

Static Type Inference for Ruby

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

University of Maryland, College Park
{furr,davidan,jfoster,mwh}@cs.umd.edu

Abstract

Many general-purpose, object-oriented scripting languages are dynamically typed, to keep the language flexible and not reject any programs unnecessarily. However, dynamic typing loses many of the benefits of static typing, including early error detection and the useful documentation provided by type annotations. We have been developing Diamondback Ruby (DRuby), a tool that aims to integrate static typing into Ruby, a popular object-oriented scripting language. DRuby makes three main contributions toward this aim. First, we developed a new GLR parser for Ruby that cleanly separates the core Ruby grammar from a set of disambiguation rules, making the parser easy to extend. Second, we developed the Ruby Intermediate Language (RIL), a small, simple subset of Ruby to which we can translate the entire source language. RIL makes implementing Ruby analyses much easier than working with the complex surface syntax. Third, DRuby includes a type annotation language and type inference system that has important features that allow it to accurately type Ruby programs. We applied DRuby to a suite of small benchmarks, and found that most of our benchmarks are statically typeable. We believe that DRuby makes a major step forward toward the goal of bringing the benefits of static typing to Ruby and other object-oriented scripting languages.

1. Introduction

Dynamic type systems are popular in general-purpose, object-oriented scripting languages like Ruby, Python and Perl. Dynamic typing is appealing because it ensures that no correct program execution is stopped prematurely—only programs about to “go wrong” at run-time are rejected.

However, this flexibility comes at a price. Programming mistakes that would be caught by static typing, e.g., calling a method with the wrong argument types, remain latent until

run time. Such errors can be painful to track down, especially in larger programs. Moreover, with pure dynamic typing, programmers lose the concise, automatically-checked documentation provided by type annotations. For example, the Ruby standard library includes textual descriptions of types, but they are not used by the Ruby interpreter in any way.

We have been developing Diamondback Ruby¹ (DRuby), a tool that integrates static typing into Ruby. Our ultimate 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. DRuby is focused on Ruby, but we expect the advances we make to apply to many other scripting languages as well. Our vision of DRuby was inspired by an exciting body of recent work on mixing static and dynamic typing, including ideas such as soft typing, the dynamic type, quasi-static typing, and gradual typing (see Section 6).

As we started developing DRuby, however, we discovered that applying these ideas to a language as large and complex as Ruby was a major challenge. In this paper, we present three contributions that are a major step forward towards our vision: The development of a clean generalized LR (GLR) parser for Ruby; the Ruby Intermediate Language (RIL), a simplified representation for Ruby programs that is far easier to analyze than the surface syntax; and a static type annotation and inference system that can type check all of Ruby except its metaprogramming and reflective facilities.

The first contribution, a new parser for Ruby, arises because like many scripting languages, Ruby tries to “feel natural to programmers” (Stewart 2001) by providing a very forgiving, almost ambiguous syntax. While we could have tried to reuse the Ruby interpreter’s parser, it is hard to modify, with extremely complex parsing actions that resolve ambiguities in subtle ways. Instead, we developed a GLR parser for Ruby and extended it to support type annotations. Our parser specification cleanly separates the core Ruby language productions from the disambiguation rules, which makes the grammar easier to understand and the parser much easier to modify. (Section 2.2)

[Copyright notice will appear here once ‘preprint’ option is removed.]

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

Even with a clean parser, Ruby is still a dauntingly large language, with many constructs and complex control flow. Our second contribution is RIL, a simplified subset of Ruby designed to make analysis more tractable, in the style of CIL (Necula et al. 2002). RIL includes only side effect-free expressions and sequences of primitive statements, each of which includes at most one assignment and/or method call. DRuby includes a phase that translates parsed Ruby source code into RIL, simplifying the subsequent type inference process. We believe that RIL will also prove useful for other Ruby analysis and transformation tools. (Section 2.3)

Our parser and translator to RIL enabled our final contribution, a static type annotation and inference system for Ruby. DRuby’s type system has three main elements:

- First, DRuby defines an expressive type language that includes features we found necessary to precisely type Ruby code: union and intersection types (Pierce 1991), object types (to complement traditional class name-based types) (Abadi and Cardelli 1996), a self type, parametric polymorphism (Pierce 2002), tuple types for heterogeneous arrays, and optional and variable arguments in method types.
- Second, DRuby adds a surface type annotation syntax. Annotations are required to give types to the core of the Ruby standard library, since it is written in C rather than Ruby, and they are also useful for providing types to Ruby library code that uses hard-to-analyze features. (Section 3)
- Third, DRuby performs type inference 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. Because Ruby is such a flexible language, type inference is rather challenging, and DRuby makes three assumptions for the sake of practicality: DRuby assumes classes and methods are always defined before use; DRuby fully trusts type annotations, and does not check the bodies of annotated methods; and DRuby ignores metaprogramming constructs such as `eval`. However, modulo these assumptions, DRuby is able to accurately model many other common programming idioms we encountered. (Section 4)

We have applied DRuby to a suite of small benchmark programs (ranging from 100–900 lines of code) that are representative of the kinds of scripts programmers write with Ruby. Out of 8 benchmarks, DRuby reports no warnings for 5, i.e., they are statically type safe according to our type system. All of the warnings for the other 3 benchmarks were false positives, i.e., warnings of type errors that cannot actually occur at run time. Two of the remaining benchmarks had one warning each, and the last benchmark has 35 warnings (all due to one problem). These results show that DRuby is promising, and that in fact much of Ruby is amenable to static typing. (Section 5)

```

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 = [1, 2, 3] # array literal
29 j = i.collect { |x| x + 1 } # j is [2, 3, 4]
30
31 module My_enumerable # module creation
32   def leq(x)
33     (self ⇔ x) ≤ 0 # note method "⇔" does not exist here
34   end
35 end
36
37 class Container
38   include My_enumerable # mix in module
39   def ⇔(other) # define required method
40     @x ⇔ other.get
41   end
42 end
43
44 f ⇔ f # returns 0

```

Figure 1. Sample Ruby code

We believe DRuby is a major step toward integrating static and dynamic typing into Ruby in particular, and object-oriented scripting languages in general, and that DRuby lays the foundation for much interesting future work.

