tau_1' \times \cdots \times \tau_n') \to \tau_{n+1}'}\ (\textsc{Fun}_{<:})$$

$$\frac{\Gamma \vdash \tau_i <: \tau_i'}{\Gamma \vdash (\tau_1 \times \cdots \times \tau_n) <: (\tau_1' \times \cdots \times \tau_n')}\ (\textsc{Tuple}_{<:}) \qquad \frac{\Gamma, \alpha : \tau \vdash \sigma <: \sigma'}{\Gamma \vdash (\forall \alpha <: \tau.\sigma) <: (\forall \alpha <: \tau.\sigma')}\ (\textsc{Kernel}_{<:})$$

$$\frac{\Gamma \vdash \tau_1 <: \tau_3 \qquad \Gamma \vdash \tau_2 <: \tau_3}{\Gamma \vdash \tau_1 \cup \tau_2 <: \tau_3}\ (\textsc{UnionL}_{<:}) \qquad \frac{\Gamma \vdash \tau_1 <: \tau_i \qquad \text{for some } i \in \{2,3\}}{\Gamma \vdash \tau_1 <: \tau_2 \cup \tau_3}\ (\textsc{UnionR}_{<:})$$

$$\frac{\Gamma \vdash \eta_i <: \eta_3 \qquad \text{for some } i \in \{1,2\}}{\Gamma \vdash \eta_1 \cap \eta_2 <: \eta_3}\ (\textsc{InterL}_{<:}) \qquad \frac{\Gamma \vdash \eta_1 <: \eta_2 \qquad \Gamma \vdash \eta_1 <: \eta_3}{\Gamma \vdash \eta_1 <: \eta_2 \cap \eta_3}\ (\textsc{InterR}_{<:})$$

$$\frac{n \geq p \qquad r \geq q \qquad \tau_i = \tau_i' \qquad \Gamma \vdash \eta_k <: \eta_k'}{\Gamma \vdash [@x_i : \tau_i, m_j : \eta_j]^{i \in 1..n, j \in 1..r} <: [@x_k : \tau_k', m_l : \eta_l']^{k \in 1..p, l \in 1..q}}\ (\textsc{Object}_{<:})$$

$$\frac{\Gamma \vdash \tau <: \tau'}{\Gamma \vdash \tau\ \mathsf{class}\ <: \tau'\ \mathsf{class}}\ (\textsc{Class}_{<:}) \qquad \frac{\Gamma \vdash \tau_1' <: \tau_1 \qquad \Gamma \vdash \tau[\alpha/\tau_1'] <: \tau'}{\Gamma \vdash (\forall \alpha <: \tau_1.\tau) <: \tau'}\ (\textsc{App}_{<:})$$

Figure 3.4: Subtyping Judgments

$$\frac{\Omega;\Gamma;\Delta \vdash \mathsf{self} : \tau;\Delta \qquad \Gamma \vdash \tau <: [@x : \alpha] \qquad \alpha \text{ fresh}}{\Omega;\Gamma;\Delta \vdash @x : \alpha;\Delta}\ (\textsc{Field}'_\tau)$$

$$\frac{\Omega;\Gamma;\Delta_i \vdash e_i : \tau_i;\Delta_{i+1} \qquad i \in 1..n \qquad \Omega;\Gamma;\Delta_{n+1} \vdash e_0 : \tau_0;\Delta_{n+2} \qquad \Gamma \vdash \tau_0 <: [m : (\tau_1 \times \cdots \times \tau_n) \to \alpha] \qquad \alpha \text{ fresh}}{\Omega;\Gamma;\Delta_0 \vdash e_0.m(e_1,\ldots,e_n) : \alpha;\Delta_{n+2}}\ (\textsc{Call}'_\tau)$$

$$\frac{\Gamma \vdash \beta <: \tau_1 \qquad \beta \text{ fresh} \qquad \Gamma \vdash \tau[\alpha/\beta] <: \tau'}{\Gamma \vdash \forall\alpha <: \tau_1.\tau <: \tau'}\ (\textsc{App}'_{<:})$$

$$\frac{}{\Gamma \vdash \tau <: \tau \cup \tau'}\ (\textsc{UnionR}'_{<:})$$

$$\frac{\Gamma \vdash \tau_{n+1} <: \tau'_{n+1}}{\Gamma \vdash (\tau_1 \times \cdots \times \tau_n) \to \tau_{n+1} \cap \eta <: (\tau_1 \times \cdots \times \tau_n) \to \tau'_{n+1}}\ (\textsc{InterL}'_{<:})$$

Figure 3.5: Type Inference Rules (Updates Only)

subtype of any valid instantiation of the type.

We have proven our type system sound in Appendix B. The operational semantics use two stores: S for values stored in the heap and V for local variables and self. Our semantics then define a reduction relation of the form $\langle S, V, e\rangle \to \langle S', V', v\rangle$. Here, an expression e in state S and V reduces to a value v and yields new stores S' and V'. Our proof verifies that every well-typed program always reduces to a value, i.e., it does not go wrong. Formally:

Theorem 2 (Type Soundness) *If* $\vdash P$ *then* $\langle \emptyset, \emptyset, P\rangle \to \langle S, V, r\rangle$ *where* $r \neq error$.

3.3.3 Type Inference

Our type inference system is constructed as a *constraint-based analysis*. We first traverse the entire program (including `base_types.rb`), visiting each statement once

and generating subtyping constraints between program expressions. Our inference rules are very similar to the checking rules presented in Section 3.3 and therefore we present only the updated rules in Figure 3.5.

Subtyping constraints are generated by rules (FIELD'$_\tau$) and (CALL'$_\tau$) by using fresh type variables for the types of fields and method return values. For example, if we see $\mathsf{x.m()}$, then (CALL'$_\tau$) requires that x have method $\mathsf{m}()$ by generating the constraint: $\Gamma \vdash \tau_x <: [m : () \to \alpha]$, meaning the type of x is a subtype of an object type with an m method, but with an as-yet-unknown return type α. Similarly, we instantiate polymorphic types with a fresh type variable α using (APP'$_{<:}$). Note that in general, $\Gamma \vdash \alpha \not<: \beta$ when α and β are fresh type variables.

We resolve the generated set of constraints by exhaustively applying a set of rewrite rules to ensure the subtyping judgments in Γ are *consistent* [26]. For example, given $\Gamma \vdash \tau <: \alpha$ and $\Gamma \vdash \alpha <: \tau'$, we add the closure constraint $\Gamma \vdash \tau <: \tau'$, implementing (TRANS$_{<:}$). During this process, we issue a warning if any constraints are immediately inconsistent by violating one of our subtyping judgments. For example, if $A <: [m : \eta]$ and class A has no method m, then (OBJECT$_{<:}$) is violated and we have found a type error. If we detect no errors, the constraints are satisfiable, and we have found a valid typing for the program.

When solving a constraint with unions on the right hand side, e.g., $\Gamma \vdash \tau_1 <: \tau_2 \cup \tau_3$, we require τ_1 to have a fully-resolved type that is equal to (not a subtype of) either τ_2 or τ_3 by rule (UNIONR'$_{<:}$). These restrictions mean DRuby cannot find a solution to all satisfiable constraint systems, but keeps constraint solving tractable. We place a similar restriction on method parameters in constraints with intersection

on the left-hand side using rule (InterL'$_{<:}$).

Finally, note that none of our inference rules are able to generalize type variables or produce an intersection type. Therefore, a class or method can only be given a non-monotype by using an explicit annotation. We do this for two reasons. First, since we have higher-order polymorphism, inference can easily become undecidable [82]. Second, since we do not have principle types, our inference system might generate arbitrarily complex intersection types that would likely be incomprehensible by the programmer.

3.4 Implementation

We have implemented the type system described in Section 3.3 as part of DRuby. DRuby is a drop-in replacement for Ruby: the programmer invokes "`druby` *filename*" instead of "`ruby` *filename*." DRuby also accepts several custom options to control its behavior, e.g., to specify the location of `base_types.rb`. Another option instructs DRuby to execute the script with Ruby after performing its analysis (even in the presence of static type errors), which can be useful for testing specific execution paths during development. When DRuby is run, it first loads in the library annotations in `base_types.rb`, and then analyzes the "main" program passed as a command line argument. Ruby programs can load code from other files by invoking either require or load. When DRuby sees a call to one of these methods, it analyzes the corresponding file if the name is given by a string literal, and otherwise DRuby issues an error. In Chapter 4 we will present a technique that allows DRuby to

precisely determine which files are loaded by require.

Language constructs like new and include, which appear to be primitives, are actually method calls in Ruby. However, because they implement fundamental language operations, DRuby recognizes calls to these methods syntactically and models them specially, e.g., creating a new class instance, or adding a mixin module. Thus if a Ruby program redefines some of these methods (a powerful metaprogramming technique), DRuby may report incorrect results. DRuby also has special handling for the methods attr, attr_accessor, attr_writer, and attr_reader, which create getter and/or setter methods named according to their argument. Finally, while MiniRuby included a syntactic type case construct, Ruby has no single distinguished form for this operation. Instead, there are a myriad of ways to perform a type test using a variety of techniques. We leave handling of these forms for future work (Chapter 5), and instead rely on dynamic checks to verify method bodies that perform type tests.

3.5 Experimental Evaluation

We evaluated DRuby's core static type system by applying it to a suite of programs gathered from our colleagues and RubyForge. We believe these programs to be representative of the kind of small-to-medium sized scripts that people write in Ruby and that can benefit from the advantages of static typing. We will explore larger benchmarks once we add support for dynamic features in Chapter 4. The left portion of Figure 3.6 lists the benchmark names, their sizes as computed by

Program	LOC	Changes	Tm(s)	E	W	FP
pscan-0.0.2	29	None	3.7	0	0	0
hashslice-1.0.4	91	S	2.2	1	0	2
sendq-0.0.1	95	S	1.9	0	3	0
merge-bibtex	103	None	2.6	0	0	0
substitution_solver-0.5.1	132	S	2.5	0	4	0
style-check-0.11	150	None	2.7	0	0	0
ObjectGraph-1.0.1	153	None	2.3	1	0	1
relative-1.0.2	158	S	2.3	0	1	5
vimrecover-1.0.0	173	None	2.8	2	0	0
itcf-1.0.0	183	S	4.7	0	0	1
sudokusolver-1.4	201	R-1, S	2.7	0	1	1
rawk-1.2	226	None	3.1	0	0	2
pit-0.0.6	281	R-2, S	5.3	0	0	1
rhotoalbum-0.4	313	None	12.6	0	1	0
gs_phone-0.0.4	827	S	37.2	0	0	0
StreetAddress-1.0.1	877	R-1, S	6.4	0	0	0
ai4r-1.0	992	R-10, S	12.2	1	6	1
text-highlight-1.0.2	1,030	M-2, S	14.0	0	0	2

M–expanded meta-programming code
R–require with non-constant String
S–simulate unit test
E–Errors
W–Warnings
FP–False Positives

Figure 3.6: Type Inference Results

SLOCCount [83], and the number and kinds of changes required to be analyzable by DRuby (discussed below). The analyzed code includes both the application or library and any accompanying test suite.

In this chapter, we will not analyze the standard library due to its frequent use of dynamic language features. Instead, we created a stub file with type annotations for the portion of the standard library, 120 methods in total, used by our benchmarks. Surprisingly, we found that although the standard library itself is quite dynamic, the APIs revealed to the user very often have precise static types in DRuby. These stub files were the only places we added type annotations in our benchmarks, and were therefore all checked dynamically. For those benchmarks that supplied test suites (8 of the 18), we ran the test suite with and without dynamic annotation checking enabled. None of the dynamic checks failed, and the overhead was minimal. Five test suites took less than one second with and without dynamic checking. The remaining three test suites ran in 2, 6, and 53 seconds and had overheads of 7.7%, 0.37%, and -12% respectively. The last test suite actually ran faster with our instrumentation, likely due to interactions with the Ruby interpreter's method cache.

We made three kinds of changes to the benchmarks to analyze them. First, the benchmarks labeled with "R" included require calls that were passed dynamically constructed file names. For instance, the test suite in the *StreetAddress* benchmark contained the code

```
1 require File.dirname(__FILE__) + '../lib/street_address'
```

In the benchmarks that used this technique, we manually replaced the argument of require with a constant string.

Second, several of the benchmarks used a common unit testing framework provided by the standard library. This framework takes a directory as input, and invokes any methods prefixed by test_ contained in the Ruby files in the directory. To ensure we analyzed all of the testing code, we wrote a stub file that manually required and executed these test methods, simulating the testing harness. Benchmarks with this stub file are labeled with "S."

Finally, the *text-highlight* benchmark used metaprogramming (as shown in Section 3.2) to dynamically generate methods in two places. DRuby did not analyze the generated methods and hence thought they were missing, resulting in 302 false positives the first time we ran it. We manually replaced the metaprogramming code with direct method creation, eliminating this source of imprecision. This manual transformation directly motivated our solution to handling dynamic features in general, which we discuss in Chapter 4.

3.5.1 Experimental Results

The right portion of Figure 3.6 shows the running times for DRuby, the number of errors, the number of warnings, and the number of false positives. Times were the average of 5 runs on an AMD Athlon 4600 processor with 4GB of memory. We define *errors* (E) as source locations that may cause a run-time type error, given appropriate input, and *warnings* (W) as locations that do not cause a run-time error

for any input, but involve questionable programming practice. *False positives* (FP) are locations where DRuby falsely believes a run-time error can occur, but the code is actually correct.

For these 18 programs, DRuby runs quickly (under 7 seconds except four mid-size benchmarks), producing 37 messages that we classified into 5 errors, 16 warnings, and 16 false positives.

Errors The 5 errors discovered by DRuby occurred in 4 benchmarks. The error in *ai4r* was due to the following code:

```
return rule_not_found if !@values.include?(value)
```

There is no `rule_not_found` variable in scope at this point, and so if the condition ever returns true, Ruby will signal an error. Interestingly, the *ai4r* program includes a unit test suite, but it did not catch this error because it did not exercise this branch. The two errors in *vimrecover* were also due to undefined variables, and both occurred in error recovery code.

The error in *hashslice* is interesting because it shows a consequence of Ruby's expressive language syntax, which has many subtle disambiguation rules that are hard to predict. In *hashslice*, the test suite uses assertions like the following:

```
assert_nothing_raised { @hash['a', 'b'] = 3, 4 }
assert_kind_of (Fixnum, @hash['a', 'b'] = 3, 4)
```

The first call passes a code block whose body calls the `[]=` method of `@hash` with the argument `[3,4]`. The second call aims to do a similar operation, verifying the return value is a `Fixnum`. However, the Ruby interpreter (and DRuby) parse the second

method call as having three arguments, not two as the author intended (the literal 4 is passed as the third argument). In the corresponding standard library stub file, this method is annotated as the following:

```
assert_kind_of <t> : (Class, t, ?String) → NilClass
```

Because 4 is a Fixnum and not a String, DRuby considers this a type error. Although our type annotation matches the documentation for assert_kind_of, the actual implementation of the method coerces its third argument to a string, thus masking this subtle error at run time.

The last error is in *ObjectGraph*, which contains the following:

```
1  $baseClass = ObjectSpace.each_object(Class)
2    { |k| break k if k.name == baseClassName }
```

The method each_object takes a code block, applies it to each element of the collection, and returns the total number of elements visited (a Fixnum). However, the block supplied above terminates the iteration early (**break** k), returning a specific element that has type Class. The programmer intended the loop to always terminate in this fashion, using $baseClass throughout the program as a Class, not a Fixnum. However, it is easy to make the loop terminate normally and therefore return a Fixnum, since the above code depends on data provided by the end user.

Warnings 14 of the 16 reported warnings are due to a similar problem. Consider the code "5.times { |i| print "∗" }", which prints five stars: The times function calls the code block n times (where n is the value of the receiver), passing the values $0..n-1$ as arguments. In the above code, we never used parameter i, but we still

included it. However, Ruby allows blocks to be called with too many arguments (extra arguments are ignored) or too few (missing arguments are set to **nil**). We think this is bad practice, and so DRuby reports an error if blocks are called with the wrong number of arguments, resulting in 14 warnings. In DRuby, if the user intends to ignore block parameters, they can supply an anonymous vararg parameter such as `5.times {|*| ...}` to suppress the warning.

Of the two remaining warnings, one occurs in *substitution_solver*, which contains the block arguments |tetragraph, freq| instead of |(tetragraph, freq)|. In the former case, DRuby interprets the block as taking two arguments, whereas the latter is a single (tuple) argument. As `each` calls the block with only a single argument, DRuby signals an error. As it turns out, Ruby is fairly lenient and allows the programmer to omit the parentheses in this case. However, we feel this leads to confusing code, and so DRuby always requires parentheses around tuples used in pattern matching.

The last warning occurs in *relative*, which, similarly to *ObjectGraph*, calls an iterator where all executions are intended to exit via a **break** statement. In this case, the normal return path of the iterator block appears to be infeasible, and so we classify this as a warning rather than an error.

False positives DRuby produced a total of 16 false positives, due to several causes. Three false positives are due to union types that are discriminated by run-time tests. For example, one of the methods in the *sudokusolver* benchmark returns either **false** or an array, and the clients of this method check the return

values against **false** before using them as arrays. DRuby does not model type tests to discriminate unions, something we plan on addressing in future work (Chapter 5).

Three additional false positives occurred because a benchmark redefined an existing method with a different type, which DRuby forbids since then it cannot tell at a call site whether the previous or the redefined method is called. (Unlike statically typed object-oriented languages, redefined methods in Ruby can have radically different types.)

The remaining false positives occurred because DRuby could not resolve the use of an intersection type; because DRuby could not locate the definition of a constant; because of a wrapper around require that dynamically changed the argument; and because of rebinding of the new method.

3.6 Related Work

Many researchers have previously studied the problem of applying static typing to dynamic languages. Some of the earliest work in this area is *soft typing* for Scheme, which uses a set-based analysis to determine what data types may reach the destructors in a program [20, 6, 85, 30]. Typed Scheme adds type annotations and type checking to Scheme [77]. One of the key ideas in this system is *occurrence types*, which elegantly model type testing functions used to discriminate elements of union types. As mentioned earlier, DRuby support for modeling type tests would be useful future work.

Several researchers investigated static typing for Smalltalk, a close relation of

Ruby. Graver and Johnson [33] propose a type checking system that includes class, object, union, and block types. Strongtalk [70], a variant of Smalltalk extended with static types, has a similar system with additional support for mixins [11]. Spoon [67] and RoelTyper [87] each a present type system that trades precision for scalability. Agesen et al. [4, 3] explored type inference for Self, which is an object-based (rather than class-based) language. DRuby differs from these system as it integrates both static type inference and dynamically checked annotations.

Type inference for Ruby has been proposed before. Kristensen [44] developed a Ruby type inference system based on the cartesian product algorithm. However, their system does not yet support the full Ruby language, including complete support for code blocks. They also do not include a type annotation language or support for the same features as DRuby's type system such as F-bounded polymorphism. Morrison [49] developed a type inference algorithm that has been integrated into RadRails, an IDE for Ruby on Rails. In RadRails, type inference is used to select the methods suggested during method completion. There is no formal description of RadRails's type inference algorithm, but it appears to use a simple intraprocedural dataflow analysis, without support for unions, object types, parameter polymorphic, tuples, or type annotations.

Several type systems have been proposed to improve the performance of dynamic languages. The CMUCL[47] and SBCL[64] Lisp compilers use type inference to catch some errors at compile time, but mainly rely on inference to eliminate run-time checks. Similarly, aggressive type inference [10], Starkiller [62], and a system proposed by Cannon [17] all infer types for Python code to improve performance.

RPython is a statically-typed subset of Python designed to compile to JVM and CLI bytecode [8]. RPython includes type inference, though it is unclear exact what typing features are supported. One interesting feature of RPython is that it performs type inference after executing any load-time code, thus providing some support for metaprogramming. This work motivated our solution to handling metaprogramming in Ruby (Chapter 4).

There are several proposals for type inference for Javascript [9, 73]. The proposed ECMAScript 4 (Javascript) language includes a rich type annotation language with object types, tuple types, parametric polymorphism, and union types [35]. The challenges in typing Ruby are somewhat different than for Javascript, which builds objects by assigning pre-methods to them rather than using classes.

Aside from work on particular dynamic languages, the question of statically typing dynamic language constructs has been studied more generally. Abadi et al. [2] propose adding a type Dynamic to an otherwise statically typed language. Quasi-static typing takes this basic idea and makes type coercions implicit rather than explicit [71]. Gradual type systems improve on this idea further [66, 36], and have been proposed for object-oriented type systems [65]. Sage mixes a very rich static type system with the type Dynamic [34]. Tobin-Hochstadt and Felleisen [76] present a framework for gradually changing program components from untyped to typed.

Finally, our run-time type checks are based heavily on contracts, which were developed as the dynamic counterpart to static types to generalize and extend run-time assertions [29, 27]. An important property of contracts is to track *blame*

through a program to ensure that when a contract is violated, the cause of the failure is correctly correlated with the proper syntactic origin, which can be tricky in the higher-order case [28].

Chapter 4

Analyzing Dynamic Features

In the previous chapter, we showed that DRuby can successfully infer types for small Ruby scripts. However, there is a major challenge in scaling up static typing to large script programs: Ruby includes a range of hard-to-analyze, highly dynamic constructs. In addition to eval, Ruby includes other constructs such as allowing programmers to use reflection to invoke methods via send, and define a method_missing method to handle calls to undefined methods. These kinds of features lend themselves to a range of terse, flexible, and expressive coding styles, but they also impede standard static analysis. In fact, in Ruby it is even hard to statically determine what source files to analyze, because scripts can perform computation to decide what other files to load.

In this chapter, we solve this problem by combining run-time profiling of dynamic features with static typing. We begin with several real-world examples in Section 4.1 that motivate our solution. We then give a brief, high-level overview of our approach in Section 4.2 before formally presenting the details using DynRuby, a small object-oriented language with eval, send, and method_missing in Section 4.3.

Next, we describe our implementation in Section 4.4. To evaluate our tech-

```
1 require  File.join(File.dirname(__FILE__),'..',
2                    'lib','sudokusolver')
3
4 Dir.chdir("..") if base == "test"
5 $LOAD_PATH.unshift(Dir.pwd + "/lib")
6 ...
7 require "memoize"
```

Figure 4.1: Using require with dynamically computed strings

nique, we applied DRuby to a suite of benchmarks that use dynamic features, either directly, via the standard library, or via a third-party library. Sections 4.5 and 4.6 describe our experience profiling dynamic constructs in our benchmarks, and in applying DRuby's type inference algorithm to these programs. Lastly, we describe potential threats of validity to our results in Section 4.7, conclude with related work in Section 4.8.

4.1 Motivation

In this section, we motivate the need for our solution by giving examples showing uses of Ruby's dynamic features. All of the examples in this section were extracted from the benchmarks in Section 4.5. DRuby also handles several other dynamic features of Ruby, discussed in Section 4.4.

Require To load code stored in a file, a Ruby program invokes the require method, passing the file name as a string argument. Since this is an ordinary method call, a Ruby program can actually perform run-time computation to determine which file to load. Figure 4.1 gives two examples of this. Lines 1–2, from the *sudokusolver*

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

Figure 4.2: Example of require from *Rubygems* package manager

benchmark, call dirname to compute the directory containing the currently executing file, and then call File.join to create the path of the file to load. We have found similar calls to require (with computed strings) are common, occurring 11 times across 5 of our benchmarks. As another example, lines 4–7, from the *memoize* benchmark, first modify the load path on line 5 before loading the file memoize on line 7. This example shows that even when require is seemingly passed a constant string, its behavior may actually vary at run time.

For a much more complex use of require, consider the code in Figure 4.2. This example comes from *Rubygems*, a popular package management system for Ruby. In *Rubygems*, each package is installed in its own directory. *Rubygems* redefines the require method, as shown in the figure, so that require'ing a package loads it from the right directory. Line 1 makes an alias of the original require method. Then lines 3–11 give the new definition of require. First, line 4 attempts to load the file normally, using the old version of require. If that fails, the resulting LoadError exception is caught on line 5 and handled by lines 6–11. In this case, *Rubygems* searches the file

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

Figure 4.3: Use of send to initialize fields

system for a library of the same name (line 6). If found, the package is "activated" on line 7, which modifies the load path (as in Figure 4.1), and then the file is loaded with the old require call on line 8.

This implementation is convenient for package management, but it makes pure static analysis quite difficult. Even if we could statically determine what string was passed to the new version of require, to find the corresponding file we would need to reimplement the logic of the Gem.searcher.find method.

Send When a Ruby program invokes e_0.send("meth", $e_1, \ldots, e_n$), the Ruby interpreter dispatches the call reflectively as e_0.meth($e_1, \ldots, e_n$). Figure 4.3 shows a typical use of this feature, from the *StreetAddress* benchmark. This code defines a constructor initialize that accepts a hash args as an argument. For each key attrib in the hash, line 3 uses send to pass args[attrib], the value corresponding to the key, to the method named "#{attrib} =", where #{e} evaluates expression e and inserts the resulting value into the string. For example, if initialize is called with the argument {"x" $\Rightarrow$ 1}, it will invoke the method self.x=(1), providing a lightweight way to configure a class through the constructor.

Another common use of send is in test drivers. For example, the Ruby community makes heavy use of Ruby's standard unit testing framework (not shown). To

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

Figure 4.4: Defining methods with eval

write a test case in this framework, the programmer creates a class with test methods whose names begin with test_. Given an instance of a test class, the framework uses the methods method to get a string list containing the names of the object's methods, and then calls the appropriate ones with send.

Eval Ruby also provides an eval method that accepts a string containing arbitrary code that is then parsed and executed. Our experiments show that use of eval is surprisingly common in Ruby—in total, eval and its variants are used to evaluate 423 different strings across all our benchmark runs (Section 4.5). Figure 4.4 shows one example of metaprogramming with eval, taken from the *text-highlight* benchmark (as previously shown in Section 3.2.12). This code iterates through the ATTRIBUTES array defined on line 1, creating a method named after each array element on lines 3–4. We found many other examples like this, in which Ruby programmers use eval to create methods via macro-style metaprogramming.

Method Missing Figure 4.5 gives an example use of method_missing, which receives calls to undefined methods. This code (slightly simplified) is taken from the *ostruct* library, which creates record-like objects. In this definition, line 2 converts the first argument, the name of the invoked method, from a symbol to a string

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

Figure 4.5: Intercepting calls with method_missing

mname. If mname ends with = (line 3), then on line 5 we update @table to map mname (with the = removed and interned back into a symbol) to the first argument. Otherwise there must be no arguments (line 6), and we read the value corresponding to the invoked method out of @table. For example, if o is an instance of the *ostruct* class, the user can call o.foo = (3) to "write" 3 to foo in o, and o.foo() to "read" it back. Notice that we can use method invocation syntax even though method foo was never defined. This particular use of method_missing from *ostruct* is one of two occurrences of method_missing that are dynamically executed by our benchmark test suites.

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

4.2 Overview

Our key insight is that even though the examples in the previous section use constructs that *appear* to be dynamic, in fact their use is almost always heavily constrained, so that in practice they *act* statically. As an extreme example, a call eval "x + 2" is morally the same as the expression x + 2, and can be typed just as easily with DRuby. Thus, the goal of our approach is to determine the restricted ways in which these constructs act, and to replace them with forms that are more amenable to static analysis.

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

Notice that DRuby relies on the programmer to provide test cases to guide

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

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

Figure 4.6: DynRuby source language

profiling. We think this is a reasonable approach because not only do most Ruby programs already come with test suites (testing is widely adopted in the Ruby community), but it gives the programmer an easy to understand trade-off: The more dynamic features covered in the profile, the more static checking is achieved. Moreover, run-time profiling gives DRuby very precise information for type inference. This is in contrast to using, e.g., purely static string analysis [46, 21, 32], which could easily over-approximate the set of strings seen at run time [63]. It also allows us to statically analyze effectful dynamic code. For example, in our experiments, we found many cases where eval'd strings define methods, and those methods are referred to in other parts of the program. As far as we are aware, techniques such as gradual typing [66, 65, 36] would be unsound in the presence of such effects in dynamic code—static guarantees could be undermined if dynamically eval'd code overwrites a method used in statically typed code.

4.3 Dynamic Features in DynRuby

We model our approach to statically type checking dynamic language features with DynRuby, shown in Figure 4.6. The core language includes local variables x (such as the distinguished local variable self) and *values* v. Values include strings s, booleans true and false, objects created with $\mathsf{new}\ A$, and *wrapped* values $[\![v]\!]_\ell$, which indicate values with dynamic rather than static types. We annotate $[\![v]\!]_\ell$ with a *program location* ℓ so that we may later refer to it. In DynRuby, objects do not contain fields or per-object methods, and so we can represent an object simply by its class name. We could add richer objects to DynRuby, similarly to MiniRuby, but we keep the language simple to focus on its dynamic features.

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

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

The call to $x.m()$ is valid because $A.m()$ was defined before the call, even though the definition was not in effect at $\mathsf{new}\ A$. This mimics the behavior of Ruby, in

which changes to classes affect all instances, and allows eval to be used for powerful metaprogramming techniques, as shown in Figure 4.4. Our method definition syntax also allows defining the special method_missing method for a class, which, as we saw in Section 4.1, receives calls to non-existent methods.

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

DynRuby also includes two additional dynamic constructs we saw in Section 4.1. The expression $\mathsf{eval}_\ell\ e$ evaluates e to produce a string s, and then parses and evaluates s to produce the result of the expression. The expression $e_0.\mathsf{send}_\ell(e_1, \ldots, e_n)$ evaluates e_1 to a string and then invokes the corresponding method of e_0 with arguments e_2 through e_n. We annotate both constructs with a program location ℓ.

The last three expressions in DynRuby, $\mathsf{safe\_eval}_\ell\ e$, $[\![e]\!]_\ell$, and $\mathsf{blame}\ \ell$, are used to support dynamic typing and blame tracking. These expressions are inserted by our translation below to handle uses of dynamic constructs we cannot fully resolve with profiling. Our approach is somewhat non-standard, but these constructs in our formalism closely match our implementation (Section 4.4), which performs blame tracking without modifying the Ruby interpreter. We delay discussing the details of these expressions to Section 4.3.3.

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

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

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

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

Figure 4.7: Instrumented operational semantics (partial)

4.3.1 An Instrumented Semantics

To track run-time uses of eval, send, and method_missing, we use the instrumented big-step operational semantics shown in Figure 4.7. Since most of the rules are straightforward, we show only selected, interesting reduction rules, and similarly for the other formal systems we discuss below. The complete set of rules and proofs are presented in Appendix C. In our implementation, we add the instrumentation suggested by our semantics via a source-to-source translation.

Reduction rules in our semantics have the form $\langle M,\, V, e\rangle \to \langle M', \mathcal{P}, v\rangle$. Here M and M' are the initial and final *method tables*, containing a list of method definitions; V is a *local variable environment*, mapping variables to values; e is the expression

being reduced; v is the resulting value; and $\mathcal{P}$ is a *profile* that maps program locations (occurrences of eval, send, and method_missing definitions) to sets of strings. In these rules, we use $parse(s)$ to denote the expression produced by parsing string s, and we use $unparse(e)$ to denote the string produced by unparsing e.

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

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

The last three rules in Figure 4.7 handle the novel features of DYNRUBY. (EVAL) reduces its argument e to a string s, parses s and then reduces the resulting expression to compute the final result v. The resulting profile is the union of the profiles $\mathcal{P}_1$ (from evaluating e), $\mathcal{P}_2$ (from evaluating $parse(s)$), and $[\ell \mapsto s]$, which means s should be added to the set of strings associated with ℓ. In this way, we track the relationship between eval_ℓ e and the string s passed to it a run-time.

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

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

4.3.2 Translating Away Dynamic Features

After profiling, we can translate a DynRuby program into a simpler form that eliminates features that are hard to analyze statically. Figure 4.8 gives a portion of our translation. Excluding the final rule, our translation uses judgments of the form $\mathcal{P} \vdash e \rightsquigarrow e'$, meaning given profile $\mathcal{P}$, we translate expression e to expression e'. For most language forms, we either do nothing, as in ($\text{Refl}_{\rightsquigarrow}$), or translate subexpressions recursively, as in ($\text{Seq}_{\rightsquigarrow}$); we omit other similar rules.

The first interesting rule is ($\text{Eval}_{\rightsquigarrow}$), which translates $\mathsf{eval}_\ell\ e$. First, we recursively translate e. Next, recall that (Eval) in Figure 4.7 includes in $\mathcal{P}(\ell)$ any strings evaluated by this occurrence of eval. We parse and translate those strings s_j to yield

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

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

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

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

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

Figure 4.8: Transformation to static constructs (partial)

expressions e_j. Then we replace the call to eval by a conditional that binds e' to a fresh variable x (so that e' is only evaluated once) and then tests x against the strings in $\mathcal{P}(\ell)$, yielding the appropriate e_j if we find a match. If not, we fall through to the last case, which evaluates the string with $\mathsf{safe\_eval}_\ell\ x$, a "safe" wrapper around eval that adds additional dynamic checks we describe below (Section 4.3.3). This catch-all case allows execution to continue even if we encounter an unprofiled string, and also allows us to blame the code from location ℓ if it causes a subsequent run-time type error. In our formalism, adding the form blame ℓ allows us to formally state soundness: DynRuby programs that are profiled, transformed, and type checked never go wrong at run time, and reduce either to values or to blame. In practice, by tracking blame we can also give the user better error messages.

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

(Meth-Missing$_{\rightsquigarrow}$) follows a similar pattern. First, we recursively translate the body as e'. For each string s_j in $\mathcal{P}(\ell)$ (which by (Call-M) in Figure 4.7 contains the methods intercepted by this definition), we define a method named s_j that takes all but the first argument of method_missing. The method body is e', except we bind x_1, the first argument, to s_j, since it may be used in e'.

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

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

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

The challenge here is that the definition of $A.m$ might occur under complex cir-

cumstances, e.g., under a conditional, or deep in a method call chain. To ensure (new A).$m(\ldots)$ is valid, we must know $A.m$ has been defined.

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

Instead, we assume that any method syntactically present in the source code is available everywhere and rely on dynamic, rather than static, checking to find violations of our assumption. Translation $\mathcal{P} \vdash e \rightrightarrows (e_d; e')$, defined by (Prog$_{\rightsquigarrow}$) in Figure 4.8, enforces this discipline. Here e_d is a sequence of method definitions, and e' is the translation of e using the other rules. For each definition of $A.m$ occurring in e', we add a mock definition of $A.m$ to e_d, where the body of the mock definition signals an error using blame ℓ to blame the location of the actual definition.

We could also have built e_d from the method definitions actually seen during execution, e.g., (Def) in Figure 4.7 could record what methods are defined. We think this would also be a reasonable design, but would essentially require that users have tests to drive profiling runs in order to statically analyze their code, even if they do not use features such as eval. Thus for a bit more flexibility, we build e_d based on static occurrences of definitions, but we might make dynamic method definition tracking an option in the future.

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

(If$_\hookrightarrow$)
$$\frac{e_1 \hookrightarrow_\ell e_1' \qquad e_2 \hookrightarrow_\ell e_2' \qquad e_3 \hookrightarrow_\ell e_3'}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow_\ell \mathsf{if}\ [\![e_1']\!]_\ell\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3'}$$

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

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

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

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

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

Figure 4.9: Safe evaluation rules (partial)

4.3.3 Safe Evaluation

To handle uses of dynamic features not seen in a profile, our translation in Figure 4.8 inserts calls to $\mathsf{safe\_eval}_\ell\ e$, a "safe" wrapper around eval. Figure 4.9 gives some of the reduction rules for this form. In the first rule, (SEval), we reduce $\mathsf{safe\_eval}_\ell\ e$ by evaluating e to a string s, parsing s, translating the result to e' via the $\hookrightarrow_\ell$

relation (a source-to-source transformation), and then evaluating $[\![e']\!]_\ell$, a wrapped e'. The expression $[\![e']\!]_\ell$ behaves the same as e', except if it is used type-unsafely then our semantics produces $\mathsf{blame}\ \ell$, meaning there was an error due to dynamic code from ℓ. This is contrast to type-unsafe uses of unwrapped values, which cause the semantics to go wrong (formally, reduce to *error*). In practice, we implement $[\![e']\!]_\ell$ by a method that accepts an object and modifies it to have extra run-time checking (Section 4.4).

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

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

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

are used in the method body. Similarly, we must wrap the value returned from the call so that it is checked when used later.

Notice that our semantics for $\mathsf{safe\_eval}_\ell\ e$ does not use any static type information. Instead, it performs extensive object wrapping and forbids method definitions in dynamic code. One alternative approach would be to run DRuby's type inference algorithm at run time on the string e returns. However, this might incur a substantial run-time overhead (given the space and time requirements of DRuby's type inference system), and it disallows any non-statically typed parts of the program. Another alternative would be to only keep objects wrapped until they are passed to statically typed code. At that point, we could check their type against the assumed static type, and either fail or unwrap the object and proceed. This would be similar to gradual typing [66, 65, 36]. We may explore this approach in the future, as having static types available at run time could reduce the overhead of our wrappers at the expense of additional space overhead.

4.3.4 Formal Properties

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

Theorem 3 (Translation Faithfulness) *Suppose* $\langle \emptyset, \emptyset, e \rangle \to \langle M, \mathcal{P}', v \rangle$ *and* $\mathcal{P}' \subseteq \mathcal{P}$ *and* $\mathcal{P} \vdash e \rightrightarrows e'$. *Then there exist* $M_\mathcal{P}$ *such that* $\langle \emptyset, \emptyset, e' \rangle \to \langle M_\mathcal{P}, \emptyset, v \rangle$.

In other words, if we translate an expression based on its profile (or a superset of the information in its profile), both the original and translated program produce the same result. Also, since our translation has removed all dynamic features, we will record no additional profiling information in the second execution, making the final profile $\emptyset$.

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

Corollary 4 *Suppose* $\langle \emptyset, \emptyset, (e; e_1) \rangle \to \langle M_1, \mathcal{P}_1, v_1 \rangle$. *Further, suppose that* $\langle \emptyset, \emptyset, (e; e_2) \rangle \to \langle M_2, \mathcal{P}_2, v_2 \rangle$. *Then if* $\mathcal{P}_2 \subseteq \mathcal{P}_1$ *and* $\mathcal{P}_1 \vdash (e; e_2) \rightrightarrows e'$, *then* $\langle \emptyset, \emptyset, e' \rangle \to \langle M_2', \emptyset, v_2 \rangle$.

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

Finally, the last step is to show that we can perform sound static analysis on the translated program. Appendix C gives a (mostly standard) type system for this

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

Theorem 5 (Type Soundness) *If* $\emptyset \vdash e$ *and* $\langle \emptyset, \emptyset, e \rangle \rightarrow \langle M, \mathcal{P}, r \rangle$*, then* r *is either a value or* blame ℓ*. Thus,* $r \neq$ *error.*

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

Recall that the translation from Section 4.3.2 eliminates the three dynamic features that this type system does not permit, and inserts appropriate mock definitions at the beginning of the program. Thus, if we start with an arbitrary program, gather information about its dynamic feature usage via the instrumentation in Figure 4.7, and translate it according to Figure 4.8, we can then apply sound static type checking to the resulting program, while still precisely modeling uses of eval, send, and method_missing in the original program.

4.4 Implementation

To use DRuby's profiling library, the user simply invokes DRuby with the command

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

This command executes all of the stages described in Figure 2.6. That is, it first runs *filename.rb* to discover the set of required files, and then adds instrumentation to record uses of eval, send, method_missing, and other dynamic features, to mimic the semantics in Section 4.3.1. DRuby then executes the instrumented program to gather a profile, transforms the program to eliminate dynamic constructs according to the profile (as in Section 4.3.2), and then runs its type inference on the resulting program. In the future, we expect profiling to be done separately and the results saved for later use, but for experimental purposes our current all-in-one setup is convenient.

4.4.1 Additional Dynamic Constructs

In addition to the constructs discussed in Section 4.3, DRuby also handles several other closely related dynamic features. Similarly to eval, Ruby includes instance_eval, class_eval, and module_eval methods that evaluate their string argument in the context of the method receiver (an instance, class, or module, respectively). For example, calling

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

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

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

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

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

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

Ruby includes some dynamic features DRuby does not yet support. In particular, DRuby's type system treats certain low-level methods specially, but these methods could be redefined, effectively changing the semantics of the language. For instance, if a programmer changes the Module# append_features method, they can alter the semantics of module mixins. Other special methods include Class#new,

Class#inherited, Module#method_added, and Module# included. DRuby also does not support applying dynamic constructs to per-object classes (eigen-classes) or calling dynamic features via the Method class. In addition to these features, DRuby currently does not support const_missing, which handles accesses to undefined constants, similarly to method_missing; we expect to add support for this in the future.

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

4.4.2 Implementing safe_eval

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

The druby_eval program is written using RIL, and it implements the $\hookrightarrow_\ell$ translation as shown in Figure 4.9. For example, it translates method definitions to calls to blame(), and it inserts calls to wrap() where appropriate. There are a few additional issues when implementing $\hookrightarrow_\ell$ for the full Ruby language. First, we need not wrap the guard of if, because in Ruby, if accepts any object, not just booleans. Second, in addition to forbidding method definitions, we must also disallow calls to methods that may change the class hierarchy, such as undef_method. Lastly, we add calls to wrap() around any expressions that may escape the scope of safe_eval, such

as values assigned to global variables and fields.

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

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

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

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

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

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

(b) Library results (as covered by benchmarks)

Figure 4.10: Dynamic feature profiling data from benchmarks

4.5 Profiling Effectiveness

We evaluated DRuby by running it on a suite of 13 programs downloaded from RubyForge. We included any dependencies directly used by the benchmarks, but not any optional components. Each benchmark in our suite uses at least some of the dynamic language features handled by DRuby, either in the application itself or indirectly via external libraries. All of our benchmarks included test cases, which we used to drive the profiling runs for our experiments. Finally, many projects use the rake program to run their test suites. Rake normally invokes tests in forked subprocesses, but as this would make it more difficult to gather profiling information, we modified rake to invoke tests in the same process.

4.5.1 Dynamic Feature Usage

Figure 4.10 measures usage of the dynamic constructs we saw in our profiling runs. We give separate measurements for the benchmark code (part (a)) and the library modules used by the benchmarks (part (b)). We should note that our measurements are only for features seen during our profiling runs—the library modules in particular include other uses of dynamic features, but they were in code that was not called by our benchmarks.

For each benchmark or module, we list its lines of code (computed by SLOCCount [83]) and a summary of the profiling data for its dynamic features, given in the form n/m, where n is the number of syntactic occurrences called at least once across all runs, and m is the number of unique strings used with that feature. For

Req and G/S, we only count occurrences that are used with non-constant strings. Any library modules that did not execute any dynamic features are grouped together in the row labeled *Other* in Figure 4.10(b).

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

4.5.2 Categorizing Dynamic Features

The precision of DRuby's type inference algorithm depends on how much of the full range of dynamic feature usage is observed in our profiles. To measure this, we manually categorized each syntactic occurrence from Figure 4.10 based on how "dynamically" it is used. For example, eval "x + 2" is not dynamic at all since the eval will always evaluate the same string, whereas eval ($stdin.readline) is extremely dynamic, since it could evaluate any string.

Figure 4.11 summarizes our categorization. We found that all of the dynamic

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

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

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

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

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

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

Collection A slightly more expressive use of dynamic constructs is to apply them to a small, fixed set of arguments. One common idiom (18 occurrences) we observed was to apply a dynamic construct uniformly across a fixed collection of values. For example, the code in Fig. 4.4 iterates over an Array of string literals, evaling a method definition string from each literal. Thus, while multiple strings are passed to this occurrence of eval, the same strings will be used for every execution of the program.

Additionally, any profile that executes this code will always see all possible strings for the construct.

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

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

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

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

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

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

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

n/m – n=occ, m=uniq strs

Figure 4.11: Categorization of profiled dynamic features

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

File System The next category covers those dynamic features whose use depends on the local file system. This includes most occurrences of require, e.g., the code at the top of Figure 4.1, which loads a file who name is derived from __FILE__, the current file name. Another example is the following convoluted code from *rubyforge*:

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

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

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

Open module The last category covers cases in which dynamic features are called within a library module, but the library module itself does not determine the uses. For example, the *testunit* module uses `send` to invoke test methods that the module users specify. Similarly, the *rake* module loads client-specified Ruby files containing test cases. As another example, the *ostruct* module is used to create record-like objects, as shown in Figure 4.5.

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

4.6 Type Inference

Finally, we used DRuby to perform type inference on each of the benchmarks, i.e. DRuby gathered the profiling data reported in Figure 4.10, transformed the code as outlined in Sections 4.3 and 4.4, and then applied DRuby's type inference algorithm on the resulting program.

When we first ran DRuby on our benchmarks, it produced hundreds of mes-

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

Figure 4.12: Type inference results

sages indicating potential type errors. As we began analyzing these results, we noted that most of the messages were false positives, meaning the code would actually execute type safely at run time. In fact, we found that much of the offending code is *almost* statically typable with DRuby's type system. To measure how "close" the code is to being statically typable, we manually applied a number of refactorings and added type annotations so that the programs pass DRuby's type system, modulo several actual type errors we found.

The result gives us insight into what kind of Ruby code programmers "want" to write but is not easily amenable to standard static typing. In the remainder of this section, we discuss the true type errors we found (Section 4.6.1), and what refactorings were needed for static typing (Section 4.6.2). Overall, we found that most programs could be made statically typable, though in a few cases code seems truly dynamically typed.