2. The Ruby Intermediate Language

This section presents some background on Ruby and then describes our parser and RIL.

2.1 Ruby Background

Since readers may not be familiar with Ruby, we first introduce some of its 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 variable 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`.²

Figure 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 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 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 array literals (line 28) and hash literals (not shown). Ruby also supports higher-order functions, called *code blocks*. Line 29 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, \dots, 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 1), then the declared class inherits from `Object`. Ruby also includes *modules* (a.k.a. *mixins*) (Bracha and Cook 1990) for supporting multiple inheritance. For example, lines 31–35 define a module `My_enumerable`, which defines an `leq` method in terms of another method \Leftrightarrow that implements three-way comparison. On lines 37–42, we mix in the `My_enumerable` module (line 38) via `include`, and then define the needed \Leftrightarrow method (lines 39–41). From line 44 onward, we can invoke `Container.leq`. Note also that on line 37 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. Such modifications are difficult to model statically, and our type system may be unsound in the presence of some of these features (end of Section 3).

Further introduction to Ruby is available elsewhere (Thomas et al. 2004; Flanagan and Matsumoto 2008).

2.2 Parsing Ruby

The first step toward typing Ruby, or indeed performing any similar static analysis, is parsing Ruby code. One option would be to use the parser built in to the Ruby interpreter. Unfortunately, that parser is written in C, and is tightly integrated with the rest of the interpreter. Another option would be to start with the front end used by JRuby, a Ruby implementation written in Java.³ However, this front-end was not very robust when we started our work, and we felt that it was also not very declarative, and was therefore hard to understand. Indeed, Ruby lacks a clean specification of its grammar, including how it prefers one form to another when the syntax is ambiguous. For example, the pseudo-BNF formulation of the Ruby grammar from the on-line Ruby 1.4.6 language manual⁴ essentially ignores the many exceptional cases.

Thus, we opted to write a Ruby parser from scratch with two goals. First, we wanted the grammar specification to be as understandable as possible while still correctly parsing all the potentially-ambiguous cases. Second, we wanted the parser to be able produce an intermediate representation in OCaml (Leroy 2004), a language we feel is extremely convenient for writing static analyzers, due to its data type language and pattern matching features. Meeting the first goal turned out to be far harder than we anticipated when we started this project, although we were ultimately able to develop a very robust parser (and intermediate representation, described below) that we hope others will find useful.

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 (Venners 2003). This philosophy extends not only to Ruby’s language con-

² As an aside, there are a few oddball variables like `$!` that are prefixed as a global but are in fact local.

³ <http://jruby.codehaus.org/>

⁴ <http://docs.huihoo.com/ruby/ruby-man-1.4/yacc.html>

structs, but also to the surface syntax. As a result, Ruby is highly ambiguous from an LL/LR parsing standpoint.

For example, consider the following. In Ruby a top-level method call may omit parentheses, and method names may be postfixed by “=”. It would be natural to specify the grammar for method calls using the following BNF productions:

```

methcall ::= expr '(' expr (',' expr)* ')' block?
          | expr expr '(',' expr)* block?
block    ::= '{' expr '}'
expr     ::= expr = expr | id | literal | methcall
literal  ::= num | true | '{'expr (','expr)*'}' | ...

```

(Here, the last literal production defines the syntax for a hash literal.) However, these productions would produce several parse trees for the expression `x=y{z}`:

- (1) `x = (y() {z})` – (assign to x; z a codeblock)
- (2) `x = (y({z}))` – (assign to x; z a hash literal)
- (3) `self .x=(y({z}))` – (call “x=” method; z a hash)
- (4) `self .x=(y() {z})` – (call “x=” method; z a block for y)
- (5) `self .x=(y) {z}` – (call “x=” method; z a block for self)

It turns out that through a combination of subtle disambiguation rules, Ruby will produce parse (4). However, encoding these rules into a series of grammar productions is extremely messy. Generic productions like `expr` have to be broken apart to account for each special case, which is both difficult to get correct and a code maintenance nightmare. We could use complex parser actions to account for these cases (as the Ruby interpreter does), but again, this yields a parser than is hard to maintain.

Instead, we built our parser using the `dypgen` generalized LR (GLR) parser generator (Onzon 2008), which supports ambiguous grammars. Our parser uses productions similar to the grammar given above, and without further change would produce all five trees. To indicate which tree we 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. The production `command`, defined on line 5, handles top-level method calls in which the parentheses have been omitted, as in the example above. The action for this rule calls two helper functions, `well_formed_command` and `methodcall`, to prune ill-formed sub-trees. The `well_formed_command` function is defined in the preamble of the parser file, an excerpt from which is given on lines 1–2. This function checks whether a hash literal has been passed as an argument to the parsed method and, if so, it raises the `Dyp.Giveup` exception to tell `dypgen` to abandon this parse tree. This rule has the effect of marking parse trees (2) and (3) in the above example invalid, since these each have a hash literal as an argument.

By cleanly separating the grammar’s productions from the disambiguation rules, 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

```

1 let well_formed_command dyp m args = match m,args with
2 | -, [E.Hash _] -> raise Dyp.Giveup
3 ...
4 %%
5 command:
6 | command_name[m] call_args[args]
7   { well_formed_command dyp m args;
8     methodcall m args None (pos_of m)}

```

Figure 2. Example GLR Code

only the disambiguation rules and were able to leave the productions alone. We believe that our GLR specification comes fairly close to serving as a standalone Ruby grammar: the production rules are quite similar to the “pseudo-BNF” used now, while the disambiguation rules describe the exceptional cases. Our parser currently consists of 59 productions and 260 lines of OCaml for disambiguation and helper functions.

There are still a few places our parser disagrees with the Ruby interpreter. For example, in `x %(y/2)`, the text `%` may be a method name or the start of a string delimited by `%(` and `)`. In other words, this expression may correspond to `x.%(y/2)` or `x("y/2")`. Our parser assumes the latter parse, and if this is incorrect, it can be manually fixed by inserting a space after the `%`. We think these remaining parsing issues are extremely minor, and we leave them to future work.

We ran our parser on 1,239 Ruby files (totalling 163,297 source lines of code) collected from the Ruby standard library and the Ruby interpreter’s internal test suite. We had to manually disambiguate (by adding a space or `()`’s) 10 locations. We also wrote 280 tests cases to ensure specific Ruby constructs were parsed into the correct AST representation.

2.3 An Intermediate Representation

After parsing, we simplify the AST to produce a program in the *Ruby Intermediate Language* (RIL), a subset of Ruby that is smaller and simpler than the full language. There are two main issues RIL seeks to address. First, we wanted to canonicalize the language constructs into a small set of disjoint primitives, so that our analysis would need to handle fewer cases. For example, the following four statements are all equivalent, and are translated into form (1) in RIL:

- (1) `if p then e end` (3) `e if p`
- (2) `unless (not p) then e end` (4) `e unless (not p)`

Second, we wanted to make the control flow in Ruby programs apparent. In Ruby, almost any construct can be nested inside of any other construct, which makes the sequencing of side effects tedious to unravel. For example, in the expression `w = x().y(z())` the evaluation of `x()` occurs first, then `z()`, and then `y()`. Thus, RIL transforms the above to `t1 = x(); t2 = z(); w = t1.y(t2)`; This simplification makes it easier to build a flow-sensitive type inference system (Section 4), or any other analysis of Ruby that needs to know the order of side effects.

$s ::=$	$s_1; s_2$	Sequencing
	if e then s_1 else s_2	Conditional
	$lval = e$	Expression assignment
	$lval = e.new$	Object creation
	$lval = e_0.m(e_1, \dots, e_n)[b]^?$	Method invocation
	$lval = yield(e_1, \dots, e_n)$	Block invocation
	return e	Return
	class $A = s$	Class definition
	inherit A	Inheritance/module mixin
	def $m(p) = e$	Method definition
$e ::=$	nil	Nil
	n	Integers
	id	Identifiers
	$[e_1, \dots, e_n]$	Arrays
$lval ::=$	$x \mid @x \mid A \mid lval, \dots, lval$	
$id ::=$	self $\mid x \mid @x \mid A$	
$b ::=$	$\lambda x_1, \dots, x_n.s$	
$p ::=$	$x_1, \dots, [x_n = e_n, \dots]^?, [*x_m]^?$	
$x \in$	local variable names	
$@x \in$	instance variable names	
$A \in$	constants	
$m \in$	method names	

Figure 3. Ruby Intermediate Language (RIL)

Figure 3 gives a grammar for the key parts of RIL. In the figure, optional elements are enclosed in $[]^?$. The grammar is simplified from the actual implementation to make presenting type inference easier. Section 2.4 describes the differences between this figure and our implementation.

RIL separates statements s , which have side effects, from expressions e , which are side-effect free. Briefly, statements include sequencing, one form of conditional, and assignment operations in which the right-hand side may be an expression, object creation, a method invocation, or a call to yield, which invokes the block passed to the current method. It is an error to call yield when no block is passed. Method invocation optionally takes a code block b . In RIL we use standard λ notation: a code block b is simply a function of some number of arguments x_1, \dots, x_n with body s . Statements also include a return construct that exits the current method.

Somewhat unusually, class definitions and inheritance are statements that can appear anywhere in a program. This is in contrast to, e.g., Java, in which class definitions may occur only at the top level, and inheritance is specified when the class is defined. In our formal syntax, a class is either a class or a module, and inherit A corresponds to either inheriting from class A or mixing in module A . While there are some surface restrictions in Ruby that distinguish classes and modules—classes can only inherit from one other class, and modules cannot be instantiated—we conflate these two

constructs in this paper because their typing rules are the same. In our actual implementation, we distinguish the two in case another client of RIL needs to differentiate them.

Lastly, statements include method definitions. Note that as with class definitions and inheritance, a method definition may occur anywhere in a statement list, e.g., it may occur conditionally depending on how an if statement evaluates. (However, no matter whether or when a method definition is executed, the defined method is always added to the lexically enclosing class.) After the regular parameters in a method definition, a parameter list p may contain zero or more *optional* arguments, which are declared with the form $x_i = e_i$. Such a parameter takes on the value e_i if no corresponding argument is passed in position i . Finally, the parameter list may end with at most one *vararg* parameter, written $*x_m$. If a vararg parameter is present, then any actual parameters passed in positions m or higher are gathered into an array that is passed as argument x_m .