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

Figure 4.13: Changes needed for static typing

4.6.1 Performance and Type Errors

Figure 4.12 shows the time it took DRuby to analyze our modified benchmarks. For each benchmark, we list the total lines of code analyzed (the benchmark, its test suite, and any libraries it uses), along with the analysis time. Times were the average of three runs on an AMD Athlon 4600 processor with 4GB of memory. These results show that DRuby's analysis takes only a few minutes, and we expect the time could be improved further with more engineering effort.

Figure 4.13 lists, for each benchmark or library module used by our benchmarks, its size, the number of refactorings and annotations we applied (discussed in detail in the next section), and the number of type errors we discovered. The last row, *Other*, gives the cumulative size of the benchmarks and library modules with no changes and no type errors.

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

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

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

The type error in *archive-minitar* occurs in code that attempts to raise an

exception but refers to a constant incorrectly. Thus, instead of throwing the intended error, the program instead raises a NameError exception.

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

Finally, the *openssl* module adds code to the Integer class that calls OpenSSL :: BN :: new(self). In this call, self has type Integer, but the constructor for the OpenSSL :: BN class takes a string argument. Therefore, calling this code always triggers a run-time type error.

4.6.2 Changes for Static Typing

To enable our benchmarks and their libraries to type check (modulo the above errors), we applied 226 refactorings and added 177 type annotations. We can divide these into the following categories. For the moment, we refrain from evaluating whether these changes are reasonable to expect from the programmer, or whether they suggest possible improvements to DRuby; we discuss this issue in detail in Chapter 5.

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

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

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

Here, the case expression on line 1 tests the class of x against two possibilities. The annotations on lines 2 and 4 tell DRuby to treat x as having type Fixnum and String,

respectively, on each branch. These annotations were extremely common—we added them to 135 branches in total. We also added 9 method annotations for intersection types and 33 method annotations for higher order polymorphic types. Polymorphic type signatures can be used by DRuby given annotations, but cannot currently be inferred. DRuby adds instrumentation to check all the above annotations dynamically at run time, to ensure they are correct.

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

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

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

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

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

Block Argument Counts (24 Refactorings) In Ruby, higher-order methods can accept *code blocks* as arguments. However, the semantics of blocks is slightly different than regular methods. Recall from Chapter 3, that Ruby does not require the formal parameter list of a block to exactly match the actual arguments: formal arguments not supplied by the caller are set to nil, and extra actual arguments are ignored. DRuby, on the other hand, requires strict matching of the number of block arguments, since otherwise we could never discover mismatched argument counts for blocks. Thus we modified our benchmarks where necessary to make arguments lists match. We believe this is the right choice, because satisfying DRuby's requirement is a very minor change.

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

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

Here line 2 loads net/https if the conditional on line 1 is true. The method called on line 3 is added by a load-time eval inside of net/https. Thus, to successfully analyze this code, DRuby needs to not only analyze the source code of net/https, but it also

must have its profile to know this method exists. However, the branch on line 1 was never taken in our benchmarks, and so this require was never executed and the eval was not included in the profile.

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

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

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

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

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

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

Heterogeneous Containers (12 Refactorings) DRuby supports homogeneous containers with types such as Array<T> and Hash<K,V>. Since arrays are sometimes

used heterogeneously, DRuby also includes a special type Tuple<T_1, ..., T_n>, where the T_i are the tuple element types from left to right. Such a type is automatically coerced to Array<T_1 **or** ... **or** T_n> when one of its methods is invoked.

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

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

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

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

Other (65 Refactorings) We also needed a few other miscellaneous refactorings. In our benchmarks, there were 32 calls handled by method_missing that were never

seen in our benchmark runs. Hence DRuby reported these calls as going to undefined methods. We fixed this by manually copying the method_missing bodies for each method name they were called with, simulating our translation rules. We could also have fixed this with additional test cases to expand our profiles, so that DRuby would add these methods automatically during its transformation.

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

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

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

manually wrapping the code using the wrap() method from our safe_eval library (Section 4.4.2).

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

Discussion In chapter 3, we found that small benchmarks are statically typable. We believe that our results with DRuby suggest that even large Ruby programs are mostly statically typable—on balance, most of our refactorings and type annotations indicate current limitations of DRuby, and a few more suggest places where Ruby programmers could easily change their code to be typable (e.g., making argument counts for blocks consistent). Given the extreme flexibility of Ruby, we think this result is very encouraging, and it suggests that static typing could very well succeed in practice. Furthermore, these benchmarks were not written with static typing in mind and we believe that DRuby could be most useful to programmers while they are developing their code, so that potential errors can be caught early in the development life cycle.

4.7 Threats to Validity

There are several potential threats to the validity of our results. Figures 4.10(a) and (b) only include dynamic constructs that were observed by our benchmark runs. As we mentioned earlier, there are also other dynamic constructs that are present in the code (particularly the library modules) but were not called via our test suites. However, additional profiling to try to exhibit these features would only bolster our claim that dynamic features are important to model. A more important consequence is that our categorization in Figure 4.11 may not generalize. It is possible that if we examined more constructs, we would find other categories or perhaps some features used in very dynamic ways. However, this would not affect our other results, and we believe we looked at enough occurrences (66 total) to gather useful information.

In Ruby, it is possible for code to "monkey-patch" arbitrary classes, changing their behavior. Monkey patching could invalidate our categorization from Section 4.5.2, e.g., by exposing a dynamic feature whose uses were previously bounded within a class. However, this would only affect our categorization and not DRuby, which can still easily profile and analyze the full, monkey-patched execution.

Similarly, Ruby's low-level object API could allow a programmer to subvert our analysis, as discussed at the end of Section 4.6.2. Because we cannot verify these unsafe features, they could potentially disable our run-time instrumentation, causing a Ruby script to fail. However, we hope that programmers who use unsafe features will treat them with appropriate caution.

4.8 Related Work

The key contribution of DRuby is our sound handling of highly dynamic language constructs. The previous chapter avoided these features by sticking to small examples, using programmer annotations for library APIs, and eliminating dynamic constructs with manual transformation. However, as we saw in Section 4.5, highly dynamic features are pervasive throughout Ruby, and so this approach is ultimately untenable. Kristensen [44] has also developed a type inference system for Ruby based on the cartesian product algorithm. This system does not handle any of Ruby's dynamic features, making it unsound in the presence of these constructs.

In addition to DRuby, researchers have proposed a number of other type systems for dynamic languages including Scheme [20, 77], Smalltalk [33, 70, 87], Javascript [73, 35, 9], and Python [62, 10, 17], though these Python type systems are aimed at performance optimization rather than at the user level. To our knowledge, none of these systems handles `send`, `eval`, or similar dynamic features.

One exception is RPython [8], a system that inspired our work on DRuby. RPython translates Python programs to type safe back-ends such as the JVM. In RPython, programs may include an initial bootstrapping phase that uses arbitrary language features, including highly dynamic ones. RPython executes the bootstrapping phase using the standard Python interpreter, and then produces a type safe output program based on the interpreter state. The key differences between RPython and DRuby are that DRuby supports dynamic feature use at arbitrary execution points; that we include a formalization and proof of correctness; that

we provide some information about profile coverage with test runs; and, perhaps foremost, that DRuby operates on Ruby rather than Python.

Another approach to typing languages with dynamic features is to use the type Dynamic [2]. Extensions of this idea include quasi-static typing [71], gradual type systems [66, 65, 36], and hybrid types [34]. However, we believe these approaches cannot handle cases where dynamic code might have side effects that interact with (what we would like to be) statically typed code. For example, recall the code from Figure 4.4, which uses eval to define methods. Since these definitions are available everywhere, they can potentially influence any part of the program, and it is unclear how to allow some static and some dynamic typing in this context. In contrast, DRuby explicitly supports constructs that would *look* dynamic to a standard type system, but *act* essentially statically, because they have only a few dynamic behaviors that can be seen with profiling; for code that is truly dynamic, DRuby reverts to full dynamic checking.

Several researchers have proposed using purely static approaches to eliminating dynamic language constructs. Livshits et al. [46] use a static points-to analysis to resolve reflective method calls in Java by tracking string values. Christensen et al. [21] propose a general string analysis they use to resolve reflection and check the syntax of SQL queries, among other applications. Gould et al. [32] also propose a static string analysis to check database queries, and several proposed systems use partial evaluation to resolve reflection and other dynamic constructs [15, 72]. The main disadvantage of all of these approaches is that they rely purely on static analysis. Indeed, Sawin and Rountev [63] observe that pure static analysis of strings

is unable to resolve many dynamic class loading sites in Java. They propose solving this problem using a semi-static analysis, where partial information is gathered dynamically and then static analysis computes the rest. In DRuby, we opted to use a pure dynamic analysis to track highly dynamic features, to keep DRuby as simple and predictable as possible.

Chugh et al. [22] present a hybrid approach to information flow in Javascript that computes as much of the flow graph as possible statically, and performs only residual checks at run time when new code becomes available. In Ruby, we found that the effects of dynamic features must be available during static analysis, to ensure that all defined methods are known to the type checker. Our runtime instrumentation for blame tracking is similar to a proposed system for tracking NULL values in C [13]. One difference is that we must check for and allow type-correct methods at runtime, whereas NULL supports no operations.

Finally, there is an extensive body of work on performing static analysis for optimization of Java. A major challenge is handling both dynamic class loading and reflection. Jax [75] uses programmer specifications to ensure safe modeling of reflective calls. Sreedhar et al. [68] describe a technique for ahead-of-time optimization of parts of a Java program that are guaranteed unaffected by dynamic class loading. Pechtchanski and Sarkar [55] present a Java optimization system that reanalyzes code on seeing any dynamic events that would invalidate prior analysis. Hirzel et al. [37] develop an online pointer analysis that tracks reflective method calls and can analyze classes as they are dynamically loaded. All of these systems are concerned with optimizing a program, whereas in contrast, DRuby extracts run-time profiling

information to guide compile-time (user-level) type inference.

Chapter 5

Future Work

DRuby has already proven useful by detecting several previously unknown type errors in our benchmarks. However, DRuby's type system and its profiling library could be benefit from further improvements. We now sketch a few ways in which DRuby could be improved in the future.

5.1 Type System Improvements

Annotated Expressions DRuby uses inference to discover the types of a user's code automatically. This frees the programmer from having to write down the types needed to type his program. However, when the inference algorithm discovers a type error, the resulting error message can be opaque. One reason for this is that a type error is produced when a collection of typing constraints is unsatisfiable, and DRuby has no idea which constraint was at fault. For example, consider the code

```
1 class A
2   def bar() ... end
3 end
4 class B
5   def baz() ... end
6 end
7 def c(x) x.baz() end
8 c(A.new)
```

When run on this program, DRuby will emit an error message saying that class A does not have a baz method. One possibility is that the programmer mistakenly passed an instance of class A to method c method, where he should have passed an instance of B. Or, another possibility is that the call x.baz on line 7 is a typo, and it should actually be x.bar. Other implicitly typed languages such as OCaml solve this problem by allowing individual expressions to be annotated with a type expression, thereby reducing the set of constraints that influence an error. Currently, DRuby only allows annotations on classes and method signatures. Allowing annotations on arbitrary expressions would help with user comprehension of error messages.

Modular Analysis DRuby currently must analyze an entire Ruby program at once, taking up to a few minutes to complete its analysis on a single benchmark. While we believe this time could be reduced with some engineering effort, DRuby would ultimately benefit most from analyzing parts of a program in isolation (e.g., analyzing the standard library separately from a client program).

One solution would be to use *interface* files that can be analyzed in isolation. Interface files would be similar to Ruby files, except instead of containing source code, they would only contain type signatures for classes and methods. This would allow DRuby to analyze the bodies of a methods only once (when the interface file is first verified). For example, consider the following Ruby implementation and interface files:

```
Implementation (A.rb):
  class A
    def foo(data)
      max = data.map {|s| s.to_i }.max
```

```
      sqr = max * max
      return(sqr - 2 / 13)
    end
  end

Interface (A.rbi):
  class A
    foo : Array<String> -> Fixnum
  end
```

Analyzing the body of the foo method requires DRuby to generate constraints for 6 different method calls (map, to_i, max, $*$, $-$, and /). However, once the code in A.rb is found to match the interface file A.rbi, the constraints on these calls do not need to be regenerated every time a program requires "A.rb". Instead, DRuby would load "A.rbi" in place of "A.rb" and would only need to verify that code uses the foo method at the right type in other files. By reducing the number of constraints generated by DRuby, its performance would certainly improve.

Statically Modeling Dynamic Type Tests Section 4.6.2 demonstrated that dynamic type tests are clearly important to Ruby programmers but are not modeled by DRuby. Modeling type tests statically poses two challenges. First, we need to be able to detect when a type test occurs in the source code. Unfortunately, there are a myriad of ways to perform such a test in Ruby. For example, each of the following lines checks to see if x is an instance of Fixnum (except for the last 2 lines, which merely check that it responds to the + method):

```
1   x.class == Fixnum
2   Fixnum === x
3   case x when Fixnum ...
4   x.is_a? Fixnum
5   x.responds_to? :+
6   x.methods.include? ''+''
```

Second, the type system must be able to reason about the paths within a method body to generate the proper typing constraints. For example, consider the code:

```
1   ##% foo: Fixnum -> Fixnum
2   ##% foo: Boolean -> Boolean
3   def foo(x)
4    y = x.clone
5    case y
6    when Boolean
7      return true
8    else
9      return x + 1
10   end
11  end
```

Here, we define a method foo with an intersection type. One way to statically type this method would be to first assume that the parameter x has type Fixnum and then verify the return type of the method also has type Fixnum (and then repeat this for Boolean). However, this method would still be difficult to type check for two reasons. First, even if DRuby were to assume x has type Fixnum, it would need to ignore the return statement on line 7 that returns the literal true. Second, if DRuby were to assume x has type Boolean, it must know to ignore the else branch on line 9 since x + 1 would otherwise result in a type error.

Occurrence Typing [77], previously proposed for Scheme, is one possible solution we plan to explore. Although given the multitude of ways to test the dynamic type of a value in Ruby, some care must be taken to strike the right balance between supporting common uses and producing an easy-to-use system.

Detecting Nil Errors Like Java, DRuby treats nil as a subtype of any class. Therefore, errors related to nil are not detected by the type system. However, unlike

Java, nil is an object in Ruby (and an instance of the NilClass class) and does respond to some methods. Thus, if DRuby were able to model type tests like those mentioned above, it would be able to detect when a programmer was performing a type test on nil. If this analysis still proved to be too coarse, then another alternative would be to extend DRuby's runtime contracts to check for nil values dynamically so that DRuby would not falsely reject too many programs based on subtle usages of nil.

5.2 Profiling Improvements

Some of the refactorings we performed in Section 4.6.2 may be difficult to address with changes to DRuby's type inference algorithm, but could be handled with improvements to our profiling technique. For example, currently DRuby performs profiling, transformation, and type inference in one run (Section 4.4). If we could combine profiles from multiple runs, we could run additional tests to improve code coverage. For example, instead of hoisting require to the top-level of a file, a better solution may be to use additional test suites (such as those provided by a library maintainer), or for libraries to ship a profile database that could be used by library clients.

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

Lastly, our refactorings for code that uses strings to select different classes (e.g., "r" for TarInput, "w" for TarOutput) were similar to the automated transformations that DRuby performs. Therefore, we might be able to support this idiom by extending DRuby to profile user-defined constructs similarly to Ruby's reflective constructs such as send.

Chapter 6

Conclusions

In this dissertation, we have presented DRuby, a tool that blends static and dynamic typing for Ruby. DRuby allows programmers to select the amount of static checking desired, allowing for dynamic scripts to be incrementally hardened into robust code bases. In order to maintain the "feel" of a dynamic language in the presence of static types, DRuby uses three techniques:

First, DRuby uses a type inference algorithm to automatically infer static types for existing Ruby code, minimizing the burden on the programmer to write down types in their code. Common Ruby idioms are modeled precisely, e.g., using flow sensitive types for local variables and heterogeneous types for tuples. We have proven DRuby's static type system is sound, ensuring that a well-typed Ruby program will never go wrong at runtime.

Second, DRuby allows programmers to add annotations to their code when necessary or desired. DRuby can then check these annotations either statically using its type checking algorithm or dynamically by using runtime contracts with blame tracking. The latter allows programmers to write dynamic code without affecting the safety guarantees of statically checked code. DRuby's type annotation

syntax is similar to existing informal documentation format used by Ruby and so should be familiar to existing Ruby programmers.

Third, programmers may use dynamic features such as eval to write expressive code. Using the program's test suite, DRuby can profile these constructs discovering their effects. Again, the programmer is given flexibility here: the more dynamic features covered in their test suite, the more static checking is achieved. These profiles then guide a transformation phase that replaces dynamic constructs with statically analyzable code that approximate its behavior. We have proven that our transformation is faithful and typing checking the resulting code is sound for a small Ruby-like calculus with dynamic features. We believe that using profiles to enhance static analysis is a promising technique for analyzing programs written in highly dynamic scripting languages in general.

We have also presented the implementation details of DRuby. DRuby is built on top of RIL, which provides a representation of Ruby source code that makes it easy to develop source code analysis and transformation tools. We believe RIL minimizes redundant work and reduces the chances of mishandling certain Ruby features, making RIL an effective and useful framework for working with Ruby source code. We hope that RIL's features will enable others to more easily build analysis tools for Ruby, and that our design will inspire the creation of similar frameworks for other dynamic languages.

Finally, we evaluated the effectiveness of DRuby on a range of benchmarks. We found that our static type system worked well and discovered several latent type errors as well as a number of questionable coding practices. Additionally,the use of

dynamic features is pervasive throughout our benchmarks, but that most uses of these features are essentially static, and hence can be profiled. In conclusion, this dissertation has shown that DRuby effectively integrates static typing into Ruby without losing the feel of a dynamic language.

Appendix A

RIL Example Source Code

```
open Cfg
open Cfg_printer
open Visitor
open Utils

let method_formal_name = function
  | `Formal_meth_id var
  | `Formal_amp var
  | `Formal_star var
  | `Formal_default(var, _) -> var

module NilAnalysis = struct

  type t = fact StrMap.t
  and fact = MaybeNil | NonNil

  let top = StrMap.empty
  let eq t1 t2 = StrMap.compare Pervasives.compare t1 t2 = 0
  let fact_to_s = function MaybeNil -> "MaybeNil" | NonNil -> "NonNil"
  let to_string t = strmap_to_string fact_to_s t

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

  let update s fact map = StrMap.add s fact map

  let meet_fact s v map =
    let fact =
      try meet_fact (StrMap.find s map) v
      with Not_found -> v
    in StrMap.add s fact map
```

```
  let meet lst =
    List.fold_left (fun acc map -> StrMap.fold meet_fact map acc)
      StrMap.empty lst

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

  let transfer map stmt = match stmt.snode with
    | Assign(lhs, #literal) ->
        update_lhs NonNil map lhs
    | Assign(lhs, `ID_Var(`Var_Local, rvar)) ->
        update_lhs (StrMap.find rvar map) map lhs

    | MethodCall(lhs_o, {mc_target=Some (`ID_Var(`Var_Local,targ))}) ->
        let map = match lhs_o with
          | None -> map
          | Some lhs -> update_lhs MaybeNil map lhs
        in
          update targ NonNil map

    | Class(Some lhs,_,_) | Module(Some lhs,_,_)
    | MethodCall(Some lhs,_) | Yield(Some lhs,_)
    | Assign(lhs, _) -> update_lhs MaybeNil map lhs

    | _ -> map

  let init_formals args fact =
    List.fold_left
      (fun acc param ->
         update (method_formal_name param) fact acc
      ) top args

end

module NilDataFlow = Dataflow.Forwards(NilAnalysis)

open Cfg_refactor
open Cfg_printer.CodePrinter

let transform targ node =
  freparse ~"env:node. lexical_locals "unless %a.nil? then %a end"
    format_expr targ format_stmt node
```

```
class safeNil inf = object( self )
  inherit default_visitor as super
  val facts = inf

  method visit_stmt node = match node.snode with
    | Method(mname,args,body) ->
        let in', out' = NilDataFlow.fixpoint body NilAnalysis.top in
        let me = {<facts = in'>} in
        let body' = visit_stmt (me:> cfg_visitor) body in
          ChangeTo (update_stmt node (Method(mname,args,body')))

    | MethodCall(_, {mc_target=(Some `ID_Self| None)}) -> SkipChildren
    | MethodCall(_, {mc_target=Some (`ID_Var(`Var_Local,var) as targ)}) ->
        begin try let map = Hashtbl.find facts node in
          begin try match StrMap.find var map with
            | NilAnalysis.MaybeNil -> ChangeTo (transform targ node)
            | NilAnalysis.NonNil -> SkipChildren
          with Not_found -> ChangeTo (transform targ node)
          end
        with Not_found -> assert false
        end

    | MethodCall(_, {mc_target=Some (#expr as targ)}) ->
        ChangeTo (transform targ node)
    | _ -> super#visit_stmt node
end

open Dynamic

module NilProfile : DynamicAnalysis = struct
  module Domain = Yaml.YString
  module CoDomain = Yaml.YBool

  let name = "dynnil"

  let really_nonnil lookup mname pos =
    let uses = lookup mname pos in
      if uses = [] then false
      else not (List.mem false uses)

  class dyn_visitor lookup ifacts =
  object( self )
    inherit (safeNil ifacts) as super

    method visit_stmt node = match node.snode with
      | Method(defname,args,body) ->
          let mname = format_to_string format_def_name defname in
```

```
          let init_facts =
            if really_nonnil lookup mname body.pos
            then NilAnalysis.init_formals args NilAnalysis.NonNil
            else NilAnalysis.top
          in
          let in', _ = NilDataFlow.fixpoint body init_facts in
          let me = {<facts = in'>} in
          let body' = visit_stmt (me:>cfg_visitor) body in
            ChangeTo (update_stmt node (Method(defname,args,body')))

    | _ -> super#visit_stmt node
end

let transform_cfg lookup stmt =
  compute_cfg stmt;
  compute_cfg_locals stmt;
  let i,_ = NilDataFlow.fixpoint stmt NilAnalysis.top in
    visit_stmt (new dyn_visitor lookup i :> cfg_visitor) stmt

let instrument_ast ast = ast

let get_pos pos =
  pos.Lexing.pos_fname, pos.Lexing.pos_lnum

let format_param ppf p =
  Format.pp_print_string ppf (method_formal_name p)

open Cfg.Abbr

let instrument mname args body pos =
  let file, line = get_pos pos in
  let code = freparse ~env:body. lexical_locals
    "DRuby::Profile::Dynnil.watch('%s',%d,self,'%a',[%a])"
     file line format_def_name mname
    (format_comma_list format_param) args
  in
  let body' = seq [code;body] body.pos in
    meth mname args body' pos

let should_instrument stmt = true

class instrument_visitor = object(self)
  inherit default_visitor as super
  method visit_stmt stmt = match stmt.snode with
    | Method(mname,args,body) ->
      if should_instrument stmt
      then ChangeTo (instrument mname args body stmt.pos)
```

```
        else SkipChildren
      | _ -> super#visit_stmt stmt
  end

  let instrument_cfg stmt =
    compute_cfg stmt;
    compute_cfg_locals stmt;
    visit_stmt (new instrument_visitor ) stmt

  let transform_ast ym ast = ast
end

let dyn_main fname =
  let module Dyn = Make(Singleton(NilProfile)) in
  let loader = File_loader.create File_loader.EmptyCfg ["../lib"] in
    print_stmt stdout (Dyn.run loader fname)

let main fname =
  let loader = File_loader.create File_loader.EmptyCfg [] in
  let s = File_loader. load_file loader fname in
  let () = compute_cfg s in
  let () = compute_cfg_locals s in
  let ifacts, _ = NilDataFlow.fixpoint s NilAnalysis.top in
  let s' = visit_stmt (new safeNil ifacts :> cfg_visitor ) s in
    print_stmt stdout s'

let _ =
  if (Array.length Sys.argv) != 2
  then begin
    Printf.eprintf "Usage: print_cfg <ruby_file> \n";
    exit 1
  end;
  let fname = Sys.argv.(1) in
    dyn_main fname;
    (*main fname;*)
    ()
```

Appendix B

Proofs for MiniRuby

B.1 Static Semantics

The first step to proving soundness is to augment our typing rules to include the runtime values used by our dynamic semantics:

$$
\begin{array}{rcl}
e & ::= & \ldots \mid v \\
v & ::= & l \mid [A; F_v; M_v] \mid [v, \ldots, v] \\
F_v & ::= & \emptyset \mid @x = v, F_v \\
M_v & ::= & \emptyset \mid \mathsf{def}\ m(x_1, \ldots, x_n) : \eta = e, M_v \\
\Omega & ::= & \ldots \mid \Omega, l : \tau \\
r & ::= & \mathit{error} \mid v
\end{array}
$$

Values v include locations l (which point to objects), object literals $[A; F_v; M_v]$, which include their class name A, as well as field and method sets. Although only fields may be updated in our calculus, we use a single form for objects that is updated during program evaluation. Thus each instance of an object includes its own set of (immutable) method definitions. Values also include a literal form for

$\boxed{\Omega; \Gamma; \Delta \vdash e : \tau; \Delta'}$

$$\frac{l \in dom(\Omega) \qquad \Omega(l) = [F; M]}{\Omega; \Gamma; \Delta \vdash l : [F; M]; \Delta} \ (\text{Loc}_\tau)$$

$$\frac{\begin{array}{c}\Omega(A) = \sigma \qquad i \in 1..n \\ \Gamma \vdash \sigma <: [F; M]\ \mathsf{class} \\ A : [F; M]\ \mathsf{class}\ ; \Omega; \Gamma; \Delta \vdash d_i\end{array}}{\Omega; \Gamma; \Delta \vdash [A; d_1, \ldots, d_n] : [F; M]\ \mathsf{class}\ ; \Delta} \ (\text{Class-Lit}_\tau)$$

Figure B.1: Type Checking Rules for Values

tuples $[v, \ldots, v]$. Our type checking judgments for these new forms are shown in Figure B.1.

(Loc_τ) looks up locations in the current store, which also must be objects. Similarly, class literals are typed using (Class-Lit_τ) by looking up the class name in Ω. We then check the body of the class using the (Field Decl_τ) and (Method Decl_τ).

B.2 Dynamic Semantics

We now give our evaluation rules for MiniRuby. Our operational semantics will use "big-step" rules for evaluation [43]. We use the additional stores S and V: S maps either class names A or heap locations l to values, and V maps local variables x to values. We define a *configuration* to be a tuple $\langle S, V, e\rangle$ meaning that expression e is being evaluated in the context of stores S and V.

Our dynamic semantics for expressions are shown in Figures B.2 and B.3. We omit the error states and implicitly assume that any time none of our rules apply then the expression reduces to *error*.

$$\frac{x \in dom(V)}{\langle S, V, x\rangle \to \langle S, V, V(x)\rangle}\ (\textsc{Var}_{\to})$$

$$\frac{\langle S, V, e\rangle \to \langle S', V', v\rangle \quad V'' = V'[x \mapsto v]}{\langle S, V, x = e\rangle \to \langle S', V'', v\rangle}\ (\textsc{Assign Local}_{\to})$$

$$\frac{A \in dom(S)}{\langle S, V, A\rangle \to \langle S, V, S(A)\rangle}\ (\textsc{Class}_{\to})$$

$$\frac{\mathsf{self} \in dom(V) \qquad l = V(\mathsf{self})}{\langle S, V, \mathsf{self}\rangle \to \langle S, V, l\rangle}\ (\textsc{Self}_{\to})$$

$$\frac{\langle S_i, V_i, e_i\rangle \to \langle S_{i+1}, V_{i+1}, v_i\rangle \quad i \in 1..n}{\langle S_1, V_1, [e_1, \ldots, e_n]\rangle \to \langle S_{i+1}, V_{i+1}, [v_1, \ldots, v_n]\rangle}\ (\textsc{Tuple}_{\to})$$

$$\frac{\langle S, V, e\rangle \to \langle S', V', [v_1, \ldots, v_n]\rangle \quad V'' = V'[x_i \mapsto v_i] \qquad i \in 1..n}{\langle S, V, x_1, \ldots, x_n = e\rangle \to \langle S', V'', [v_1, \ldots, v_n]\rangle}\ (\textsc{Assign Tuple}_{\to})$$

$$\frac{\langle S, V, \mathsf{self}\rangle \to \langle S, V, l\rangle \quad S(l) = [A; @x = v, F_v; M_v]}{\langle S, V, @x\rangle \to \langle S, V, v\rangle}\ (\textsc{Field}_{\to})$$

$$\frac{\langle S, V, e\rangle \to \langle S', V', v\rangle \quad \langle S', V', \mathsf{self}\rangle \to \langle S', V', l\rangle \quad S'(l) = [A; @x = v', F_v; M_v] \quad S'' = S'[l \mapsto [A; @x = v, F_v; M_v]]}{\langle S, V, @x = e\rangle \to \langle S'', V', v\rangle}\ (\textsc{Assign Field}_{\to})$$

$$\frac{\langle S, V, e\rangle \to \langle S', V', v\rangle \qquad v = [A; F_v; M_v] \quad S'' = S'[l \mapsto v] \qquad l \notin dom(S')}{\langle S, V, \mathsf{new}\ e\rangle \to \langle S'', V', l\rangle}\ (\textsc{New}_{\to})$$

$$\frac{\langle S, V, e_1\rangle \to \langle S', V', v_1\rangle \quad \langle S', V', e_2\rangle \to \langle S'', V'', v_2\rangle}{\langle S, V, e_1; e_2\rangle \to \langle S'', V'', v_2\rangle}\ (\textsc{Seq}_{\to})$$

Figure B.2: Big-step Operational Semantics for Expressions (1/2)

(TypeCase Match$_\rightarrow$)

$$\frac{\begin{array}{c}\langle S_1, V_1, e_1\rangle \rightarrow \langle S_2, V_2, l\rangle \\ S_2(l) = [A; F_v; M_v] \qquad V_3 = V_2[x \mapsto l] \\ \langle S_2, V_3, e_2\rangle \rightarrow \langle S_3, V_4, v_2\rangle \\ V' = V_4|_{dom(V_2)}\end{array}}{\begin{array}{c}\langle S_1, V_1, \mathsf{typecase}\ e_1\ \mathsf{when}\ (x : A)\ e_2\ \mathsf{else}\ e_3\rangle \\ \rightarrow \langle S_3, V', v_2\rangle\end{array}}$$

(TypeCase Else$_\rightarrow$)

$$\frac{\begin{array}{c}\langle S_1, V_1, e_1\rangle \rightarrow \langle S_2, V_2, l\rangle \\ S_2(l) \neq [A; F_v; M_v] \qquad \langle S_2, V_2, e_3\rangle \rightarrow \langle S_3, V_3, v_3\rangle \\ V' = V_3|_{dom(V_2)}\end{array}}{\begin{array}{c}\langle S_1, V_1, \mathsf{typecase}\ e_1\ \mathsf{when}\ (x : A)\ e_2\ \mathsf{else}\ e_3\rangle \\ \rightarrow \langle S_3, V', v_3\rangle\end{array}}$$

(Call$_\rightarrow$)

$$\frac{\begin{array}{c}\langle S_i, V_i, e_i\rangle \rightarrow \langle S_{i+1}, V_{i+1}, v_i\rangle \qquad i \in 1..n \\ \langle S_{i+1}, V_{i+1}, e_0\rangle \rightarrow \langle S', V', l\rangle \\ S'(l) = [A; F_v; \mathsf{def}\ m(x_1, \ldots, x_n) : \eta = e, M_v] \\ V_m = [\mathsf{self} \mapsto l, x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \\ \langle S', V_m, e\rangle \rightarrow \langle S'', V'_m, v\rangle\end{array}}{\langle S_1, V_1, e_0.m(e_1, \ldots, e_n)\rangle \rightarrow \langle S'', V', v\rangle}$$

Figure B.3: Big-step Operational Semantics for Expressions(2/2)

(VAR$_{\rightarrow}$) looks up the variable x in V and reduces to its value if present. (ASSIGN LOCAL$_{\rightarrow}$) updates the variable in V and then reduces to the value of the right hand side. (CLASS$_{\rightarrow}$) and (SELF$_{\rightarrow}$) look up their respective identifiers in S and V.

(TUPLE$_{\rightarrow}$) reduces a tuple of expressions into a tuple of values evaluating left to right. Similarly, (ASSIGN TUPLE$_{\rightarrow}$) stores each x_i in V if the tuple is the correct width.

(FIELD$_{\rightarrow}$) first reduces self to a location, and then looks up l in the heap store S, extracting the value stored at $@x$. Analogously, (ASSIGN FIELD$_{\rightarrow}$) updates the store S to point to a new object where the value of $@x$ has been updated.

(NEW$_{\rightarrow}$) allocates a new object by storing a class object at a fresh location l in the heap store S.

(SEQ$_{\rightarrow}$) is straight forward.

(TYPECASE MATCH$_{\rightarrow}$) applies when e_1 is an instance of class A. In this case, x is added to V_2 with the value that the guard e_1 reduced to v_1, and then evaluates e_2 with that store. We then restrict the final variable store to include only variables defined before the typecase . (TYPECASE ELSE$_{\rightarrow}$) applies when e_1 is not an instance of class A. In this case, we simply reduce e_3 and make a similar restriction on V_3.

Finally, (CALL$_{\rightarrow}$) reduces method calls. First we evaluate each of the n arguments from left to right. Then, we reduce the receiver to a location l and look up its object literal in S'. We then create a new local environment V_m and evaluate the body of the method with the corresponding formal parameters set to their values. The resulting local environment is discarded and we reduce to v.

$$\frac{\begin{array}{c}d = (@x_i = e_i); (\mathsf{def}\ m_j(x_1, \ldots, x_{j_n}) : \eta_j = e'_j) \qquad i \in 1..p, j \in 1..q \\ \langle S_i, \emptyset, e_i\rangle \to \langle S_{i+1}, V_i, v_i\rangle \\ v = [A; @x_i = v_i; \mathsf{def}\ m_j(x_1, \ldots, x_{j_n}) : \eta_j = e'_j] \\ S' = S_{p+1}[A \mapsto v]\end{array}}{\langle S_1, V, \mathsf{class}\ A : \sigma = d\rangle \to \langle S', V, v\rangle}\ (\textsc{Class Def}_{\to})$$

$$\frac{\begin{array}{c}S_1 = \emptyset \qquad i \in 1..n \\ \langle S_i, \emptyset, c_i\rangle \to \langle S_{i+1}, V_i, v_i\rangle \\ \langle S_{n+1}, \emptyset, e\rangle \to \langle S, V, v\rangle\end{array}}{\langle \emptyset, \emptyset, c_1 \cdots c_n; e\rangle \to \langle S, V, v\rangle}\ (\textsc{Program}_{\to})$$

Figure B.4: Big-step Operational Semantics for Definitions

Our rules for definitions and programs are shown in Figure B.4. Class definitions are handled by rule (Class Def$_{\to}$). A class definition evaluates each of its fields accumulating a single heap store S_i, but using a fresh local store for each definition. Note that methods need no evaluation at this point. We then construct a class literal v containing each field and method value. Finally, we add the class A to the store S'.

Lastly, we evaluate programs by evaluating each class definition then evaluating the "main" expression in its own local scope.

B.3 Soundness

We proceed to show soundness in the style of Wright and Felleisen [86].

Definition 6 (Compatibility) *The environments Ω, Γ, and Δ are said to be compatible with the stores S and V (written $\langle \Omega, \Gamma, \Delta\rangle \sim \langle S, V\rangle$) if:*

- $dom(\Omega) = dom(S)$

- $dom(\Delta) = dom(V)$
- $\forall l \in dom(S)$, *there exists* τ *such that* $\Omega; \Gamma; \Delta \vdash l : \Omega(l); \Delta$ *and* $\Omega; \Gamma; \Delta \vdash S(l) : \Omega(l)$ class $; \Delta$
- $\forall A \in dom(S)$, *there exists* τ *such that* $\Omega; \Gamma; \Delta \vdash A : \Omega(A); \Delta$ *and* $\Omega(A) = \tau$ class .
- $\forall x \in dom(V)$, $\Omega; \Gamma; \Delta \vdash V(x) : \Delta(x); \Delta$.

Definition 7 (Well-Formedness) *The stores* S, V *and environments* Ω *are said to be well-formed if*

- $\forall v \in dom(S)$, $S(v) = [A; F_v; M_v]$ *for some* A, F_v, *and* M_v
- $V(\mathsf{self}) = l$ *for some* l
- $\forall A \in dom(\Omega)$, *if* ($\mathsf{def}$ $m(x_1, \ldots, x_n) : \eta = e) \in A$, *then* $A : \Omega(A); \Omega; \Gamma; \emptyset \vdash$ def $m(x_1, \ldots, x_n) : \eta = e$

Definition 8 (Restriction) *Let* $\mathcal{M}$ *be any map from* $\mathcal{A} \to \mathcal{B}$ *(such as* Δ, S, *or* V*).*

Define $\mathcal{M}|_Y = \{a \mapsto b \mid a \in dom(\mathcal{M}) \wedge a \in Y \wedge b = \mathcal{M}(a)\}$ *where* $a \mapsto b$ *is a mapping from* a *to* b.

Definition 9 (Extension) $\mathcal{M}'$ *is an* extension *of* $\mathcal{M}$, *written* $\mathcal{M} \Rightarrow \mathcal{M}'$ *if* $\mathcal{M} = \mathcal{M}'|_{dom(\mathcal{M})}$

Lemma 10 (Variable Weakening) *If* $\langle \Omega, \Gamma, \Delta \rangle \sim \langle S, V \rangle$, *then for any set* X, *and any environment* Δ', $\langle \Omega, \Gamma, \Delta|_X \uplus \Delta'|_X \rangle \sim \langle S, V|_X \rangle$

Proof: Assume $\langle \Omega, \Gamma, \Delta \rangle \sim \langle S, V \rangle$, and let X be any set and Δ' be any environment. Let $V' = V_X$ and let $\Delta'' = \Delta|_X \uplus \Delta'|_X$. Then we must show $\forall x \in V'$, $\Omega; \Gamma; \Delta'' \vdash V(x) : \Delta''(x); \Delta''$.

Without loss of generality, let $x \in V'$. By Definition 8, $x \in V$ and therefore $\Omega; \Gamma; \Delta \vdash V(x) : \Delta(x); \Delta$ by assumption. By examining Definition 1, either $\Delta''(x) = \Delta(x) = \Delta'(x)$, or $\Delta''(x) = \Delta(x) \cup \Delta'(x)$. In the first case, $\Omega; \Gamma; \Delta'' \vdash V'(x) : \Delta''(x); \Delta''$ holds trivially.

Assume $\Delta''(x) = \Delta(x) \cup \Delta'(x)$. Then $\Omega; \Gamma; \Delta \vdash V(x) : \Delta(x) \cup \Delta'(x); \Delta$ by ($\textsc{Subsumption}_\tau$) and ($\textsc{UnionR}_{<:}$). That is, $\Omega; \Gamma; \Delta \vdash V(x) : \Delta''(x); \Delta$ Then by ($\textsc{var}_\tau$), $\Omega; \Gamma; \Delta'' \vdash V(x) : \Delta''(x); \Delta''$ □

Lemma 11 (Inversion) *If Ω is well-formed, then:*

- *If $\Omega; \Gamma; \Delta \vdash v : [F; M]; \Delta$ then $v = l$.*
- *If $\Omega; \Gamma; \Delta \vdash v : [F; M]$ class $; \Delta$ then $v = [A; F_v; M_v]$ with $\Omega(A) = [F; M]$.*
- *If $\Omega; \Gamma; \Delta \vdash v : (\tau_1 \times \cdots \times \tau_n); \Delta$ then $v = [v_1, \ldots, v_n]$ with $\Omega; \Gamma; \Delta \vdash v_i : \tau_i; \Delta$.*

Proof: By inspecting the type rules, there is a one to one correspondence between type rules and values which satisfies this equality. □

Lemma 12 *If $\Omega; \Gamma; \Delta \vdash r : \tau; \Delta$ then r is not error.*

Proof: No type rules are given for *error*, thus if a type rule applies, $r \neq error$. □

Lemma 13 (Extension Transitivity) *If $\mathcal{M} \Rightarrow \mathcal{M}'$ and $\mathcal{M}' \Rightarrow \mathcal{M}''$ then $\mathcal{M} \Rightarrow \mathcal{M}''$.*

Lemma 14 (Extension Weakening) *If* $\Omega; \Gamma; \Delta \vdash e : \tau; \Delta'$ *and* $\Omega \Rightarrow \Omega'$ *then* $\Omega'; \Gamma; \Delta \vdash e : \tau; \Delta'$.

Lemma 15 (Value Strengthening) *If* $\Omega; \Gamma; \Delta \vdash v : \tau; \Delta$, *then for any* Δ', *we have* $\Omega; \Gamma; \Delta' \vdash v : \tau; \Delta'$.

Proof: By inspecting the type rules, none of the rules for values make use of Δ.

□

Lemma 16 (Field Select) *If* $\Omega; \Gamma; \Delta \vdash [A; @x = v, F_v; M_v] : [@x : \tau, F; M]$ class $; \Delta$ *and* $A \in dom(\Omega)$, *then* $\Omega; \Gamma; \Delta \vdash v : \tau; \Delta$.

Lemma 17 (Field Update) *If* $\Omega; \Gamma; \Delta \vdash [A; @x = v, F_v; M_v] : [@x : \tau, F; M]$ class $; \Delta$ *and* $\Omega; \Gamma; \Delta \vdash v' : \tau; \Delta$ *then* $\Omega; \Gamma; \Delta \vdash [A; @x = v', F_v; M_v] : [@x = \tau, F; M]$ class $; \Delta$.

Lemma 18 (Store Update) *If* $\langle \Omega, \Gamma, \Delta \rangle \sim \langle S, V \rangle$ *and* $l \notin dom(\Omega)$ *and* $\Omega' = \Omega[l \mapsto \tau]$ *and* $\Omega'; \Gamma; \Delta \vdash l : \tau; \Delta$ *and* $\Omega'; \Gamma; \Delta \vdash v : \tau$ class $; \Delta$ *then* $\langle \Omega[l \mapsto \tau], \Gamma, \Delta \rangle \sim \langle S[l \mapsto v], V \rangle$

Theorem 19 (Subject Reduction for Expressions) *If all of the following hold:*

- $\Omega; \Gamma; \Delta \vdash e : \tau; \Delta'$
- $\langle \Omega, \Gamma, \Delta \rangle \sim \langle S, V \rangle$
- $S, V,$ *and* Ω *are well-formed*
- $\langle S, V, e \rangle \rightarrow \langle S', V', r \rangle$

then there exists Ω' *such that*

(i) $\Omega \Rightarrow \Omega'$

(ii) $\Omega'; \Gamma; \Delta' \vdash r : \tau, \Delta'$

(iii) $\langle \Omega', \Gamma, \Delta' \rangle \sim \langle S', V' \rangle$

(iv) Ω'*, *S'*, and *V' *are well-formed.*

Proof: We proceed by induction on the structure of e:

case v: Values, compatibility and well-formedness are re-established trivially.

case x: By assumption, $\Omega; \Gamma; \Delta \vdash x : \tau; \Delta'$. Therefore, by ($\textsc{Var}_\tau$)

$$\frac{x \in dom(\Delta)}{\Omega; \Gamma; \Delta \vdash x : \Delta(id); \Delta}$$

and thus $\Delta' = \Delta$. Therefore, since $x \in dom(\Delta)$ and $\langle \Omega, \Gamma, \Delta \rangle \sim \langle S, V \rangle$, then $x \in dom(V)$. Thus, ($\textsc{Var}_\rightarrow$) applies and $r = V(x)$ giving $\Omega, \Delta \vdash V(x) : \Delta(x); \Delta$ by compatibility. Let $\Omega' = \Omega$. Then the conclusions hold trivially as the stores are unchanged in this case.

case A: By a parallel argument to x using ($\textsc{Class}_\tau$) and S instead of ($\textsc{Var}_\tau$) and V.

case self: By assumption, $\Omega; \Gamma; \Delta \vdash \mathsf{self} : \tau; \Delta'$. Therefore ($\textsc{Self}_\tau$) applies:

$$\frac{\mathsf{self} \in dom(\Delta) \qquad \Delta(\mathsf{self}) = [F; M]}{\Omega; \Gamma; \Delta \vdash \mathsf{self} : [F; M]; \Delta}$$

Since $\mathsf{self} \in dom(\Delta)$ and $\langle \Omega, \Gamma, \Delta \rangle \sim \langle S, V \rangle$, then $\mathsf{self} \in dom(V)$ and $\Omega; \Gamma; \Delta \vdash V(\mathsf{self}) : \Delta(\mathsf{self}); \Delta$ by compatibility. Since V is well-formed, $V(\mathsf{self}) = l$ and

the reduction ($\textsc{Self}_{\rightarrow}$) applies with $r = l$. Thus, $\Omega; \Gamma; \Delta \vdash l : \Delta(\mathsf{self}); \Delta$ showing (ii). Let $\Omega' = \Omega$. As the stores are unchanged in this reduction, extension, compatibility, and well-formedness are re-established trivially.

case $@x$: By inspecting the type rules, the only rule that can apply is ($\textsc{Field}_{\tau}$):

$$\frac{\Omega; \Gamma; \Delta \vdash \mathsf{self} : \tau'; \Delta \qquad \Gamma \vdash \tau' <: [@x : \tau]}{\Omega; \Gamma; \Delta \vdash @x : \tau; \Delta}$$

and thus $\Delta' = \Delta$. In fact, by ($\textsc{Self}_{\tau}$), we know that $\tau' = [F; M]$ for some F, M. Since $\tau' <: [@x : \tau]$, then $@x : \tau \in F$ by ($\textsc{Object}_{<:}$). That is, $\Omega; \Gamma; \Delta \vdash \mathsf{self} : [@x : \tau, F'; M]; \Delta$.