RIL expressions e consist of literals such as nil and integers n ; identifiers id ; and arrays of expressions. Identifiers id include self, local and instance variables, and *constants* A , which always begin with a capital letter. In Ruby (and RIL), constants can be assigned to exactly once. For example, $A = 1$ creates the constant A , which is read-only from the assignment statement forward. Class names are also constants, but are initialized with class rather than assignment. The value associated with a class name is an instance of the Class class, and class names can appear anywhere an *id* may occur.

Finally, an *lval*, which may appear on the left-hand side of an assignment, is either a local or instance variable name, a constant, or a sequence of *lvals*, which can be used for parallel assignment from an array.

The translation from Ruby to RIL is straightforward. Redundant forms, such as the different conditionals, are translated to the appropriate kind of statement. We hoist side-effecting computations out of complex statements or expressions, introducing fresh temporary local variables as necessary. For consistency, our translation also places a return statement at the end of every possible path through a function. For example, the body of the get function on lines 8–10 of Figure 1 is translated to **return** $@x$. We also translate conditionals with $\&\&$ and $\|\$ into cascaded if statements.

2.4 Additional Language Features

Our implementation of RIL includes several additional constructs that are omitted from Figure 3, including support for all of Ruby’s expression and statement forms. One very useful construct we support is the stringizer $\#{e}$ which, if it appears inside a string, is replaced at execution time by the text returned by $e.to_s$. RIL also includes a full grammar for identifiers, including class variables $@@x$, global variables $\$x$, and super, which can be used to invoke superclass methods. Since classes (and other constants) can be nested, RIL includes scoped naming constructs, as does Ruby. The iden-

```

  typ ::= A<typ1, ...> | obj_typ | typ or typ | t
  obj_typ ::= [m0 : meth_typ0, ...]
  meth_typ ::= (param_typs)[{meth_typ}]? → typ
  param_typs ::= typ1, ..., [?typn, ...]?, [*typ]?
  typ_decl ::= m<t1, ...> : meth_typ

  A ∈ class and module names
  m ∈ method names
  t ∈ type variables

```

Figure 4. Type annotation language

tifier $A :: B$ refers to the constant B nested inside of class A , and $:: B$ refers to the constant at the top level.

Ruby allows per-object methods, called *eigenmethods*, which are part of an object’s *eigenclass* (a.k.a. *virtual class*). For example, suppose x is an instance of A . Writing `def x.m ... end` adds a method to x ’s eigenclass but not to A ; thus no instance of A except x can be used to invoke m . There are several other ways eigenclasses are used in Ruby, and RIL has full support for them.

Finally, in Figure 3 we called out `new` and `inherit` as special constructs, but in RIL as in Ruby these are ordinary method calls. The call to `new` may also pass arguments to the constructor method, which is named `initialize`.

3. Static Types for Ruby

As we discussed earlier, everything in Ruby is an object, and thus the fundamental operation in Ruby is dynamic dispatch. When the Ruby interpreter evaluates $e_0.m(e_1, \dots, e_n)$, it searches for a method m of arity n in object e_0 . If such a method exists it is invoked, and otherwise the interpreter throws a `NoMethodError` exception. Since almost everything in Ruby reduces to dynamic dispatch, `NoMethodError` is the core type error in Ruby. Thus, the goal of our static type system is to detect, prior to running the program, the possibility of such errors. For example, we should accept the program `(String.new).length` and reject the programs `(String.new).length(3)` and `(String.new).foo`, since `Strings` have a `length` method that takes no arguments and do not have a `foo` method.

In this section we discuss the basics of our static type system by example, focusing on the type language’s surface-level syntax, which we use to annotate the standard library. The next section describes how we perform type inference on Ruby programs, and presents details of the internal analysis type language.

Type Annotations for the Standard Library Ruby includes an extensive standard library of useful classes. The core of this library is written in C, and is integrated into Ruby via a foreign function interface. Therefore while instances of classes such as `Fixnum`, `String`, and `Array` look

```

1 class String
2   "+" : (String) → String
3   insert : (Fixnum, String) → String
4   upto : (String) {String → Object} → String
5
6  .chomp : (?String) → String
7   delete : (String, *String) → String
8
9  .include? : Fixnum → Boolean
10  .include? : String → Boolean
11
12  .slice : (Fixnum) → Fixnum
13  .slice : (Range) → String
14  .slice : (Regexp) → String
15  .slice : (String) → String
16  .slice : (Fixnum, Fixnum) → String
17  .slice : (Regexp, Fixnum) → String
18 end
19
20 module Kernel
21   .print : (*[to_s : () → String]) → NilClass
22   .clone : () → self
23 end
24
25 class Array<t>
26   .at : (Fixnum) → t
27
28   .first : () → t
29   .first : (Fixnum) → Array<t>
30
31   .collect<u> : () {t → u} → Array<u>
32   ."+<u> : (Array<u>) → Array<t or u>
33 end

```

Figure 5. Selected type annotations for the standard library

like ordinary objects, they are modeled specially inside the interpreter, and their types cannot be inferred with our analysis. We therefore introduce a new type annotation syntax to allow us to declare the types of these classes.

Figure 4 presents our surface type annotation syntax. In this figure, lists $\langle \dots \rangle$ that are empty may be omitted. A basic Ruby type is simply a class name A , e.g., `Fixnum`, `Object`, etc. We discuss support for instantiated types $A<typ_1, \dots>$, object types, and union types below.

In our implementation, type annotations appear before the corresponding class or method declaration. All annotations appear on a line beginning with `###`, and therefore appear as comments to the standard Ruby interpreter. Since we are using type annotations for methods with no actual Ruby implementation, we currently ignore the body of an annotated method, and simply trust the annotation.

We developed a file `base_types.rb` that includes annotations for the Core Standard library (or simply the Core Library): classes and globals that are pre-loaded by the Ruby interpreter. This file includes types for some or all parts of 37 classes and 5 modules, using 848 lines of type annotations in total for 787 methods. Our implementation analyzes this file before applying type inference to a program.

Figure 5 lists sample annotations from `base_types.rb`. The examples in Figure 5 are copied almost directly from `base_types.rb`—the only differences are the removal of `###` and some minor transformations to make discussion easier. We have also omitted the dummy method bodies, since they are empty and otherwise ignored.

Lines 2–4 illustrate the basic annotation syntax (production `typ_decl` in Figure 4, which ascribes `m` a type `meth_typ`). On line 2, we type the method `+` (non-alphanumeric method names appear in quotes), which concatenates a `String` argument with the current object and returns a new `String`. Similarly, line 3 declares that `insert` takes a `Fixnum` (the index to insert at) and another `String`, and produces a new `String` as a result. Finally, Line 4 types `upto`, which takes a `String` `s` and calls the block on each `String` in the range `self..s` (as determined by the `String.succ` function). The return type of the block is unconstrained, and `upto` itself returns a `String`.

We discuss the remainder of the annotations as we continue our examination of various language features.

Intersection Types Many methods in the standard library have different behaviors depending on the number and types of their arguments. For example, lines 9–10 give a type to the `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 similar inclusion test. The type of `include?` is an example of *intersection type* (Pierce 1991). 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 an instance of a common subclass of both `A` and `B`. In our annotation syntax for methods, the **and** keyword is omitted, and each conjunct of the intersection appears on its own line.

Lines 12–17 show another example of intersection types, this time for the `slice` method, which returns either a character or a substring. 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.⁵ 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. Adding annotation support for intersection types was critical for accurately modeling key parts of the standard library. 111 out of 787 methods (14%) in `base_types.rb` use intersections.

Union Types Because Ruby is dynamically typed, it is easy to freely mix different classes that share common methods. For example, consider the following code:

```
class A def f() end end
class B def f() end end
x = (if ... then A.new else B.new)
x.f
```

Even though we cannot statically decide whether `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`. To support this coding idiom without requiring interface declarations, our type system includes *union types* of the form `t or t'`, where `t` and `t'` are types (which can themselves be unions). 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` would be both an `A` and a `B`, and so it would have *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?` 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.

Object Types While most types refer to particular class names, Ruby code usually only requires that objects have certain methods, and is agnostic with respect to actual class names. For example, line 21 in Figure 5 gives the type of the `print` method, which takes zero or more objects and displays them on standard output. The `print` method only requires that its arguments have a `no-argument to_s` method that produces a `String`.

Here the *object type* $[m_0 : t_0, \dots, m_n : t_n]$ is a structural type (*obj_typ* in Figure 4) that describes an object in which each method `mi` has type `ti`. Object types are not that common in the core library (23 methods or 3% in `base_types.rb` include object types), but they are essential for precisely describing user code. For example, consider the following code snippet:

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

The most precise type for `x` is `[foo : () → t, bar : () → u]` for some `t` and `u`. If we wanted to stick to nominal typing only, we could instead 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`.

The self type Consider the following code snippet:

```
class A; def me() self end end
class B < A; end
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 this method, and the method invocation on the last line returns an instance of `B`, not of `A`. A similar method

⁵ <http://ruby-doc.org/core/classes/String.html#M000858>

that occurs in practice is `clone`, typed on Line 22 of Figure 5. This method, no matter what class it is inherited in, returns an instance of the class of `self`. Self types are de-sugared into a form of parametric polymorphism, described next.

Parametric Polymorphism To give precise types to container classes, we use *parametric polymorphism*, also called *generics* in Java. Lines 25–33 show a segment of the `Array` class, which is parameterized by a *type variable* t , which is the type of the contents of the array. As usual, type variables bound at the top of a class can be used anywhere inside that class. For example, line 26 gives the type of the `at` method, which takes an index and returns the element at that index. Type variables can also be used in intersection types, e.g., the `first` method, typed on lines 28–29, takes either no arguments, in which case it returns the first element of the array, or a value n , in which case it returns the first n elements of the array, which is itself an array of type `Array<t>`.

We also support parametric polymorphism on methods. For example, line 31 types the `collect` method, which for any type u takes a code block from t to u and produces an array of u . Line 32 types the array concatenation method, which takes an array of u 's and (non-destructively) appends it to the end of the current array, producing a new array whose elements are either t 's or u 's.

We also use parametric polymorphism internally for self types by exposing the method receiver in types. For example, we translate the method type for `clone` to the type `<u> : (self : u) → u`, where the receiver object of the method call is bound to the `self` argument. Our surface syntax also allows constraints on polymorphic types, but we have omitted them for simplicity.

Tuple Types The `Array<t>` type describes *homogeneous* arrays, whose elements all consist of the same type. However, Ruby's dynamic typing allows programmers to also create heterogeneous arrays, in which each element may be 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