Suppose $\langle S, V, \mathsf{self}\rangle \rightarrow \langle S', V', r\rangle$. As $\Omega; \Gamma; \Delta \vdash \mathsf{self} : \tau'; \Delta'$ then by induction there exists Ω' such that $\Omega \Rightarrow \Omega'$ (showing i), $\Omega'; \Gamma; \Delta \vdash r : \tau'; \Delta$ and $\langle \Omega', \Gamma, \Delta\rangle \sim \langle S, V\rangle$ (showing iii), and Ω' is well-formed (showing iv). Thus r is not *error* and so it must be a value. Therefore ($\textsc{Self}_{\rightarrow}$) applies and $r = l$, and $\Omega'; \Gamma; \Delta \vdash l : [@x : \tau, F'; M]; \Delta$. Note also that this means that $l \in \mathit{dom}(\Delta)$, $S = S'$ and $V = V'$. This also means we can reduce with ($\textsc{Field}_{\rightarrow}$).

By compatibility, $\Omega'; \Gamma; \Delta \vdash S(l) : [@x : \tau, F'; M]$ class $; \Delta$ and therefore $S(l)$ must be a class object with a $@x$ field: $S(l) = [A; @x = v, F_v; M]$ for some v. Then by Lemma 16, $\Omega'; \Gamma; \Delta \vdash v : \tau; \Delta$ showing (ii).

case $[e_1, \ldots, e_n]$: By inspecting the type rules, the only rule that can apply is ($\textsc{Tuple}_{\tau}$):

$$\frac{\Omega; \Gamma; \Delta_i \vdash e_i : \tau_i; \Delta_{i+1} \qquad i \in 1..n}{\Omega; \Gamma; \Delta_1 \vdash [e_1, \ldots, e_n] : (\tau_1 \times \cdots \times \tau_n); \Delta_{n+1}}$$

with $\Delta_1 = \Delta$. Let $\Omega_1 = \Omega$.

Suppose $\langle S_1, V_1, e_1\rangle \to \langle S_2, V_2, r_1\rangle$. Since $\Omega_1; \Gamma; \Delta_1 \vdash e_1 : \tau_1; \Delta_2$, then by induction there exists Ω_2 such that $\Omega_1 \Rightarrow \Omega_2$, $\Omega_2; \Gamma; \Delta_2 \vdash r_1 : \tau_1; \Delta_2$, $\langle \Omega_2, \Gamma, \Delta_2\rangle \sim \langle S_2, V_2\rangle$, and Ω_2 is well-formed. Therefore r_1 can not be *error* and must be a value v_1. We can apply this argument iteratively to show that for each $i \in 1..n$, r_i is a value v_i and there exists Ω_{i+1} such that $\Omega_{i+1}; \Gamma; \Delta_{i+1} \vdash r_i : \tau_i; \Delta_{i+1}$, and $\langle \Omega_{i+1}, \Gamma, \Delta_{i+1}\rangle \sim \langle S_{i+1}, V_{i+1}\rangle$ (showing (iii)), and Ω_{i+1} is well-formed (showing (iv)). Thus we can reduce with (TUPLE$_\to$). Note that by Lemma 13, $\Omega \Rightarrow \Omega_{i+1}$, and we have shown (i).

By Lemma 14 and Lemma 15, $\Omega_{i+1}; \Gamma; \emptyset \vdash v_j : \tau_i; \emptyset$ for $j \in 1..n$. Thus we can again apply (TUPLE$_\tau$) $\Omega_{i+1}; \Gamma; \Delta_{i+1} \vdash [v_1, \ldots, v_n] : (\tau_1 \times \cdots \times \tau_n); \Delta_{i+1}$, thus showing (ii).

case $e_1; e_2$: By inspecting the type rules, the only rul that can apply is (SEQ$_\tau$):

$$\text{(SEQ}_\tau\text{)}\quad \frac{\Omega; \Gamma; \Delta_1 \vdash e_1 : \tau_1; \Delta_2 \qquad \Omega; \Gamma; \Delta_2 \vdash e_2 : \tau_2; \Delta_3}{\Omega; \Gamma; \Delta_1 \vdash e_1; e_2 : \tau_2; \Delta_3}$$

Suppose $\langle S, V, e_1\rangle \to \langle S', V', r_1\rangle$. Then by induction, there exists Ω_2 such that $\Omega \Rightarrow \Omega_2$, $\Omega_2; \Gamma; \Delta_2 \vdash r_1 : \tau_1; \Delta_2$ and $\langle \Omega_2, \Gamma, \Delta_2\rangle \sim \langle S', V'\rangle$. Thus r_1 is not *error* by Lemma 12 and so r_1 must be a value. Therefore we can reduce via $\langle S', V', e_2\rangle \to \langle S'', V'', r_2\rangle$. Also by induction, there exists Ω_3 such that $\Omega_2 \Rightarrow \Omega_3$, $\Omega_3; \Gamma; \Delta_3 \vdash r_2 : \tau_2; \Delta_3$ and $\langle \Omega_3, \Gamma, \Delta_3\rangle \sim \langle S'', V''\rangle$. Thus r_2 must be a value v_2 and we can reduce with (SEQ$_\to$) and $\Omega_3; \Gamma; \Delta_3 \vdash v_2 : \tau_2; \Delta_3$. By Lemma 13, $\Omega \Rightarrow \Omega_3$, we have shown all of the conclusions.

case $x = e$: By assumption, $\Omega; \Gamma; \Delta \vdash x = e : \tau; \Delta'$. Therefore by (VAR ASSIGN$_\tau$),

$$\frac{\Omega; \Gamma; \Delta \vdash e : \tau; \Delta_1 \qquad \Delta_2 = \Delta_1[x \mapsto \tau]}{\Omega; \Gamma; \Delta \vdash x = e : \tau; \Delta_2}$$

with $\Delta_2 = \Delta'$. Suppose $\langle S, V, e\rangle \to \langle S_1, V_1, r\rangle$. Then by induction, there exists Ω' such that $\Omega \Rightarrow \Omega'$ (showing (i)) and $\Omega'; \Gamma; \Delta_1 \vdash r : \tau; \Delta_1$ and $\langle \Omega', \Gamma, \Delta_1\rangle \sim \langle S_1, V_1\rangle$. Thus r is not *error* and so it must be a value v and $\Omega'; \Gamma; \Delta_2 \vdash v : \tau; \Delta_2$ (showing (ii)). Therefore we can reduce via (ASSIGN LOCAL$_\to$) and so $S = S_1$, $V' = V_1[x \mapsto v]$. Since $V'(x) = v$ and $\Delta_2(x) = \tau$, then $\Omega'; \Gamma; \Delta_2 \vdash V'(x) : \Delta_2(x); \Delta_2$. Therefore $\langle \Omega', \Gamma, \Delta_2\rangle \sim \langle S', V'\rangle$ (showing (iii)). Also, (iv) holds since we did not update self in V' or update S, and by our inductive choice of Ω'.

case $@x = e$: By assumption, $\Omega; \Gamma; \Delta \vdash @x = e : \tau; \Delta'$. Therefore by (FIELD ASSIGN$_\tau$),

$$\frac{\Omega; \Gamma; \Delta \vdash e : \tau; \Delta' \qquad \Omega; \Gamma; \Delta' \vdash \mathsf{self} : \tau_s; \Delta' \qquad \Gamma \vdash \tau_s <: [@x : \tau'] \qquad \Gamma \vdash \tau <: \tau'}{\Omega; \Gamma; \Delta \vdash @x = e : \tau; \Delta'}$$

Suppose $\langle S, V, e\rangle \to \langle S', V', r_1\rangle$. Then by induction, there exists Ω' such that $\Omega \Rightarrow \Omega'$ and $\Omega'; \Gamma; \Delta' \vdash r_1 : \tau; \Delta'$ and $\langle \Omega', \Gamma, \Delta'\rangle \sim \langle S', V'\rangle$. Thus r_1 is not *error* and so it must be a value v. This establishes $\Omega'; \Gamma; \Delta' \vdash v : \tau; \Delta'$ and the first hypothesis in (ASSIGN FIELD$_\to$).

Suppose $\langle S', V', \mathsf{self}\rangle \to \langle S_2, V_2, r_2\rangle$. As $\Omega; \Gamma; \Delta' \vdash \mathsf{self} : \tau_s; \Delta'$ then $\Omega'; \Gamma; \Delta' \vdash \mathsf{self} : \tau_s; \Delta'$ by Lemma 14. Therefore by induction there exists Ω'' such that $\Omega' \Rightarrow \Omega''$, $\Omega''; \Gamma; \Delta' \vdash r_2 : \tau_s; \Delta'$, and $\langle \Omega'', \Gamma, \Delta'\rangle \sim \langle S_2, V_2\rangle$. Thus r_2 is not

error and so it must be a value. Therefore we can reduce using (SELF$_{\rightarrow}$) and $r = l$, establishing $\Omega''; \Gamma; \Delta' \vdash l : \tau_s; \Delta'$ and $S_2 = S'$ and $V_2 = V'$. We have also established the second hypothesis in (ASSIGN FIELD$_{\rightarrow}$).

Since, $\Omega''; \Gamma; \Delta' \vdash l : \tau_s; \Delta'$, then $\tau_s = [F; M]$ by (LOC$_{\tau}$). Further, since $\Gamma \vdash \tau_s <: [@x : \tau']$, $\Gamma \vdash [F; M] <: [@x : \tau']$. By inspecting the subtyping rules, (OBJECT$_{<:}$) must apply, which means that $@x : \tau' \in F$. Thus, $\Omega''; \Gamma; \Delta' \vdash l : [@x : \tau', F'; M]$ and $l \in dom(\Omega'')$. By compatibility, $\Omega''; \Gamma; \Delta' \vdash S'(l) : [@x : \tau', F'; M]$ class $; \Delta'$. By inspecting the type rules, the only value with this type is a class literal, thus, $S'(l)$ must be of the form: $[A; @x = v', F_v; M_v]$ for some A. By Lemma 16, $\Omega''; \Gamma; \Delta' \vdash v' : \tau' : \Delta'$. We have established the third hypothesis of (ASSIGN FIELD$_{\rightarrow}$).

Therefore, we can reduce using (ASSIGN FIELD$_{\rightarrow}$) with $S'' = S'[l \mapsto [A; @x = v, F_v; M_v]]$. Note that since we have not updated self, S'' is well-formed under this update, and Ω'' and V' are well-formed inductively, showing (iv). Since $\Omega'; \Gamma; \Delta' \vdash v : \tau; \Delta'$, $\Omega''; \Gamma; \Delta' \vdash v : \tau; \Delta'$ by Lemma 14. Thus we need only show that compatibility is re-established to show our conclusion.

Since $\Omega'; \Gamma; \Delta' \vdash v : \tau : \Delta'$ and $\Gamma \vdash \tau <: \tau'$, then $\Omega'; \Gamma; \Delta' \vdash v : \tau' : \Delta'$ by (SUBSUMPTION$_{\tau}$). Also $\Omega''; \Gamma; \Delta' \vdash v : \tau' : \Delta'$ by Lemma 14 which shows (ii).

Therefore, since $\Omega''; \Gamma; \Delta' \vdash [A; @x = v', F_v; M_v] : [@x : \tau', F'; M]$ class $; \Delta'$, then $\Omega''; \Gamma; \Delta' \vdash [A; @x = v, F_v; M_v] : [@x : \tau', F'; M]$ class $; \Delta'$ by Lemma 17. That is, $\Omega''; \Gamma; \Delta' \vdash S''(l) : [@x : \tau', F'; M]$ class and so $\langle \Omega'', \Gamma, \Delta' \rangle \sim \langle S'', V' \rangle$ showing (iii).

Finally, by Lemma 14, $\Omega \Rightarrow \Omega''$ showing (i).

case $x_1, \ldots, x_n = e$: Follows just like $x = e$ above.

case $\mathsf{new}\ e$: By assumption, $\Omega; \Gamma; \Delta \vdash \mathsf{new}\ e : \tau; \Delta'$. Therefore by ($\textsc{New}_\tau$):

$$\frac{\Omega; \Gamma; \Delta \vdash e : \tau\ \mathsf{class}\ ; \Delta' \qquad \tau = [F; M]}{\Omega; \Gamma; \Delta \vdash \mathsf{new}\ e : \tau; \Delta'}$$

Suppose $\langle S, V, e\rangle \rightarrow \langle S', V', r\rangle$. Then by induction, there exists Ω' such that $\Omega \Rightarrow \Omega'$, Ω' is well-formed, $\Omega'; \Gamma; \Delta' \vdash r : [F; M]\ \mathsf{class}\ ; \Delta'$ and $\langle \Omega', \Gamma, \Delta'\rangle \sim \langle S', V'\rangle$. Thus, r is not *error* and must be a class literal $v = [A; F_v; M_v]$ by Lemma 11. Let l be any location such that $l \notin dom(S')$. Then we can reduce using ($\textsc{New}_\rightarrow$) with $S'' = S'[l \mapsto v]$. Note S'' is well-formed since $v = [A; F_v; M_v]$. Let $\Omega'' = \Omega'[l \mapsto \tau]$. Trivially, $\Omega'', \Gamma, \Delta' \vdash l : \tau; \Delta'$, showing (ii). Also, since $l \notin dom(S')$, $l \notin dom(\Omega')$. Therefore since $\Omega' \Rightarrow \Omega''$, then $\Omega \Rightarrow \Omega''$ by Lemma 13 showing (i). Also, since $\langle \Omega', \Gamma, \Delta'\rangle \sim \langle S', V'\rangle$, then $\langle \Omega'', \Gamma, \Delta'\rangle \sim \langle S'', V'\rangle$ by Lemma 18 showing (iii). Since we did not add or change any classes in Ω' to produce Ω'', then it is also well-formed, showing (iv).

case $e_0.m(e_1, \ldots, e_n)$: By assumption, $\Omega; \Gamma; \Delta \vdash e_0.m(e_1, \ldots, e_n) : \tau; \Delta'$. Therefore

by ($\textsc{Call}_\tau$):

$$\Omega; \Gamma; \Delta_i \vdash e_i : \tau_i; \Delta_{i+1} \qquad i \in 1..n$$

$$\Omega; \Gamma; \Delta_{n+1} \vdash e_0 : \tau_0; \Delta_{n+2}$$

$$\frac{\Gamma \vdash \tau_0 <: [m : (\tau_1 \times \cdots \times \tau_n) \to \tau]}{\Omega; \Gamma; \Delta_0 \vdash e_0.m(e_1, \ldots, e_n) : \tau; \Delta_{n+2}}$$

with $\Delta_{n+2} = \Delta'$. Let $\Omega_1 = \Omega$. Suppose $\langle S_1, V_1, e_1\rangle \to \langle S_2, V_2, r_1\rangle$. By induction, there exists Ω_2 such that $\Omega_1 \Rightarrow \Omega_2$, Ω_2 is well-formed, $\Omega_2; \Gamma; \Delta_1 \vdash e_1 : \tau_1; \Delta_2$, and $\langle \Omega_2, \Gamma, \Delta_2\rangle \sim \langle S_2, V_2\rangle$. Therefore r_1 can not be *error* and must be a value v_1.

We can apply this argument iteratively to show that for each $i \in 1..n$, r_i is a value v_i and there exists Ω_{i+1} such that Ω_{i+1} is well-formed, $\Omega \Rightarrow \Omega_{i+1}$ by Lemma 13, $\Omega_{i+1}; \Gamma; \Delta_{i+1} \vdash r_i : \tau_i; \Delta_{i+1}$, and $\langle \Omega_{i+1}, \Gamma, \Delta_{i+1}\rangle \sim \langle S_{i+1}, V_{i+1}\rangle$.

Now suppose $\langle S_{n+1}, V_{n+1}, e_0\rangle \to \langle S_{n+2}, V_{n+2}, r_0\rangle$. By the same argument, r_0 is a value v_0 and there exists Ω_{n+2} such that Ω_{n+2} is well-formed, $\Omega \Rightarrow \Omega_{n+2}$ by Lemma 13, $\Omega_{n+2}; \Gamma; \Delta_{n+2} \vdash r_0 : \tau_0; \Delta_{n+2}$, and $\langle \Omega_{n+2}, \Gamma, \Delta_{n+2}\rangle \sim \langle S_{n+2}, V_{n+2}\rangle$. By subtyping, τ_0 must be an object and by Lemma 11 v_0 must be a location l. Thus, $v_0 = l$.

Also by subtyping $\tau_0 = [F; m : \eta, M]$ with $\Gamma \vdash \eta <: (\tau_1 \times \cdots \times \tau_n) \to \tau$.

By compatibility, $\Omega_{n+2}; \Gamma; \Delta_{n+2} \vdash S_{n+2}(l) : [F; m : \eta, M]$ class $; \Delta_{n+2}$ and therefore $S_{n+2}(l)$ must be a class object with a m method: $S_{n+2}(l) = [A; F_v; \mathsf{def}\ m(x_1, \ldots, x_n)$ $\eta = e_m, M_v]$.

Let $V_m = [\mathsf{self} \mapsto l, x_i \mapsto v_i]$ and $\Delta_M = [\mathsf{self} \mapsto \tau_0, x_i \mapsto \tau_i]$ for $i \in 1..n$.

Clearly V_m and Δ_M are compatible by construction and thus $\langle \Omega_{n+2}, \Gamma, \Delta_M \rangle \sim \langle S_{n+2}, V_m \rangle$.

Suppose $\langle S_{n+2}, V_m, e_m \rangle \to \langle S', V'_m, r_m \rangle$. Then by (Mono Method$_\tau$)

(Mono Method$_\tau$)

$$\frac{\Omega_{n+2}[A \mapsto \tau_a]; \Gamma; \Delta_m \vdash e_m : \tau; \Delta'_m}{A : [F; m : \eta, M] \text{ class } ; m : (\tau_1 \times \cdots \times \tau_n) \to \tau; \Omega_{n+2}; ; \Gamma; \Delta_{n+2} \vdash \text{def } m(x_1, \ldots, x_n) : \eta = e_m}$$

Thus, by induction, there exists Ω' such that $\Omega_{n+2} \Rightarrow \Omega'$, Ω' is well-formed, $\Omega', \Gamma; \Delta_M \vdash r_m : \tau; \Delta'_M$, and $\langle \Omega', \Gamma; \Delta'_M \rangle \sim \langle S', V'_m \rangle$. Then by Lemma 13, $\Omega \Rightarrow \Omega'$, showing (i). Also, r_m is not *error* and thus $r_m = v_m$ and therefore $\Omega', \Gamma; \Delta'_M \vdash v_m : \tau; \Delta'_M$. Therefore we can then apply (Call$_\to$): $\langle S_1, V_1, e_0.m(e_1, \ldots, e_n) \rangle \to \langle S', V_{n+2}, v_m \rangle$. By Lemma 15 $\Omega', \Gamma; \Delta' \vdash v_m : \tau; \Delta'$ showing (ii). Since $\Omega \Rightarrow \Omega_{n+2}$ and $\Omega_{n+2} \Rightarrow \Omega'$ then $\Omega \Rightarrow \Omega'$ by Lemma 13, showing (i).

Since $\langle \Omega', \Gamma; \Delta'_M \rangle \sim \langle S', V'_m \rangle$ and $\langle \Omega_{n+2}, \Gamma, \Delta' \rangle \sim \langle S_{n+2}, V_{n+2} \rangle$ then $\langle \Omega', \Gamma, \Delta' \rangle \sim \langle S', V_{n+2} \rangle$, showing (iii). Finally since Ω', S', and V_{n+2} are well-formed by induction, and we have shown (iv).

case $\mathsf{typecase}\ e_1\ \mathsf{when}\ (x : A)\ e_2\ \mathsf{else}\ e$: By assumption $\Omega; \Gamma; \Delta \vdash \mathsf{typecase}\ e_1\ \mathsf{when}\ (x : A)\ e_2\ \mathsf{else}\ e : \tau; \Delta'$. Therefore, by (TypeCase$_\tau$):

$$\frac{\Omega; \Gamma; \Delta \vdash e_1 : [F; M]; \Delta_1 \qquad \Omega; \Gamma; \Delta_1 \vdash A : \tau' \text{ class } ; \Delta_1 \qquad \Omega; \Gamma; \Delta_1, x : \tau' \vdash e_2 : \tau; \Delta_2 \qquad \Omega; \Gamma; \Delta_1 \vdash e_3 : \tau; \Delta_3 \qquad \Delta' = \Delta_2|_{dom(\Delta_1)} \uplus \Delta_3|_{dom(\Delta_1)}}{\Omega; \Gamma; \Delta \vdash \mathsf{typecase}\ e_1\ \mathsf{when}\ (x : A)\ e_2\ \mathsf{else}\ e_3 : \tau; \Delta'}$$

Suppose $\langle S, V, e_1\rangle \to \langle S_2, V_2, r_1\rangle$. Then by induction there exists Ω' such that $\Omega \Rightarrow \Omega'$, Ω' is well-formed, $\Omega'; \Gamma; \Delta_1 \vdash v_1 : [F; M]; \Delta_1$, and $\langle \Omega', \Gamma, \Delta_1\rangle \sim \langle S_2, V_2\rangle$. Thus, r_1 is not *error* and must be a value, and therefore must be a location $l = r_1$ by Lemma 11. Since S_2 is well-formed, $S_2(l) = [B; F_v; M_v]$ for some B, F_v, and M_v.

We have two cases, whether or not $B = A$.

Assume that $S_2(l) = [A; F_v; M_v]$. Then let $V_3 = V_2[x \mapsto l]$ and $\Delta_1' = \Delta_1[x \mapsto [F; M]]$. Since $\Omega'; \Gamma; \Delta_1 \vdash l : [F; M]; \Delta_1$, then $\langle \Omega'; \Gamma; \Delta_1'\rangle \sim \langle S_2, V_3\rangle$. Suppose $\langle S_2, V_3, e_2\rangle \to \langle S_3, V_4, r_2\rangle$. Then by induction, there exists Ω'' such that $\Omega' \Rightarrow \Omega''$, Ω'' is well-formed (showing (iv), $\Omega''; \Gamma; \Delta_2 \vdash r_2 : \tau; \Delta_2$ and $\langle \Omega''; \Gamma; \Delta_1'\rangle \sim \langle S_3, V_4\rangle$. Therefore r_2 is not *error* and must be a value $v_2 = r_2$. Therefore the rule (TypeCase Match$_\to$) applies and $\langle S, V, \mathsf{typecase}\ e_1\ \mathsf{when}\ (x : A)\ e_2\ \mathsf{else}\ e\rangle \to \langle S_3, V', v_2\rangle$ where $V' = V_4|_{dom(V_2)}$.

Recall that by (TypeCase$_\tau$), $\Delta' = \Delta_2|_{dom(\Delta_1)} \uplus \Delta_3|_{dom(\Delta_1)}$. Since $\Omega''; \Gamma; \Delta_2 \vdash v_2 : \tau; \Delta_2$ then $\Omega''; \Gamma; \Delta' \vdash v_2 : \tau; \Delta'$ by Lemma 15, showing (ii). Also $\Omega \Rightarrow \Omega''$ by Lemma 13 showing (i).

Since $\langle \Omega', \Gamma, \Delta_1\rangle \sim \langle S_2, V_2\rangle$, then $dom(\Delta_1) = dom(V_2)$ by compatibility. Furthermore, since $\langle \Omega''; \Gamma; \Delta_1'\rangle \sim \langle S_3, V_4\rangle$ then $\langle \Omega''; \Gamma; \Delta'\rangle \sim \langle S_3, V'\rangle$ by Lemma 10, showing (iii). This concludes this subcase.

Now assume that $S_2(l) = [B; F_v; M_v]$ with $B \neq A$. Suppose $\langle S_2, V_2, e_3\rangle \to \langle S_3, V_3, r_3\rangle$. Then by induction, there exists Ω_3 such that $\Omega' \Rightarrow \Omega_3$, Ω_3 is well-formed (showing (iv), $\Omega_3; \Gamma; \Delta_3 \vdash r_3 : \tau; \Delta_3$, and $\langle \Omega_3, \Gamma, \Delta_3\rangle \sim \langle S_3, V_3\rangle$.

Therefore r_3 is not *error* and must be a value $v_3 = r_3$. Therefore the rule (TypeCase Else$_\rightarrow$) applies and $\langle S, V, \mathsf{typecase}\ e_1\ \mathsf{when}\ (x : A)\ e_2\ \mathsf{else}\ e\rangle \rightarrow \langle S_3, V_3, v_3\rangle$.

By a parallel argument to the first case, $\Omega \Rightarrow \Omega_3$ (showing (i)), $\Omega_3; \Gamma; \Delta' \vdash v_3 : \tau; \Delta'$ (showing (ii)) and $\langle \Omega_3, \Gamma, \Delta'\rangle \sim \langle S_3, V_3\rangle$ (showing (iii)), which concludes this case.

□

Theorem 20 (Subject Reduction for Classes) *If* $\Omega; \Gamma; \Delta \vdash \mathsf{class}\ A : \sigma = d$ *and* $\Omega[cur\_class] = (A, \sigma)$ *and* $\langle \Omega, \Gamma, \Delta\rangle \sim \langle S, V\rangle$ *and* $\langle S, V, c\rangle \rightarrow \langle S', V, r\rangle$ *and* Ω, S, *and* V *are well-formed, then there exists* Ω' *such that*

(i) $\Omega \Rightarrow \Omega'$

(ii) $\langle \Omega', \Gamma; \Delta\rangle \sim \langle S', V\rangle$

(iii) $\Omega'; \Gamma; \Delta \vdash r : \tau; \Delta$

(iv) Ω *and* S' *are well-formed.*

Proof: Suppose $\langle S, V, c\rangle \rightarrow \langle S', V, r\rangle$. In order to show that (Class Def$_\rightarrow$) applies, we must only show that each field declaration in d reduces to a value. Proceed by induction on the number of qualifiers in σ.

Our base case is when σ is a mono-type τ. In this case, (Mono Class$_\tau$) applies:

$$\frac{\Omega; \Gamma; [\mathsf{self} \mapsto \tau] \vdash d_i \qquad i \in 1..n \qquad \text{labels}(\tau) = \{\text{label}(d_1), \ldots, \text{label}(d_n)\}}{\Omega; \Gamma; \Delta \vdash \mathsf{class}\ A : \tau = d_1, \ldots, d_n}$$

Without loss of generality, let $@x_i = e_i$ be some field declaration d_i. Then by (Mono Class$_\tau$), $\Omega; \Gamma; [\mathsf{self} \mapsto \tau] \vdash @x_i = e_i$. Therefore by (Field Decl$_\tau$):

$$\frac{\Omega; \Gamma; \emptyset \vdash e_i : \tau_i; \Delta_i \qquad \Gamma \vdash \tau <: [@x : \tau_i]}{\Omega; \Gamma; [\mathsf{self} \mapsto \tau] \vdash @x_i = e_i}$$

Since $\Omega; \Gamma; \emptyset \vdash e_i : \tau_i; \Delta_i$, then by Theorem 19 there exists Ω_i' such that $\Omega_i \Rightarrow \Omega_i'$, $\langle S_i, V_i, e_i \rangle \rightarrow \langle S_{i+1}, V_{i+1}, v_i \rangle$ with $\langle \Omega_i', \Gamma, \Delta_i \rangle \sim \langle S_{i+1}, V_{i+1} \rangle$. Note that by Lemma 14, $\Omega_i'; \Gamma; [\mathsf{self} \mapsto \tau] \vdash @x_i = e_i$. Since $@x_i = e_i$ was chosen arbitrarily, we can apply this argument to each of the n field definitions in d. Therefore (Class Def$_\rightarrow$) applies and $r \neq \mathit{error}$. Furthermore, by Lemma 13, $\Omega_n'; \Gamma; \Delta \vdash \mathsf{class}\ A : \tau = d_1, \ldots, d_n$.

The inductive case follows immediately since (Poly Class$_\tau$) uses our induction hypothesis as its only premise. □

Theorem 21 (Type Soundness) *If* $\vdash P$ *then* $\langle \emptyset, \emptyset, P \rangle \rightarrow \langle S, V, r \rangle$ *where* $r \neq$ *error.*

Proof: Let $P = c_1 \cdots c_n; e$. Assume that $\langle \emptyset, \emptyset, P \rangle \rightarrow \langle S, V, r \rangle$. Since $\vdash P$ then (Program$_\tau$) applies and we can repeatedly apply Theorem 20 to each class definition to construct Ω such that $\langle \Omega, \emptyset, \emptyset \rangle \sim \langle S', \emptyset \rangle$ with $\langle \emptyset, \emptyset, c_1 \cdots c_n \rangle \rightarrow \langle S', \emptyset, v \rangle$.

Again by ($\textsc{Program}_\tau$) $\Omega; \emptyset; \emptyset \vdash e : \tau; \Delta$ and by Theorem 19, there exists Ω' such that $\langle S', \emptyset, e\rangle \rightarrow \langle S, V, v'\rangle$ with and $\Omega'; \emptyset; \emptyset \vdash v' : \tau$. Therefore $r \neq \mathit{error}$. $\square$

Appendix C

Proofs for DynRuby

C.1 Type Checking Rules

Figure C.1 presents a portion of a type checking system designed specifically for the output of our translation (though it is in fact sound in general—it just may not be able to type programs that have not been translated). This type system is representative of the static typing discipline enforced by DRuby, though it is far simpler.

The first group of rules in Figure C.1 prove judgments of the form $MT; \Gamma \vdash e : \tau$, meaning with method type table MT, a mapping from names $A.m$ to method signatures, and in type environment Γ, a mapping from variables to types, expression e has type τ. Types τ are either *string*, *bool*, or a class A, and method signatures σ consist of argument and result types.

($\textsc{Var}_\tau$) and ($\textsc{Inst}_\tau$) are trivial. ($\textsc{Wrap}_\tau$), ($\textsc{SEval}_\tau$), and ($\textsc{Blame}_\tau$) give the corresponding expressions any type; the subexpressions in the first two forms are evaluated with full dynamic checking, and the last form is used to abort execution due to an error in a dynamic region of the code. ($\textsc{Call}_\tau$) types the receiver and the

$\boxed{MT; \Gamma \vdash e : \tau}$

$$\begin{array}{rcl} \tau & ::= & \mathit{string} \mid \mathit{bool} \mid A \\ \sigma & ::= & \tau_1 \times \cdots \times \tau_n \to \tau \end{array}$$

$$(\textsc{Var}_\tau)\ \frac{x \in \Gamma}{MT; \Gamma \vdash x : \Gamma(x)} \qquad (\textsc{Inst}_\tau)\ \frac{}{MT; \Gamma \vdash \mathsf{new}\ A : A} \qquad (\textsc{Wrap}_\tau)\ \frac{}{MT; \Gamma \vdash [\![e]\!]_\ell : \tau}$$

$$(\textsc{SEval}_\tau)\ \frac{MT; \Gamma \vdash e : \mathit{string}}{MT; \Gamma \vdash \mathsf{safe\_eval}_\ell\ e : \tau} \qquad (\textsc{Blame}_\tau)\ \frac{}{MT; \Gamma \vdash \mathsf{blame}\ \ell : \tau}$$

$$(\textsc{Call}_\tau)\ \frac{MT; \Gamma \vdash e_i : \tau_i \qquad i \in 0..n \qquad m \neq \mathsf{method\_missing} \qquad MT(\tau_0.m) = \tau_1 \times \cdots \times \tau_n \to \tau}{MT; \Gamma \vdash e_0.m(e_1, \ldots, e_n) : \tau}$$

$$(\textsc{Def}_\tau)\ \frac{MT(A.m) = \tau_1 \times \cdots \times \tau_n \to \tau \qquad \Gamma' = (\mathsf{self} \mapsto A, x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n) \qquad MT; \Gamma' \vdash e : \tau}{MT; \Gamma \vdash \mathsf{def}\ \ell\ A.m(x_1, \ldots, x_n) = e : \mathit{bool}}$$

$\boxed{MT \vdash d; MT' \qquad MT \vdash e}$

$$(\textsc{Def}'_\tau)\ \frac{\sigma = \tau_1 \times \cdots \times \tau_n \to \tau \qquad A.m \in \mathit{dom}(MT) \Rightarrow MT(A.m) = \sigma}{MT \vdash \mathsf{def}\ \ell\ A.m(x_1, \ldots, x_n) = \mathsf{blame}\ \ell; (A.m \mapsto \sigma), MT}$$

$$(\textsc{Prog}_\tau)\ \frac{MT \vdash d, MT' \qquad MT' \vdash e}{MT \vdash d; e} \qquad (\textsc{Prog-Expr}_\tau)\ \frac{MT; \emptyset \vdash e : \tau}{MT \vdash e}$$

Figure C.1: Type checking rules for DynRuby (selected rules)

arguments, and searches for a method signature for $\tau_0.m$ in MT; for this search to be successful, τ_0 must be a class A whose m method was defined. As expected, it matches the formal and actual argument types and extracts the result type from the signature. Note that we omit subtyping from our type system, also to keep things simple, but it is straightforward to add.

($\textsc{Def}_\tau$) types the definition of a method. The defined method $A.m$ must have a signature in MT, and the body e is type checked in the appropriate environment. The definition itself returns false, so the type of the definition is *bool*.

Note that ($\textsc{Def}_\tau$) applies to methods defined in the "middle" of a program. Recall that the translation defined by ($\textsc{Prog}_{\rightsquigarrow}$) produces a program of the form $(e_d; e)$, where e_d is a sequence of method definitions. The bottom part of Figure C.1 gives rules for typing programs of this form.

($\textsc{Def}'_\tau$) proves a judgment of the form $MT \vdash d; MT'$, where MT' is MT but with a method signature for d added. If there is more than one definition of the same method, it must have the same signature in MT'. In ($\textsc{Def}'_\tau$), the body of d must consist solely of a blame expression, which will be the case for the method definitions from e_d in our translated program. Because the body is a blame expression, we need not type check it. As a side note, since this is a type checking system, we have not specified how to come up with method signature σ. In practice, it could be supplied by type annotations or, in the case of DRuby, also by type inference.

The last two rules define judgment $MT \vdash e$, which given an expression $(e_d; e)$, creates a method table MT' with signatures for the definitions in e_d and then type checks e using that method table. These last two rules are non-deterministic, but

we generally will use ($\textsc{Prog}_\tau$) to accumulate as large a method table as possible from the initial set of definitions, and then check the remainder of the expression uses that method table.

C.2 Complete Formalism and Proofs

In this appendix, we give the full operational semantics (Figure C.2), program transformation (Figure C.3), and type checking system (Figure C.7) for DynRuby, which were abbreviated in the body of the paper due to lack of space.

C.2.1 Translation Faithfulness

Definition 22 *We write $\mathcal{P} \vdash M \rightsquigarrow M'$ if all of the following hold:*

1. $M = (d_1, \ldots, d_n)$
2. *For all $i \in 1..n$, we have $\mathcal{P} \vdash d_i \rightsquigarrow e_i$*
3. *M' is the method table consisting of $e_1; \ldots; e_n$ flattened and treated as a list of definitions*

Lemma 23 *Suppose $\langle M, V, e\rangle \rightarrow \langle M', \mathcal{P}', v\rangle$ and $\mathcal{P}' \subseteq \mathcal{P}$ and $\mathcal{P} \vdash e \rightsquigarrow e_\mathcal{P}$. Further assume $V_\mathcal{P}|_{dom(V)} = V$ and $\mathcal{P} \vdash M \rightsquigarrow M_\mathcal{P}$. Then $\langle M_\mathcal{P}, V_\mathcal{P}, e_\mathcal{P}\rangle \rightarrow \langle M'_\mathcal{P}, \mathcal{P}'', v\rangle$ where $\mathcal{P} \vdash M' \rightsquigarrow M'_\mathcal{P}$.*

$$\text{(Eq-T)}\quad \frac{\langle M,\, V, e_1\rangle \to \langle M_1, \mathcal{P}_1, v\rangle \quad \langle M_1,\, V, e_2\rangle \to \langle M_2, \mathcal{P}_2, v\rangle}{\langle M,\, V, e_1 \equiv e_2\rangle \to \langle M_2, (\mathcal{P}_1 \cup \mathcal{P}_2), \mathsf{true}\rangle}$$

$$\text{(Eq-F)}\quad \frac{\langle M,\, V, e_1\rangle \to \langle M_1, \mathcal{P}_1, v_1\rangle \quad \langle M_1,\, V, e_2\rangle \to \langle M_2, \mathcal{P}_2, v_2\rangle \qquad v_1 \neq v_2}{\langle M,\, V, e_1 \equiv e_2\rangle \to \langle M_2, (\mathcal{P}_1 \cup \mathcal{P}_2), \mathsf{false}\rangle}$$

$$\text{(Let)}\quad \frac{\langle M,\, V, e_1\rangle \to \langle M_1, \mathcal{P}_1, v_1\rangle \quad \langle M_1, (x : v_1,\, V), e_2\rangle \to \langle M_2, \mathcal{P}_2, v_2\rangle}{\langle M,\, V, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\rangle \to \langle M_2, (\mathcal{P}_1 \cup \mathcal{P}_2), v_2\rangle}$$

$$\text{(Blame)}\quad \frac{}{\langle M,\, V, \mathsf{blame}\ \ell\rangle \to \langle M,\, V, \mathsf{blame}\ \ell\rangle}$$

$$\text{(Value)}\quad \frac{}{\langle M,\, V, v\rangle \to \langle M, \emptyset, v\rangle}$$

$$\text{(If-T)}\quad \frac{\langle M,\, V, e_1\rangle \to \langle M_1, \mathcal{P}_1, \mathsf{true}\rangle \quad \langle M_1,\, V, e_2\rangle \to \langle M_2, \mathcal{P}_2, v_2\rangle}{\langle M,\, V, \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \to \langle M_2, (\mathcal{P}_1 \cup \mathcal{P}_2), v_2\rangle}$$

$$\text{(Var)}\quad \frac{}{\langle M,\, V, x\rangle \to \langle M, \emptyset,\, V(x)\rangle}$$

$$\text{(If-F)}\quad \frac{\langle M,\, V, e_1\rangle \to \langle M_1, \mathcal{P}_1, \mathsf{false}\rangle \quad \langle M_1,\, V, e_3\rangle \to \langle M_3, \mathcal{P}_3, v_3\rangle}{\langle M,\, V, \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \to \langle M_3, (\mathcal{P}_1 \cup \mathcal{P}_3), v_3\rangle}$$

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

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

$$\text{(Def)}\quad \frac{}{\langle M,\, V, d\rangle \to \langle (d, M), \emptyset, \mathsf{false}\rangle}$$

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

$$\text{(Seq)}\quad \frac{\langle M,\, V, e_1\rangle \to \langle M_1, \mathcal{P}_1, v_1\rangle \quad \langle M_1,\, V, e_2\rangle \to \langle M_2, \mathcal{P}_2, v_2\rangle}{\langle M,\, V, e_1; e_2\rangle \to \langle M_2, (\mathcal{P}_1 \cup \mathcal{P}_2), v_2\rangle}$$

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

Figure C.2: Instrumented big-step operational semantics for DynRuby (excluding blame and error rules)

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

$$\frac{\mathcal{P} \vdash e_1 \rightsquigarrow e_1' \qquad \mathcal{P} \vdash e_2 \rightsquigarrow e_2'}{\mathcal{P} \vdash e_1 \equiv e_2 \rightsquigarrow e_1' \equiv e_2'}\ (\textsc{Eq}_{\rightsquigarrow}) \qquad \frac{\mathcal{P} \vdash e_1 \rightsquigarrow e_1' \qquad \mathcal{P} \vdash e_2 \rightsquigarrow e_2'}{\mathcal{P} \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rightsquigarrow \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2'}\ (\textsc{Let}_{\rightsquigarrow})$$

$$\frac{\mathcal{P} \vdash e_1 \rightsquigarrow e_1' \qquad \mathcal{P} \vdash e_2 \rightsquigarrow e_2' \qquad \mathcal{P} \vdash e_3 \rightsquigarrow e_3'}{\mathcal{P} \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rightsquigarrow \mathsf{if}\ e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3'}\ (\textsc{If}_{\rightsquigarrow}) \qquad \frac{\mathcal{P} \vdash e_i \rightsquigarrow e_i' \qquad i \in 0..n \qquad m \neq \mathsf{send}}{\mathcal{P} \vdash e_0.m(e_1, \ldots, e_n) \rightsquigarrow e_0'.m(e_1', \ldots, e_n')}\ (\textsc{Call}_{\rightsquigarrow})$$

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

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

$$\frac{\mathcal{P} \vdash e \rightsquigarrow e' \qquad m \neq \mathsf{method\_missing}}{\mathcal{P} \vdash \mathsf{def}\ \ell\ A.m(x_1, \ldots, x_n) = e \rightsquigarrow \mathsf{def}\ \ell\ A.m(x_1, \ldots, x_n) = e'}\ (\textsc{Def}_{\rightsquigarrow})$$

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

$$\frac{\mathcal{P} \vdash e \rightsquigarrow e'}{\mathcal{P} \vdash [\![e]\!]_\ell \rightsquigarrow [\![e']\!]_\ell}\ (\textsc{Wrap}_{\rightsquigarrow})$$

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

Figure C.3: Transformation to static constructs (complete)

$$\frac{e \in \{x, v, \mathsf{blame}\ \ell\}}{e \hookrightarrow_\ell e}\ (\textsc{Refl}_{\hookrightarrow}) \qquad \frac{e_1 \hookrightarrow_\ell e_1' \qquad e_2 \hookrightarrow_\ell e_2'}{e_1; e_2 \hookrightarrow_\ell e_1'; e_2'}\ (\textsc{Seq}_{\hookrightarrow}) \qquad \frac{e_1 \hookrightarrow_\ell e_1' \qquad e_2 \hookrightarrow_\ell e_2'}{e_1 \equiv e_2 \hookrightarrow_\ell e_1' \equiv e_2'}\ (\textsc{Eq}_{\hookrightarrow})$$

$$\frac{e_1 \hookrightarrow_\ell e_1' \qquad e_2 \hookrightarrow_\ell e_2'}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \hookrightarrow_\ell \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2'}\ (\textsc{Let}_{\hookrightarrow}) \qquad \frac{e_1 \hookrightarrow_\ell e_1' \qquad e_2 \hookrightarrow_\ell e_2' \qquad e_3 \hookrightarrow_\ell e_3'}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \hookrightarrow_\ell \mathsf{if}\ [\![e_1']\!]_\ell\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3'}\ (\textsc{If}_{\hookrightarrow})$$

$$\frac{e_i \hookrightarrow_\ell e_i' \qquad i \in 0..n}{e_0.m(e_1, \ldots, e_n) \hookrightarrow_\ell [\![e_0']\!]_\ell.m(e_1', \ldots, e_n')}\ (\textsc{Call}_{\hookrightarrow}) \qquad \frac{e \hookrightarrow_\ell e'}{\mathsf{eval}_{\ell'}\ e \hookrightarrow_\ell \mathsf{safe\_eval}_{\ell'}\ [\![e']\!]_{\ell'}}\ (\textsc{Eval}_{\hookrightarrow})$$

$$\frac{e_i \hookrightarrow_\ell e_i' \qquad i \in 0..n}{e_0.\mathsf{send}_\ell(e_1, \ldots, e_n) \hookrightarrow_\ell \mathsf{safe\_eval}_\ell\ \text{“}e_0'.e_1'(e_2', \ldots, e_n')\text{”}}\ (\textsc{Send}_{\hookrightarrow}) \qquad \frac{}{\mathsf{def}\ \ell'\ A.m(x_1, \ldots, x_n) = e \hookrightarrow_\ell \mathsf{blame}\ \ell'}\ (\textsc{Def}_{\hookrightarrow})$$

$$\frac{e \hookrightarrow_{\ell'} e'}{[\![e]\!]_\ell \hookrightarrow_{\ell'} [\![e']\!]_\ell}\ (\textsc{Wrap}_{\hookrightarrow}) \qquad \frac{e \hookrightarrow_{\ell'} e'}{\mathsf{safe\_eval}_\ell\ e \hookrightarrow_{\ell'} \mathsf{safe\_eval}_\ell\ [\![e']\!]_\ell}\ (\textsc{SEval}_{\hookrightarrow})$$

Figure C.4: Safe evaluation rules (complete)

Proof: By induction on the derivation of $\langle M, V, e\rangle \to \langle M', \mathcal{P}, v\rangle$. We proceed by case analysis on the last rule applied. In this proof, we use $\cdot$ to indicate profiles we do not refer to.