```
def f() [1, true] end
a, b = f
```

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

Our solution is to introduce a special type `Tuple< t_1, \dots, 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, \dots, t_n >` with the type `Array< t_1 or ... or t_n >`.

Optional Arguments and Varargs Types for optional arguments are prefixed with `?`, as shown on line 6. This line shows the type of the `chomp` function, which removes a `String`—either the argument or the contents of the special global variable `$/` if no argument is supplied—from the end of `self`. `Varargs` parameters are specified with types of the form `* t` , which means zero or more parameters of type t . For example, line 7 gives a type to `delete`, which removes any characters in the intersection of its (one or more) `String` arguments from `self`. Our grammar ensures that no regular types can appear in parameter lists to the right of an optional argument type, and any `varargs` type must be rightmost in the parameter type list.

Types for Variables and Nil An important feature of our type system is our modeling of variables and `nil`. We track 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, it allows us to type check the invocation `b.length` on line 5 of Figure 1, since the last value assigned to `b` was a `String`, even though after line 2, `b` is a `Fixnum`.

We have to be careful about tracking local variables that may be captured by blocks. For example, in the code `"x = 1; foo() { |y| 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, and global variables flow-insensitively.

Finally, like Java, 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.

Unsupported features Currently, there are standard library methods with types that we 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 .

Additionally, some features of Ruby are problematic to model in a static type system. In particular, Ruby allows classes and methods to be changed arbitrarily at run time (e.g., added via class reopening or removed via the special `undef` statement). To produce a tool that is practical, we assume that all classes and methods defined somewhere in the code base are available at all times. This may cause us to miss some type errors, and we leave weakening this assumption to future work.

Finally, Ruby includes an `eval` method that interprets an arbitrary string as a Ruby program and executes it. Com-

$$\begin{aligned}
\tau &::= \alpha \mid \tau \cup \tau \mid \tau \langle \vec{\tau} \rangle \mid (\tau \times \dots \times \tau) \\
&\quad \mid \forall \vec{\alpha}. \tau \mid \text{Struct} \mid \text{Nom} \\
\text{Struct} &::= [\vec{F}; \vec{M}] \\
\text{Nom} &::= \{\text{kind}:\tau; \text{par}:\vec{\tau}; \text{flds}:\vec{F}; \text{mths}:\vec{M}\} \\
F &::= @x : \tau \\
M &::= m : \text{mt} \\
\text{mt} &::= (\tau_0 \times \dots \times [?\tau_n \times \dots]^? \times [* \tau_m]^? \times [bt]^?) \rightarrow \tau \\
&\quad \mid \text{mt} \cap \dots \cap \text{mt} \mid \forall \vec{\alpha}. \text{mt} \\
\text{bt} &::= \beta \mid (\tau \times \dots \times \tau) \rightarrow \tau \\
\\
\Gamma &::= \emptyset \mid \Gamma, x \mapsto \tau \mid \Gamma, \text{self} \mapsto \tau \\
\Omega &::= [\text{cls}:\text{Nom}; \text{blk}:\text{bt}; \text{ret}:\tau] \\
C &::= C \cup C \mid \tau \leq \tau \mid \text{bt} \leq \text{bt} \mid \text{mt} \leq \text{mt} \mid \tau \Rightarrow \tau \\
&\quad \mid \tau \in \tau.\text{par} \mid (m:\text{mt}) \in \tau.\text{mths} \mid A:\tau \mid x:\tau
\end{aligned}$$

Figure 6. Types, contexts, and environments for inference

combined with the ability to change classes and methods at run-time, this gives programmers a very powerful tool for metaprogramming. For example, one of our benchmarks includes the following code:

```

ATTRIBUTES.each do |attr|
  code = "def #{attr}(&blk) ... end"
  eval code
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. However, no reasonable static type system will be able to analyze this code. When we encounter this code, we simply check that each is passed arguments of the correct type and ignore its evaluation behavior entirely. Ultimately, we think that approaches that mix analysis at compile time with analysis at run time (as done by RPython (Ancona et al. 2007)) may be the best option.

4. Type Inference for Ruby

Our type inference system is constructed as a *constraint-based analysis*. We first traverse the RIL representation of a program, visiting each statement once and generating constraints that capture dependencies between types. For example, if we see `x.m()`, we require that `x` have a method `m()`. We then solve the constraints to find a valid typing for the program, or report a type error if we find an inconsistency in the constraints.

4.1 Types for Inference

To perform type inference, we need a slightly richer type grammar, shown in Figure 6, than the annotation language

available in our surface syntax. Here types τ include type variables α , union types, and type instantiations $\tau \langle \vec{\tau} \rangle$, as before. We also include *tuple types* ($\tau_1 \times \dots \times \tau_n$), which model heterogeneous arrays with element types τ_1 through τ_n , from left to right. We write polymorphic types as $\forall \vec{\alpha}. \tau$ where $\vec{\alpha}$ is the set of quantified variables and τ is the base type. Structural object types (previously *obj_typ*) are defined by the non-terminal *Struct*, and now include types for fields as well as methods.

During inference, we use *nominal types* *Nom* to describe instances that come from a known type, e.g., `String`. Nominal types contain four fields: *kind*, which indicates what class the nominal type is an instance of; *par*, the parents inherited by the type; and *flds* and *mths*, the fields and methods of the type. We write *Nom.par* for the *par* field of *Nom*, and similarly for the other fields. It is easiest to explain nominal types by example. Consider the following code:

```

class A < B
  include C
  def f() @x = 3 end
end

```

In the nominal type Nom_A of `A`, *kind* is Nom_{Class} , the type corresponding to `Class`, because `A` itself is an instance of `Class`. The field *par* includes Nom_B and Nom_C , the types of `B` and `C`. The field *flds* is empty, and the field *mths* includes a type for method `f`. On the other hand, in the nominal type of `A.new`, *kind* is Nom_A , since this object is an instance of `A`; *par* and *mths* are empty; and *flds* contains a type for `@x`. This structure makes the instance of relationship clear in the type system, which helps support eigenclass-related operations (omitted for simplicity).