Case (Eval): We have

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

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

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

$$\text{(SEval-Blame)}\quad \frac{\langle M, V, e\rangle \to \langle M', \mathcal{P}, v\rangle \qquad v \in \{[\![\mathsf{true}]\!]_{\ell'}, [\![\mathsf{false}]\!]_{\ell'}, [\![\mathsf{new}\ A]\!]_{\ell'}\}}{\langle M, V, \mathsf{safe\_eval}_\ell\ e\rangle \to \langle M', \mathcal{P}, \mathsf{blame}\ \ell\rangle}$$

$$\text{(SEval-Blame-Parse)}\quad \frac{\langle M, V, e\rangle \to \langle M', \mathcal{P}, v\rangle \qquad v = s \vee v = [\![s]\!]_{\ell'} \qquad \nexists parse(s)}{\langle M, V, \mathsf{safe\_eval}_\ell\ e\rangle \to \langle M', \mathcal{P}, \mathsf{blame}\ \ell\rangle}$$

$$\text{(Wrap)}\quad \frac{\langle M, V, e\rangle \to \langle M, \mathcal{P}, r\rangle}{\langle M, V, [\![e]\!]_\ell\rangle \to \langle M, \mathcal{P}, [\![r]\!]_\ell\rangle}$$

$$\text{(Wrap-Define)}\quad \frac{\langle M, V, e\rangle \to \langle M', \mathcal{P}, r\rangle \qquad M' \neq M}{\langle M, V, [\![e]\!]_\ell\rangle \to \langle M, \mathcal{P}, \mathsf{blame}\ \ell\rangle}$$

$$\text{(Unwrap)}\quad \frac{}{\langle M, V, [\![[\![r]\!]_{\ell'}]\!]_\ell\rangle \to \langle M, \emptyset, [\![r]\!]_\ell\rangle}$$

$$\text{(Wrap-error)}\quad \frac{}{\langle M, V, [\![error]\!]_\ell\rangle \to \langle M, \emptyset, \mathsf{blame}\ \ell\rangle}$$

$$\text{(Eq-Wrap-T)}\quad \frac{\langle M, V, e_1\rangle \to \langle M_1, \mathcal{P}_1, v_1\rangle \quad \langle M_1, V, e_2\rangle \to \langle M_2, \mathcal{P}_2, v_2\rangle \quad (v_1 = v \vee v_1 = [\![v]\!]_{\ell_1}) \qquad (v_2 = v \vee v_2 = [\![v]\!]_{\ell_2})}{\langle M, V, e_1 \equiv e_2\rangle \to \langle M_2, (\mathcal{P}_1 \cup \mathcal{P}_2), \mathsf{true}\rangle}$$

$$\text{(Eq-Wrap-F)}\quad \frac{\langle M, V, e_1\rangle \to \langle M_1, \mathcal{P}_1, v_1\rangle \quad \langle M_1, V, e_2\rangle \to \langle M_2, \mathcal{P}_2, v_2\rangle \quad (v_1 = v_1' \vee v_1 = [\![v_1']\!]_{\ell_1}) \qquad (v_2 = v_2' \vee v_2 = [\![v_2']\!]_{\ell_2}) \quad v_1' \neq v_2'}{\langle M, V, e_1 \equiv e_2\rangle \to \langle M_2, (\mathcal{P}_1 \cup \mathcal{P}_2), \mathsf{false}\rangle}$$

Figure C.5: Additional operational semantics rule wrapped expressions (1/2)

(If-Wrap-T)

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

(If-Wrap-F)

$$\frac{\langle M,\, V, e_1\rangle \to \langle M_1, \mathcal{P}_1, [\![\mathsf{false}]\!]_\ell\rangle \qquad \langle M_1,\, V, e_3\rangle \to \langle M_3, \mathcal{P}_3, v_3\rangle}{\langle M,\, V, \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \to \langle M_3, (\mathcal{P}_1 \cup \mathcal{P}_3), v_3\rangle}$$

(If-Wrap-Blame)

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

(Call-Wrap)

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

(Call-Meth-Blame)

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

(Call-Type-Blame)

$$\frac{\langle M_i,\, V, e_i\rangle \to \langle M_{i+1}, \mathcal{P}_i, v_i\rangle \qquad i \in 0..n \qquad v_0 \in \{[\![\mathsf{true}]\!]_{\ell''}, [\![\mathsf{false}]\!]_{\ell''}, [\![s]\!]_{\ell''}\}}{\langle M_0,\, V, e_0.m(e_1, \ldots, e_n)\rangle \to \langle M_{n+1}, \bigcup_i \mathcal{P}_i, \mathsf{blame}\ \ell''\rangle}$$

Figure C.6: Additional operational semantics rule wrapped expressions (2/2)

$\boxed{MT; \Gamma \vdash e : \tau}$

$$\begin{array}{rcl} \tau & ::= & \mathit{string} \mid \mathit{bool} \mid A \\ \sigma & ::= & \tau_1 \times \cdots \times \tau_n \to \tau \end{array}$$

$$(\textsc{Var}_\tau)\ \frac{x \in \Gamma}{MT; \Gamma \vdash x : \Gamma(x)} \qquad (\textsc{String}_\tau)\ \frac{}{MT; \Gamma \vdash s : \mathit{string}} \qquad (\textsc{Bool}_\tau)\ \frac{e \in \{\mathsf{true}, \mathsf{false}\}}{MT; \Gamma \vdash e : \mathit{bool}}$$

$$(\textsc{Inst}_\tau)\ \frac{}{MT; \Gamma \vdash \mathsf{new}\ A : A} \qquad (\textsc{Blame}_\tau)\ \frac{}{MT; \Gamma \vdash \mathsf{blame}\ \ell : \tau} \qquad (\textsc{SEval}_\tau)\ \frac{MT; \Gamma \vdash e : \mathit{string}}{MT; \Gamma \vdash \mathsf{safe\_eval}_\ell\ e : \tau}$$

$$(\textsc{Wrap}_\tau)\ \frac{}{MT; \Gamma \vdash [\![e]\!]_\ell : \tau} \qquad (\textsc{Seq}_\tau)\ \frac{MT; \Gamma \vdash e_1 : \tau_1 \quad MT; \Gamma \vdash e_2 : \tau_2}{MT; \Gamma \vdash e_1; e_2 : \tau_2} \qquad (\textsc{Eq}_\tau)\ \frac{MT; \Gamma \vdash e_1 : \tau_1 \quad MT; \Gamma \vdash e_2 : \tau_2}{MT; \Gamma \vdash e_1 \equiv e_2 : \mathit{bool}}$$

$$(\textsc{Let}_\tau)\ \frac{MT; \Gamma \vdash e_1 : \tau_1 \quad MT; x : \tau_1, \Gamma \vdash e_2 : \tau_2}{MT; \Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \qquad (\textsc{If}_\tau)\ \frac{MT; \Gamma \vdash e_1 : \mathit{bool} \quad MT; \Gamma \vdash e_2 : \tau \quad MT; \Gamma \vdash e_3 : \tau}{MT; \Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}$$

$$(\textsc{Call}_\tau)\ \frac{MT; \Gamma \vdash e_i : \tau_i \qquad i \in 0..n \qquad m \neq \mathsf{method\_missing} \qquad MT(\tau_0.m) = \tau_1 \times \cdots \times \tau_n \to \tau}{MT; \Gamma \vdash e_0.m(e_1, \ldots, e_n) : \tau}$$

Figure C.7: Type checking rules for DynRuby (complete)

$$\frac{MT(A.m) = \tau_1 \times \cdots \times \tau_n \to \tau \qquad MT; (\mathsf{self} \mapsto A, x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n) \vdash e : \tau}{MT; \Gamma \vdash \mathsf{def}\ \ell\ A.m(x_1, \ldots, x_n) = e : \mathit{bool}}\ (\textsc{Def}_\tau)$$

$\boxed{MT \vdash d; MT'}$

$$\frac{\sigma = \tau_1 \times \cdots \times \tau_n \to \tau \qquad A.m \in \mathit{dom}(MT) \Rightarrow MT(A.m) = \sigma \qquad MT' = (A.m \mapsto \sigma), MT}{MT \vdash \mathsf{def}\ \ell\ A.m(x_1, \ldots, x_n) = \mathsf{blame}\ \ell; MT'}\ (\textsc{Def}'_\tau)$$

$\boxed{MT \vdash e}$

$$\frac{MT \vdash d; MT' \qquad MT' \vdash e}{MT \vdash d; e}\ (\textsc{Prog}_\tau) \qquad \frac{MT; \emptyset \vdash e : \tau}{MT \vdash e}\ (\textsc{Prog-Expr}_\tau)$$

Figure C.8: Type checking rules for DynRuby (complete)

and $V_{\mathcal{P}}|_{\mathit{dom}(V)} = V$ and $\mathcal{P} \vdash M \rightsquigarrow M_{\mathcal{P}}$. We also have $\mathcal{P} \vdash \mathsf{eval}_\ell\ e \rightsquigarrow e_{\mathcal{P}}$ where $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2 \cup [\ell \mapsto s]$. Thus by ($\textsc{Eval}_{\rightsquigarrow}$), we have

($\textsc{Eval}_{\rightsquigarrow}$)

$$\mathcal{P} \vdash e \rightsquigarrow e' \qquad \mathcal{P} \vdash \mathit{parse}(s_j) \rightsquigarrow e_j \qquad s_j \in \mathcal{P}(\ell) \qquad x \text{ fresh}$$

$$e'' = \left(\begin{array}{l} \mathsf{let}\ x = e'\ \mathsf{in} \\ \quad \mathsf{if}\ x \equiv s_1\ \mathsf{then}\ e_1 \\ \quad \mathsf{else\ if}\ x \equiv s_2\ \mathsf{then}\ e_2 \\ \quad \cdots \\ \quad \mathsf{else\ safe\_eval}_\ell\ x \end{array}\right)$$

$$\mathcal{P} \vdash \mathsf{eval}_\ell\ e \rightsquigarrow e''$$

Then since $\mathcal{P}_1 \subseteq \mathcal{P}$, by induction we have $\langle M_{\mathcal{P}}, V_{\mathcal{P}}, e' \rangle \to \langle M'_{\mathcal{P}}, \cdot, s \rangle$ with $\mathcal{P} \vdash M_1 \rightsquigarrow M'_{\mathcal{P}}$. Notice also $s \in \mathcal{P}(\ell)$, and assume without loss of generality that $s = s_1$. Let $V'_{\mathcal{P}} = x : s, V_{\mathcal{P}}$.

Combining the last hypothesis of (Eval) with $\mathcal{P}_1 \subseteq \mathcal{P}$ and $\mathcal{P} \vdash \mathit{parse}(s_1) \rightsquigarrow e_1$ and $V'_{\mathcal{P}}|_{\mathit{dom}(V)} = V$ (since x is fresh) and $\mathcal{P} \vdash M_1 \rightsquigarrow M'_{\mathcal{P}}$, we can apply induction to get $\langle M'_{\mathcal{P}}, V'_{\mathcal{P}}, e_1\rangle \to \langle M''_{\mathcal{P}}, \cdot, v\rangle$ where $\mathcal{P} \vdash M_2 \rightsquigarrow M''_{\mathcal{P}}$.

Then combining the derived reductions using (Let) and (If-T), we have $\langle M_{\mathcal{P}}, V_{\mathcal{P}}, e''\rangle \to \langle M''_{\mathcal{P}}, \cdot, v\rangle$ where $\mathcal{P} \vdash M_2 \rightsquigarrow M''_{\mathcal{P}}$, which is the conclusion.

Case (Send): We have

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

and $V_{\mathcal{P}}|_{\mathit{dom}(V)} = V$ and $\mathcal{P} \vdash M \rightsquigarrow M_{\mathcal{P}}$. We also have $\mathcal{P} \vdash e_0.\mathsf{send}_\ell(e_1, \ldots, e_n) \rightsquigarrow e_{\mathcal{P}}$ where $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2 \cup [\ell \mapsto s]$. Thus by ($\text{Send}_{\rightsquigarrow}$), we have

$$\frac{\text{(Send}_{\rightsquigarrow}\text{)}\qquad \mathcal{P} \vdash e_i \rightsquigarrow e'_i \qquad i \in 0..n \qquad s_j \in \mathcal{P}(\ell) \qquad x \text{ fresh} \qquad e' = \left(\begin{array}{l} \mathsf{let}\ x = e'_1\ \mathsf{in} \\ \quad \mathsf{if}\ x \equiv s_1\ \mathsf{then}\ e'_0.\mathit{parse}(s_1)(e'_2, \ldots, e'_n) \\ \quad \mathsf{else\ if}\ x \equiv s_2\ \mathsf{then}\ e'_0.\mathit{parse}(s_2)(e'_2, \ldots, e'_n) \\ \quad \ldots \\ \quad \mathsf{else\ safe\_eval}_\ell\ \text{“}e'_0.x(e'_2, \ldots, e'_n)\text{”} \end{array}\right)}{\mathcal{P} \vdash e_0.\mathsf{send}_\ell(e_1, \ldots, e_n) \rightsquigarrow e'}$$

Then since $\mathcal{P}_1 \subseteq \mathcal{P}$, by induction we have $\langle M_{\mathcal{P}}, V_{\mathcal{P}}, e'_1\rangle \to \langle M'_{\mathcal{P}}, \cdot, s\rangle$ with $\mathcal{P} \vdash M_1 \rightsquigarrow M'_{\mathcal{P}}$. Notice also $s \in \mathcal{P}(\ell)$, and assume without loss of generality that $s = s_1$. Let $V'_{\mathcal{P}} = x : s, V_{\mathcal{P}}$.

Combining the last hypothesis of (SEND) with $\mathcal{P}_2 \subseteq \mathcal{P}$ and $\mathcal{P} \vdash e_i \rightsquigarrow e'_i$ and $V'_{\mathcal{P}}|_{dom(V)} = V$ (since x is fresh) and $\mathcal{P} \vdash M_1 \rightsquigarrow M'_{\mathcal{P}}$, we can apply induction to get $\langle M'_{\mathcal{P}}, V'_{\mathcal{P}}, e'_0.m(e'_2, \ldots, e'_n)\rangle \to \langle M''_{\mathcal{P}}, \cdot, v\rangle$ where $\mathcal{P} \vdash M_2 \rightsquigarrow M''_{\mathcal{P}}$.

Then combining the derived reductions using (LET) and (IF-T), and given that $m = \mathit{parse}(s)$, we have $\langle M_{\mathcal{P}}, V_{\mathcal{P}}, e'\rangle \to \langle M''_{\mathcal{P}}, \cdot, v\rangle$ where $\mathcal{P} \vdash M_2 \rightsquigarrow M''_{\mathcal{P}}$, which is the conclusion.

Case (Call-M): We have

(CALL-M)

$$\langle M_i, V, e_i\rangle \to \langle M_{i+1}, \mathcal{P}_i, v_i\rangle \qquad i \in 0..n \qquad v_0 = \mathsf{new}\ A$$

$$(\mathsf{def}\ \ell\ A.m(\ldots) = \ldots) \notin M_{n+1}$$

$$(\mathsf{def}\ \ell'\ A.\mathsf{method\_missing}(x_1, \ldots, x_{n+1}) = e) \in M_{n+1} \qquad s = \mathit{unparse}(m)$$

$$m \neq \mathsf{method\_missing} \qquad V' = [\mathsf{self} \mapsto v_0, x_1 \mapsto s, x_2 \mapsto v_1, \ldots, x_{n+1} \mapsto v_n]$$

$$\langle M_{n+1}, V', e\rangle \to \langle M', \mathcal{P}', v\rangle$$

$$\langle M_0, V, e_0.m(e_1, \ldots, e_n)\rangle \to \langle M', (\bigcup_i \mathcal{P}_i) \cup \mathcal{P}' \cup [\ell' \mapsto s], v\rangle$$

and $V_{\mathcal{P}}|_{dom(V)} = V$ and $\mathcal{P} \vdash M_0 \rightsquigarrow M_{\mathcal{P}}$, where $\mathcal{P} = (\bigcup_i \mathcal{P}_i) \cup \mathcal{P}' \cup [\ell' \mapsto s]$. We also have

(CALL$_{\rightsquigarrow}$)

$$\frac{\mathcal{P} \vdash e_i \rightsquigarrow e'_i \qquad i \in 0..n \qquad m \neq \mathsf{send}}{\mathcal{P} \vdash e_0.m(e_1, \ldots, e_n) \rightsquigarrow e'_0.m(e'_1, \ldots, e'_n)}$$

Then by induction, we have $\langle M_{\mathcal{P}}, V_{\mathcal{P}}, e'_0\rangle \to \langle M^0_{\mathcal{P}}, \cdot, v_0\rangle$ where $\mathcal{P} \vdash M_1 \rightsquigarrow M^0_{\mathcal{P}}$. Continuing this argument for each subsequent e_i, we will have corresponding reductions for the e'_i, eventually leading to a $\langle M^{n+1}_{\mathcal{P}}, \cdot, v_{n+1}\rangle$ such that $\mathcal{P} \vdash M_{n+1} \rightsquigarrow M^{n+1}_{\mathcal{P}}$.

From (CALL-M) above, we see that (def ℓ' A.method_missing$(x_1, \ldots, x_{n+1}) = e$) is the leftmost definition of A.method_missing in $M_{+n}1$. Furthermore, since $\mathcal{P} \vdash M_{n+1} \rightsquigarrow M_{\mathcal{P}}^{n+1}$, there must be a corresponding set of definitions $d_1, \ldots, d_k$ at the corresponding position in $M_{\mathcal{P}}^{n+1}$, where each d_i is the output of (METH-MISSING$_{\rightsquigarrow}$) translating the A.method_missing definition. We also have $s \in \mathcal{P}(\ell')$, and assume without loss of generality that s is the last string in $\mathcal{P}(\ell')$. Then the d_1 from $M_{\mathcal{P}}^{n+1}$ must be

$$(*) \qquad \text{def } \ell' \; A.m(x_2, \ldots, x_n) = \text{let } x_1 = s \text{ in } e'$$

where $\mathcal{P} \vdash e \rightsquigarrow e'$.

Next, we claim there cannot be any definitions of $A.m$ to the left of the definition $(*)$ above. Since $A.m$ is not in M_{n+1} (by the hypotheses of (CALL-M)) and $\mathcal{P} \vdash M_{n+1} \rightsquigarrow M_{\mathcal{P}}^{n+1}$, there cannot be any directly translated definitions of $A.m$ in $M_{\mathcal{P}}^{n+1}$. The only other possibility would be if $A.m$ were added to $M_{\mathcal{P}}^{n+1}$ as a consequence of translating a different definition of A.method_missing. For that to occur to the left of $(*)$, it would have to have come from a definition of A.method_missing that occurred to the left of the definition of A.method_missing in M_{n+1}. But from the hypotheses of (CALL-M) we know the definition of A.method_missing whose translation yielded $(*)$ is the leftmost occurrence, so that is impossible. Thus we see that $(*)$ is the leftmost definition of $A.m$.

Finally, also by induction, since $\mathcal{P} \vdash e \rightsquigarrow e'$ (from the application of (METH-MISSING$_{\rightsquigarrow}$)), we have $\langle M_{\mathcal{P}}^{n+1}, V', e' \rangle \rightarrow \langle M''_{\mathcal{P}}, \cdot, v \rangle$ where $\mathcal{P} \vdash M' \rightsquigarrow M''_{\mathcal{P}}$. Let $V'_{\mathcal{P}} = [\text{self} \mapsto v_0, x_2 \mapsto x_1, \ldots, x_{n+1} \mapsto x_n]$ Using straightforward reasoning about (LET) we can therefore

show we can show $\langle M_{\mathcal{P}}^{n+1},\, V'_{\mathcal{P}}, \mathsf{let}\ x_1 = s\ \mathsf{in}\ e' \rangle \to \langle M''_{\mathcal{P}}, \cdot, v \rangle$. Then putting all the derived reductions together with (Call), and using the fact that $(*)$ is the leftmost definition $\langle M_{\mathcal{P}},\, V_{\mathcal{P}}, e'_0.m(e_1, \ldots, e_n) \rangle \to \langle M''_{\mathcal{P}}, \cdot, v \rangle$ where $\mathcal{P} \vdash M' \rightsquigarrow M''_{\mathcal{P}}$, which is the conclusion we needed to show.

Case (Value), (Var), (Blame), (Unwrap), (Wrap-Error): Trivial.

Case (Def): We have

$$\frac{}{\langle M,\, V, d \rangle \to \langle (d, M), \emptyset, \mathsf{false} \rangle}\ (\textsc{Def})$$

and $V_{\mathcal{P}}|_{dom(V)} = V$ and $\mathcal{P} \vdash M \rightsquigarrow M_{\mathcal{P}}$. There are two cases.

If d is not defining $A.\mathsf{method\_missing}$, then the translation $\mathcal{P} \vdash d \rightsquigarrow d'$ must have been via ($\textsc{Def}_{\rightsquigarrow}$):

$$\frac{\mathcal{P} \vdash e \rightsquigarrow e' \qquad m \neq \mathsf{method\_missing}}{\mathcal{P} \vdash \mathsf{def}\ \ell\ A.m(x_1, \ldots, x_n) = e \rightsquigarrow \mathsf{def}\ \ell\ A.m(x_1, \ldots, x_n) = e'}\ (\textsc{Def}_{\rightsquigarrow})$$

By (Def), we have $\langle M_{\mathcal{P}},\, V_{\mathcal{P}}, d' \rangle \to \langle (d', M_{\mathcal{P}}), \cdot, \mathsf{false} \rangle$ and $\mathcal{P} \vdash (d, M) \rightsquigarrow (d', M_{\mathcal{P}})$ by definition.

Otherwise, d is a definition of $A.\mathsf{method\_missing}$, and our translation was via (Meth-Missing$_{\rightsquigarrow}$):

(Meth-Missing$_{\rightsquigarrow}$)

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

Letting $d'_1, \ldots, d'_k$ be the flattened list of definitions corresponding to e'', by (Def) we have $\langle M_{\mathcal{P}}, V_{\mathcal{P}}, e'' \rangle \rightarrow \langle (d'_k, \ldots, d'_1, M_{\mathcal{P}}), \cdot, \mathsf{false} \rangle$ But then $\mathcal{P} \vdash (d, M) \rightsquigarrow (d'_k, \ldots, d'_1, M_{\mathcal{P}})$, by definition.

Case (Seq), (Eq-T), (Eq-F), (Let), (If-T), (If-F): Induction following the pattern seen above in (Eval), (Send), and (Call-M)

Case (SEval), (*Wrap*)(*Blame*): Induction following the above pattern.