Continuing with the type grammar, basic method types *mt* have the form $(\tau_0 \times \dots \times [?\tau_n \times \dots]^? \times [* \tau_m]^? \times [bt]^?) \rightarrow \tau$, where τ_0 is the type of `self`; τ_n through τ_{m-1} are optional argument types; τ_m is the *varargs* parameter type; and *bt* is the block argument type. All but the regular parameters may be omitted. Method types also can be formed via intersections or generalizations of method types. Block types *bt* are similar to method types except they take only regular parameters, and their first argument is not `self`. We also include a variable β that ranges over block types.

4.2 Type Inference Overview

Type inference is specified as a pair of judgments. The judgment $C; \Gamma; \Omega \vdash_e e : \tau$ means that under constraints C , environment Γ , and context Ω , expression e has type τ . Similarly, $C; \Gamma; \Omega \vdash_s s; \Gamma'$, means that under C , Γ , and Ω , statement s produces an *output environment* Γ' . Here Γ' types local variables in scope after s , and may differ from Γ . In particular, it may include types for newly create locals, and it may differ in types of locals changed by s . Having an output type environment is how we achieve flow-sensitivity.

Environments, contexts, and constraints are defined in the lower part of Figure 6. Environments Γ track bindings for

$\frac{}{(NIL)} \frac{}{C; \Gamma; \Omega \vdash_e \text{nil} : Nom_{\text{nil}}}$	$\frac{}{(INT)} \frac{}{C; \Gamma; \Omega \vdash_e n : Nom_{\text{Fixnum}}}$
$\frac{}{(LOCAL/SELF)} \frac{id \in \text{dom}(\Gamma) \quad id \text{ not captured}}{C; \Gamma; \Omega \vdash_e id : \Gamma(id)}$	$\frac{}{(LOCAL-CAP)} \frac{C \vdash x : \alpha \quad \alpha \text{ fresh} \quad x \text{ may be captured}}{C; \Gamma; \Omega \vdash_e x : \alpha}$
$\frac{}{(FIELD)} \frac{C \vdash \Gamma(\text{self}) \leq [\text{@}x : \alpha] \quad \alpha \text{ fresh}}{C; \Gamma; \Omega \vdash_e \text{@}x : \alpha}$	$\frac{}{(CONST)} \frac{C \vdash A : \alpha \quad \alpha \text{ fresh}}{C; \Gamma; \Omega \vdash_e A : \alpha}$
$\frac{}{(TUPLE)} \frac{C; \Gamma; \Omega \vdash_e e_i : \tau_i \quad i \in 1..n}{C; \Gamma; \Omega \vdash_e [e_1, \dots, e_n] : (\tau_1 \times \dots \times \tau_n)}$	
$\frac{}{(BLOCK)} \frac{C; \Gamma[x_i \mapsto \alpha_i]; \Omega[\text{ret} : \alpha_{n+1}] \vdash s; \Gamma' \quad \alpha_1, \dots, \alpha_{n+1} \text{ fresh} \quad i \in 1..n}{C; \Gamma; \Omega \vdash_e \lambda x_1, \dots, x_n. s : (\alpha_1 \times \dots \times \alpha_n) \rightarrow \alpha_{n+1}}$	

Figure 7. Type judgments for expressions

local variables x and `self`. A context Ω stores the nominal type of the current class; the type of the block passed to the current method; and the return type of the current method or block. We write $\Omega.\text{cls}$ for the type of the current class, and similarly for `blk` and `ret`. Constraints C are sets of several kinds of primitive constraints. Subtyping constraints $\tau \leq \tau'$, $bt \leq bt'$, and $mt \leq mt'$ are standard (Mitchell 1991). *Instance constraints* $\tau \Rightarrow \tau'$ are generated for calls to $e.\text{new}$, where τ is the type of e and τ' is the type of $e.\text{new}$; their use is described below. The constraint $\tau \in \tau'.\text{par}$ requires that τ appear in the `par` field of the nominal type described by τ' , and analogously for $(m : mt) \in \tau'.\text{mths}$. The last two constraints, $A : \tau$ and $x : \tau$, require that the specified constant or local have type τ . The latter constraint is only used for constants that are captured by blocks, and hence are treated flow-insensitively.

Given a program s , type inference aims to prove a judgment $C; \Gamma_{\text{init}}; \Omega_{\text{init}} \vdash s; \Gamma'$ for some Γ' , where Γ_{init} binds `self` to an instance of the nominal type for `Object`, $\Omega_{\text{init}} = [\text{cls} : Nom_{\text{Object}}; \text{blk} : \emptyset; \text{ret} : \alpha]$ for fresh α , and C contains (at least) constraints $A : Nom_A$ for all classes A that are provided with type annotations. Here Nom_A is derived by translating the type annotations into the type grammar in Figure 6. In the next two subsections, the notation $C \vdash C'$ should be read as adding the constraints in C' to C .

4.3 Inference Rules for Expressions

Figure 7 shows our type inference rules for expressions.

(NIL) and (INT) assign the types Nom_{nil} and Nom_{Fixnum} , respectively. We assume these types have been extracted from annotations on the standard library.

(LOCAL/SELF) looks up either a local variable x or `self` in Γ , and returns the corresponding type. We perform a simple syntactic analysis (not shown) to detect local variables that may be captured by blocks and hence are modeled flow-insensitively. Accesses to such variables are typed by (LOCAL-CAP), which generates a fresh variable α and a constraint $x : \alpha$, meaning that x has type α . During constraint resolution, we will combine all such constraints for the same variable so that it has the same type everywhere within a method. Note that since this constraint is global, it technically means all local variables called x are conflated, even across different methods, but we can always alpha-convert to prevent this.