Case (Call): Similar reasoning to (Call-M). Notice that the method $A.m$ invoked in (Call) cannot be $\mathsf{method\_missing}$, by one of the hypotheses of (Call), and hence by (Def$_{\rightsquigarrow}$) it is directly translated to a corresponding definition in the output. □

Lemma 24 *If $\langle \emptyset, \emptyset, e \rangle \rightarrow \langle M, \mathcal{P}, r \rangle$ and e contains no definitions of* $\mathsf{method\_missing}$ *and $e_d = d_1; \ldots; d_n$, i.e., it is a sequence of definitions, and no d_i defines* $\mathsf{method\_missing}$*, then $\langle \emptyset, \emptyset, (e_d; e) \rangle \rightarrow \langle M', \mathcal{P}, r \rangle$.*

Proof: We have $\langle \emptyset, \emptyset, e_d \rangle \rightarrow \langle M'', \emptyset, \mathsf{false} \rangle$, using (Seq) and (Def), for some M''. We claim that $\langle M'', \emptyset, e \rangle \rightarrow \langle M', \mathcal{P}, r \rangle$. This holds because the original reduction of

e starting from the empty method table produced a value. Therefore, any methods e calls are defined before they are used (because there are no calls handled by method_missing), thereby overriding any prior definition in M''. But then by (Seq) (or one of its variants for blame or *error*) we have our conclusion. □

Theorem 25 (Translation Faithfulness) *Suppose* $\langle \emptyset, \emptyset, e \rangle \to \langle M, \mathcal{P}', v \rangle$ *and let* $\mathcal{P}' \subseteq \mathcal{P}$. *Also assume* $\mathcal{P} \vdash e \rightrightarrows e'$. *Then there exist* $M_{\mathcal{P}}, \mathcal{P}''$ *such that* $\langle \emptyset, \emptyset, e' \rangle \to \langle M_{\mathcal{P}}, \mathcal{P}'', v \rangle$, *i.e., both the original and translated program evaluate to the same result.*

Proof: From (Prog$_{\rightsquigarrow}$) we have

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

Thus we have $\mathcal{P} \vdash e \rightsquigarrow e'$. Trivially $\emptyset|_{dom(\emptyset)} = \emptyset$ and $\mathcal{P} \vdash \emptyset \rightsquigarrow \emptyset$. By observation of the translation rules, we can see that e' and e_d contain no definitions of method_missing. Thus by Theorem 23, we have $\langle \emptyset, \emptyset, e' \rangle \to \langle M_{\mathcal{P}}, \mathcal{P}'', v \rangle$ for some $M_{\mathcal{P}}, \mathcal{P}''$. But then by Lemma 24 we have $\langle \emptyset, \emptyset, (e_d; e') \rangle \to \langle M', \mathcal{P}'', v \rangle$. □

C.2.2 Type Soundness

We show soundness of the type system in Figure C.7 using a standard progress-preservation approach. We begin by defining a relationship between the run-time method table and variable store and their static approximations in the type system.

Definition 26 *We write* $\Gamma \sim V$ *if* $dom(\Gamma) = dom(V)$ *and* $\forall x \in dom(\Gamma)\ .\ \emptyset; \Gamma \vdash V(x) : \Gamma(x)$.

Definition 27 *We write* $MT \sim M$ *if* $dom(MT) = \{A.m \mid (\mathsf{def}\ \ell\ A.m(\ldots) = \ldots) \in M\}$ *and* $\forall d = (\mathsf{def}\ \ell\ A.m(x_1, \ldots, x_n) = e) \in M$ *we have* $MT; \emptyset \vdash d : bool$.

In addition to the semantics rules in Figure C.2, we assume that (a) any expression such that a sub-computation reduces to $\mathsf{blame}\ \ell$, itself reduces to $\mathsf{blame}\ \ell$, and (b) any undefined behavior causes the entire computation to reduce to *error*. In the subsequent theorem, r is either a value, $\mathsf{blame}\ \ell$, or *error*. Since the first two forms are typable, the following theorem implies well-typed programs never reduce to *error*.

Lemma 28 *If* $MT; \Gamma \vdash e : \tau$ *and* $\langle M, V, e\rangle \rightarrow \langle M', \mathcal{P}, r\rangle$ *and* $\Gamma \sim V$ *and* $MT \sim M$ *then* $\emptyset; \emptyset \vdash r : \tau$ *and* $MT \sim M'$.

Proof: By induction on the derivation of $\langle M, V, e\rangle \rightarrow \langle M', \mathcal{P}, r\rangle$. We proceed by case analysis on the expression e. Note that semantic rules that work on wrapped values requires that any extra levels of wrapping be removed by (Unwrap), and that $[\![error]\!]_\ell \notin r$, we also must have reduced it to $\mathsf{blame}\ \ell$ by (Wrap-Error) if it occurred.

Case x**:** By assumption, $MT; \Gamma \vdash x : \tau$, and therefore by ($\text{Var}_\tau$), we have $\Gamma(x) = \tau$. Then since we have $\Gamma \sim V$, we have $x \in dom(V)$ and $\emptyset; \Gamma \vdash V(x) : \tau$. Therefore $MT; \Gamma \vdash V(x) : \tau$. And since $x \in dom(V)$, reduction (Var) applies, and therefore $r = V(x)$.

Case $s, \mathsf{true}, \mathsf{false}, \mathsf{new}\ A$**:** Trivial.

Case d**:** The reduction (DEF) applies, so we have $M' = (d, M)$. Also by assumption, $MT \sim M$. But since we also assume $MT; \Gamma \vdash d : \mathit{bool}$, this implies $MT; \emptyset \vdash d : \mathit{bool}$, since Γ is not used in (DEF$_\tau$). Thus, we have $MT \vdash (d, M)$ (notice that the $A.m$ defined by d already has a type in MT; as a side effect, this implies it already has a previous definition in M). The remainder of the conclusion is trivial to show.

Case $e_1; e_2$**:** By assumption, $MT; \Gamma \vdash e_1; e_2 : \tau_2$. Therefore by (SEQ$_\tau$) we have $MT; \Gamma \vdash e_1 : \tau_1$ and $MT; \Gamma \vdash e_2 : \tau_2$. Suppose $\langle M, V, e_1 \rangle \to \langle M', \mathcal{P}_1, r_1 \rangle$. Then by induction, we have $\emptyset; \emptyset \vdash r_1 : \tau_1$ and $MT \sim M_1$. Thus r_1 is not *error*. If it is $\mathsf{blame}\ \ell$, then we are done, since by (BLAME$_\tau$) we have $\emptyset; \emptyset \vdash \mathsf{blame}\ \ell : \tau_2$. Otherwise r_1 must be a value, and we can reduce via $\langle M_1, V, e_2 \rangle \to \langle M_2, \mathcal{P}_2, r_2 \rangle$. Also by induction, we have $\emptyset; \emptyset \vdash r_2 : \tau_2$ and $MT \sim M_2$, so we have shown the conclusion.

Case $\mathsf{safe\_eval}_\ell\ e$**:** By assumption, $MT; \Gamma \vdash \mathsf{safe\_eval}_\ell\ e : \tau$. Then we have $\langle M, V, e \rangle \to \langle M', \mathcal{P}, r \rangle$. By induction, we have $\emptyset; \emptyset \vdash r : \mathit{string}$. Then there are three cases. If r is an unwrapped value, then it must be a string s. Then we must have applied either (SEVAL) or (SEVAL-BLAME-PARSE). The latter case is trivial (since we reduced to $\mathsf{blame}\ \ell$). In the former case, we had $\langle M', V, [\![e']\!]_\ell \rangle \to \langle M'', \mathcal{P}', v \rangle$. But we also have $MT; \Gamma \vdash [\![e']\!]_\ell : \tau$ by (WRAP$_\tau$). So then by induction $\emptyset; \emptyset \vdash v : \tau$ and $MT \sim M''$.

Otherwise, r must be a wrapped value (it cannot be error), in which case we applied (SEVAL-WRAP), (SEVAL-BLAME), or (SEVAL-BLAME-PARSE). The first case follows the reasoning for (SEVAL) above, and the last two cases follow trivially

by ($\textsc{Blame}_\tau$).

Case $[\![e]\!]_\ell$**:** If (Wrap) was applied, then the conclusion is trivial, since M is not changed by reduction, and the resulting value is $[\![r]\!]_\ell$, which has any type by ($\textsc{Wrap}_\tau$). If (Wrap-Define) was applied, the result holds by ($\textsc{Blame}_\tau$). If (Unwrap) was applied, then the result is trivial by ($\textsc{Wrap}_\tau$). The only other possibility is (Wrap-Error), in which case the result is also trivial by ($\textsc{Wrap}_\tau$).

Case $e_1 \equiv e_2$**:** Similar to sequencing case.

Case $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$**:** By assumption, $MT; \Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2$. Then by ($\textsc{Let}_\tau$) we have $MT; \Gamma \vdash e_1 : \tau_1$ and $MT; x : \tau_1, \Gamma \vdash e_2 : \tau_2$. Suppose $\langle M, V, e_1\rangle \to \langle M_1, \mathcal{P}_1, r_1\rangle$. By induction, we have $\emptyset; \emptyset \vdash r_1 : \tau_1$ and $MT \sim M_1$. If r_1 is $\mathsf{blame}\ \ell'$ then we are done, and otherwise r_1 must be a value.

Let $\Gamma' = x : \tau_1, \Gamma$, and let $V' = x : r_1, V$. Since $\Gamma \sim V$ and $\Gamma' \vdash x : \tau_1$, we have $\Gamma' \sim V'$. Thus if we have $\langle M_1, V', e_2\rangle \to \langle M_2, \mathcal{P}_2, r_2\rangle$, we can apply induction to get $\emptyset; \emptyset \vdash r_2 : \tau_2$ and $MT \sim M_2$, which is our conclusion.

Case $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$**:** There are several cases. If e_1 reduces to an unwrapped value, then the proof is by induction, using the assumption that e_1 has type *bool*, and hence must be a boolean. Otherwise, if e_1 reduces to a wrapped value, then we either apply (If-Wrap-T) or (If-Wrap-F), satisfying the conclusion by induction, or we apply (If-Wrap-Blame), satisfying the conclusion by induction and ($\textsc{Blame}_\tau$).

Case $e_0.m(e_1, \ldots, e_n)$**:** By assumption, $MT; \Gamma \vdash e_0.m(e_1, \ldots, e_n) : \tau$. Thus by ($\textsc{Call}_\tau$), we must have $MT; \Gamma \vdash e_i : \tau_i$ for $i \in 0..n$ and $MT(\tau_0.m) = \tau_1 \times \cdots \times \tau_n \to \tau$ and $m \neq \mathsf{method\_missing}$.

Let $M = M_0$. Then $MT \sim M_0$. Let $\langle M_0, V, e_0\rangle \to \langle M_1, \mathcal{P}, r_0\rangle$. Then by induction, we have $\emptyset;\emptyset \vdash r_0 : \tau_0$ and $MT \sim M_1$. If r_0 is $\mathsf{blame}\ \ell'$ then we are done, since by ($\textsc{Blame}_\tau$) we have $\emptyset;\emptyset \vdash r_0 : \tau$. Otherwise we know r_0 is a value, and we can continue reducing $\langle M_1, V, e_1\rangle \to \langle M_2, \mathcal{P}, r_1\rangle$. Iteratively applying the same argument for all e_i, we have $MT \vdash M_{n+1}$ and $\emptyset;\emptyset \vdash r_i : \tau_i$ (unless one of them reduces to $\mathsf{blame}\ \ell'$, in which case we can trivially show the conclusion).

There are several cases, depending on which reduction we applied. Suppose we applied ($\textsc{Call}$). Since $MT(\tau_0) = \tau_1 \times \cdots \tau_n \to \tau$, we have $\tau_0 = A$ for some A. And since $MT \sim M_{n+1}$, there must be some $d = (\mathsf{def}\ \ell\ A.m(x_1, \ldots, x_n) = e) \in M_{n+1}$ such that $MT;\emptyset \vdash d : \mathit{bool}$. Let $V' = [\mathsf{self} \mapsto r_0, x_1 \mapsto r_1, \ldots, x_n \mapsto r_n]$, and let $\Gamma' = (\mathsf{self} \mapsto A, x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n)$. Then we have $\Gamma' \sim V'$. And, since $MT;\emptyset \vdash d : \mathit{bool}$, we must have $MT;\Gamma' \vdash e : \tau$.

Then by ($\textsc{Call}$), we have $\langle M_{n+1}, V', e\rangle \to \langle M', \mathcal{P}', r\rangle$. By induction (using $\Gamma' \sim V'$, $MT \sim M_{n+1}$, and $MT;\Gamma' \vdash e : \tau$), we have $\emptyset;\emptyset \vdash r : \tau$ and $MT \sim M'$, which is the conclusion we wanted to show.

Otherwise, suppose that r_0 is a wrapped value. If we applied ($\textsc{Call-Meth-Blame}$) or ($\textsc{Call-Type-Blame}$), then we can show the conclusion by ($\textsc{Blame}_\tau$). Otherwise, we must have applied ($\textsc{Call-Wrap}$), and we have $r_0 = [\![\mathsf{new}\ B]\!]_{\ell''}$; notice that it is not necessarily the case that $B = A$, because by ($\textsc{Wrap}_\tau$), $[\![\mathsf{new}\ B]\!]_{\ell''}$ may have any type. However, by ($\textsc{Call-Wrap}$) there must be some $d = (\mathsf{def}\ \ell\ B.m(x_1, \ldots, x_n) = e) \in M$ such that $MT;\emptyset \vdash d : \mathit{bool}$. Let $V' = [\mathsf{self} \mapsto r_0, x_1 \mapsto [\![r_1]\!]_{\ell''}, \ldots, x_n \mapsto [\![r_n]\!]_{\ell''}]$. Then since $MT \sim M_{n+1}$, there must be a Γ' and τ' such that $\mathit{dom}(\Gamma') = \mathit{dom}(V')$ and $MT;\Gamma' \vdash e : \tau'$. But then since all values in V' are wrapped, by

($\textsc{Wrap}_\tau$) we have $\Gamma' \sim V'$.

Then by ($\textsc{Call-Wrap}$), we have $\langle M_{n+1}, V', e\rangle \to \langle M', \mathcal{P}', r\rangle$. By induction, as above, we have $\emptyset;\emptyset \vdash r : \tau'$ and $MT \sim M'$. Then by ($\textsc{Wrap}_\tau$), we also have $\emptyset;\emptyset \vdash [\![r]\!]_{\ell''} : \tau$, showing the conclusion.

Notice that reduction via ($\textsc{Call-M}$) is impossible, because our type system does to allow calls to undefined methods, even if a definition of $\mathsf{method\_missing}$ is present.

Case $\mathsf{eval}_\ell\ e$**:** Impossible, because we assume $MT;\Gamma \vdash \mathsf{eval}_\ell\ e : \tau$, and there are no type rules that assign a type to $\mathsf{eval}_\ell\ e$.

Case $e_0.\mathsf{send}_\ell(e_1, \ldots, e_n)$**:** Impossible, as above.

Case $\mathsf{blame}\ \ell$**:** Trivial, by ($\textsc{Blame}_\tau$). □

Lemma 29 *If $MT \vdash e$ and $\langle M, \emptyset, e\rangle \to \langle M', \mathcal{P}, r\rangle$ and $MT \sim M$ then there exists τ such that $\emptyset;\emptyset \vdash r : \tau$.*

Proof: By induction on the derivation of $MT \vdash e$. There are two cases.

Case ($\mathbf{Prog}_\tau$)**:** By ($\textsc{Prog}_\tau$), we have $MT \vdash d; MT'$ and $MT' \vdash e$. Then by ($\textsc{Def}'_\tau$), we have $d = (\mathsf{def}\ \ell\ A.m(x_1, \ldots, x_n) = \mathsf{blame}\ \ell)$ and $\sigma = \tau_1 \times \cdots \tau_n \to \tau$ and $A.m \in dom(MT) \Rightarrow MT(A.m) = \sigma$ and $MT' = (A.m \mapsto \sigma), MT$.

Furthermore, by ($\textsc{Def}$) we have $\langle M, \emptyset, d\rangle \to \langle (d, M), \emptyset, \mathsf{false}\rangle$, and by assumption we have $MT \sim M$. We need to show $MT' \sim (d, M)$. First, observe we have

$$\begin{aligned}
dom(MT') &= \{A.m\} \cup dom(MT) && \text{(def of } MT'\text{)}\\
&= \{A.m\} \cup \{B.m \mid (\mathsf{def}\ \ell\ B.m(\ldots) = \ldots) \in M\} && (MT \sim M)\\
&= \{B.m \mid (\mathsf{def}\ \ell\ A.m(\ldots) = \ldots) \in (d, M)\} && \text{(def of } d\text{)}
\end{aligned}$$

Second, we need to show

$$\forall d' = (\mathsf{def}\ \ell\ A.m(\ldots) = \ldots) \in (d, M) \text{ we have } MT'; \emptyset \vdash d' : \mathit{bool}$$

Clearly this holds for all $d' \neq d$ since $MT \sim M$ and since MT and MT' agree on the signatures of all common methods. And for $d' = d$, by ($\textsc{Def}_\tau$) and ($\textsc{Blame}_\tau$) we have $MT'; \emptyset \vdash d : \mathit{bool}$.

Now since $MT' \sim (d, M)$, let $\langle (d, M), \emptyset, e\rangle \rightarrow \langle M', \mathcal{P}, r\rangle$. By induction, there exists τ such that $\emptyset; \emptyset \vdash r : \tau$.

Case ($\textbf{Prog-Expr}_\tau$): By ($\textsc{Prog-Expr}_\tau$), we have $MT; \emptyset \vdash e : \tau$, and we have assumption $MT \sim M$. Let $\langle M, \emptyset, e\rangle \rightarrow \langle M', \mathcal{P}, r\rangle$. Then since $\emptyset \sim \emptyset$, by Lemma 28, we have $\emptyset; \emptyset \vdash r : \tau$. □

Theorem 30 (Type Soundness) *If $\emptyset \vdash e$ and $\langle \emptyset, \emptyset, e\rangle \rightarrow \langle M, \mathcal{P}, r\rangle$, then r is either a value of* blame ℓ *(i.e., $r \neq$ error).*

Proof: Since $\emptyset \sim \emptyset$, we can apply Lemma 29 to show there exists τ such that $\emptyset; \emptyset \vdash r : \tau$. Therefore r is either a value or has the form blame ℓ. □

Bibliography

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

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

[3] O. Agesen and U. Hölzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *OOPSLA*, pages 91–107, 1995.

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

[5] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1988.

[6] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft Typing with Conditional Types. In *POPL*, pages 163–173, 1994.

[7] J. D. An, A. Chaudhuri, and J. S. Foster. Static Typing for Ruby on Rails. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, Auckland, New Zealand, Nov. 2009. To appear.

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

[9] C. Anderson, P. Giannini, and S. Drossopoulou. Towards Type Inference for JavaScript. In *ECOOP*, pages 428–452, 2005.

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

[11] L. Bak, G. Bracha, S. Grarup, R. Griesemer, D. Griswold, and U. Holzle. Mixins in Strongtalk. *Inheritance Workshop at ECOOP*, 2002.

[12] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2003.

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

[14] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP*, pages 303–311, 1990.

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

[16] K. B. Bruce, A. Schuett, and R. van Gent. Polytoil: A type-safe polymorphic object-oriented language. In W. G. Olthoff, editor, *ECOOP*, pages 27–51, 1995.

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

[18] L. Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 431–507. Springer-Verlag, New York, NY, USA, 1991.

[19] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985.

[20] R. Cartwright and M. Fagan. Soft typing. In *PLDI*, pages 278–292, 1991.

[21] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *SAS*, pages 1–18, 2003.

[22] R. Chugh, J. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, June 2009. To appear.

[23] O. Danvy. Functional Unparsing. Technical Report RS-98-12, BRICS, Department of Computer Science, University of Aarhus, May 1998.

[24] R. Davies and F. Pfenning. Intersection Types and Computational Effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, Montreal, Canada, Sept. 2000. To appear.

[25] A. Demaille, R. Levillain, and B. Sigoure. Tweast: a simple and effective technique to implement concrete-syntax ast rewriting using partial parsing. In S. Y. Shin and S. Ossowski, editors, *SAC*, pages 1924–1929. ACM, 2009.

[26] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings of the tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–184, Oct. 1995.

[27] R. B. Findler and M. Blume. Contracts as pairs of projections. In *FLOPS*, volume 3945, pages 226–241, Fuji Susono, JAPAN, Apr. 2006. Springer-Verlag.

[28] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, pages 48–59, 2002.

[29] R. B. Findler, M. Flatt, and M. Felleisen. Semantic casts: Contracts and structural subtyping in a nominal world. In *ECOOP*, pages 365–389, 2004.

[30] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching Bugs in the Web of Program Invariants. In *PLDI*, pages 23–32, 1996.

[31] D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly Media, Inc, 2008.

[32] C. Gould, Z. Su, and P. Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. In *ICSE*, pages 645–654, 2004.

[33] J. O. Graver and R. E. Johnson. A type system for Smalltalk. In *PLDI*, pages 136–150, 1990.

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

[35] L. T. Hansen. Evolutionary Programming and Gradual Typing in ECMAScript 4 (Tutorial), Nov. 2007.

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

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

[38] A. Igarashi and H. Nagira. Union types for object-oriented programming. In *SAC*, pages 1435 – 1441, 2006.

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

[40] T. Jim. Rank 2 Type Systems and Recursive Definitions. Technical Report MIT/LCS/TM-531, Laboratory for Computer Science, Massachusetts Institute of Technology, Nov. 1995.

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

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

[43] G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences*, pages 22–39, London, UK, 1987. Springer-Verlag.

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

[45] X. Leroy. The Objective Caml system, Aug. 2004.

[46] B. Livshits, J. Whaley, and M. S. Lam. Reflection Analysis for Java. In *ASPLS*, 2005.

[47] R. A. MacLachlan. The python compiler for cmu common lisp. In *ACM conference on LISP and functional programming*, pages 235–246, New York, NY, USA, 1992.

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

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

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

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

[52] NIST. The Economic Impacts of Inadequate Infrastructure for Software Testing. NIST Planning Report 02-3, May 2002. `http://www.nist.gov/director/prog-ofc/report02-3.pdf`.

[53] D. North. Behavior Modification. *Better Software Magazine*, Mar. 2006.

[54] E. Onzon. `dypgen` User's Manual, Jan. 2008.

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

[56] B. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, CMU, 1991.

[57] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

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

[59] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.

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

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

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

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

[64] Steel Bank Common Lisp, 2008. `http://www.sbcl.org/`.

[65] J. Siek and W. Taha. Gradual typing for objects. In *ECOOP*, pages 2–27, 2007.

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

[67] S. A. Spoon. *Demand-driven type inference with subgoal pruning*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2005. Director-Olin Shivers.

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

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

[70] Strongtalk, 2008. `http://www.strongtalk.org/`.

[71] S. Thatte. Quasi-static typing. In *POPL*, pages 367–381, 1990.

[72] P. Thiemann. Towards partial evaluation of full scheme. In *Reflection 96*, pages 95–106, 1996.

[73] P. Thiemann. Towards a type system for analyzing javascript programs. In *ESOP*, pages 408–422, 2005.

[74] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2nd edition, 2004.

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

[76] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *OOPSLA*, pages 964–974, 2006.

[77] S. Tobin-Hochstadt and M. Felleisen. The Design and Implementation of Typed Scheme. In *POPL*, pages 395–406, 2008.

[78] M. Tomita. *Generalized LR Parsing*. Springer, 1985.

[79] G. van Rossum and J. Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2006.

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

[81] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly & Associates, 3rd edition edition, July 2000.

[82] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1–3):111–156, 1999.

[83] D. A. Wheeler. Sloccount, 2008. `http://www.dwheeler.com/sloccount/`.

[84] Wikipedia. Duck typing, 2009. `http://en.wikipedia.org/wiki/Duck_typing`.

[85] A. Wright and R. Cartwright. A practical soft type system for scheme. *ACM TOPLAS*, 19(1):87–152, 1997.

[86] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.

[87] R. Wuyts. RoelTyper, May 2007. `http://decomp.ulb.ac.be/roelwuyts/smalltalk/roeltyper/`.

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

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