(FIELD) works in a similar manner. Here we generate a constraint $\Gamma(\text{self}) \leq [\text{@}x : \alpha]$, which by width subtyping means that $\Gamma(\text{self})$ must include at least `@x` with a type compatible with α .

(CONST) is similar to (LOCAL-CAP). (TUPLE) recursively types each component of an array literal, and then assigns an appropriate tuple type to the result.

Finally, (BLOCK) assigns a type to a block. While blocks are not actually expressions, we put this type rule here because evaluating a block has no side effect. This rule creates fresh variables α_i for the formal parameter and return types. The function body s is evaluated in the outer environment Γ extended with bindings for the formals, which makes local variables bound at block definition time (including `self`) available inside the block. The context for the function body is the same as the outer context, except `ret` is bound to the fresh return type. The output environment Γ' is discarded because when a block ends, its local variables go out of scope.

4.4 Inference Rules for Statements

Figure 8 shows our type inference rules for statements.

(SEQ) types a sequence $s_1; s_2$ as expected, connecting the environments from each in order. Note that C and Ω are the same for both statements—the set of constraints is global, and the class, block, and return types vary lexically rather than flow-sensitively.

(IF) type checks a conditional. The guard may have any type; in Ruby, any value except `false` or `nil` is considered true. The two branches are typed separately in Γ , and the output environments Γ_2 and Γ_3 are combined with the pointwise union operation $\Gamma_2 \cup \Gamma_3$:

$$\begin{aligned} \Gamma \cup \emptyset &= \emptyset \cup \Gamma &= \Gamma \\ (x : \tau, \Gamma_1) \cup \Gamma_2 &= x : \tau, (\Gamma_1 \cup \Gamma_2) \quad x \notin \text{dom}(\Gamma_2) \\ \Gamma_1 \cup (x : \tau, \Gamma_2) &= x : \tau, (\Gamma_1 \cup \Gamma_2) \quad x \notin \text{dom}(\Gamma_1) \\ (x : \tau_1, \Gamma_1) \cup (x : \tau_2, \Gamma_2) &= x : (\tau_1 \cup \tau_2), (\Gamma_1 \cup \Gamma_2) \end{aligned}$$

(LASSIGN) types assignment to a local variable x that is not captured by a block, and in the output environment Γ' , the type of x is τ . In (LASSIGN-CAP) we generate a constraint that x has type τ (globally). As mentioned earlier, during constraint resolution we will merge all re-

<p>(SEQ)</p> $\frac{C; \Gamma, \Omega \vdash s_1; \Gamma' \quad C; \Gamma'; \Omega \vdash s_2; \Gamma''}{C; \Gamma; \Omega \vdash s_1; s_2; \Gamma''}$	<p>(IF)</p> $\frac{C; \Gamma; \Omega \vdash e_1 : \tau \quad C; \Gamma; \Omega \vdash s_2; \Gamma_2 \quad C; \Gamma; \Omega \vdash s_3; \Gamma_3}{C; \Gamma; \Omega \vdash \text{if } e_1 \text{ then } s_2 \text{ else } s_3; \Gamma_2 \cup \Gamma_3}$
<p>(LASSIGN)</p> $\frac{C; \Gamma; \Omega \vdash_e e : \tau \quad \Gamma' = \Gamma[x \mapsto \tau] \quad x \text{ not captured}}{C; \Gamma; \Omega \vdash x = e; \Gamma'}$	<p>(LASSIGN-CAP)</p> $\frac{C; \Gamma; \Omega \vdash_e e : \tau \quad C \vdash x : \tau \quad x \text{ may be captured}}{C; \Gamma; \Omega \vdash x = e; \Gamma}$
<p>(CASSIGN)</p> $\frac{C; \Gamma; \Omega \vdash_e e : \tau \quad C \vdash A : \tau}{C; \Gamma; \Omega \vdash A = e; \Gamma}$	<p>(FASSIGN)</p> $\frac{C; \Gamma; \Omega \vdash_e e : \tau \quad [\@x : \tau; \dots] \leq \Gamma(\text{self})}{C; \Gamma; \Omega \vdash \@x = e; \Gamma}$
<p>(PASSIGN)</p> $\frac{C; \Gamma; \Omega \vdash_e e : \tau \quad C \vdash \tau \leq (\alpha_1 \times \dots \times \alpha_n) \quad \alpha_i \text{ fresh} \quad \Gamma' = \Gamma[x_i \mapsto \alpha_i] \quad x_i \text{ not captured} \quad i \in 1..n}{C; \Gamma; \Omega \vdash x_1, \dots, x_n = e; \Gamma'}$	
<p>(NEW)</p> $\frac{C; \Gamma; \Omega \vdash_e e : \tau \quad C \vdash \tau \Rightarrow \alpha \quad \alpha \text{ fresh} \quad \Gamma' = \Gamma[x \mapsto \alpha]}{C; \Gamma; \Omega \vdash x = e.\text{new}; \Gamma'}$	<p>(RETURN)</p> $\frac{C; \Gamma; \Omega \vdash_e e : \tau \quad C \vdash \tau \leq \Omega.\text{ret}}{C; \Gamma; \Omega \vdash \text{return } e, \Gamma}$
<p>(CALL)</p> $\frac{C; \Gamma; \Omega \vdash_e e_i : \tau_i \quad i \in 0..n \quad C; \Gamma; \Omega \vdash_e b : bt \quad C \vdash \tau_0 \leq [m : (\tau_0 \times \dots \times \tau_n \times bt) \rightarrow \alpha] \quad \Gamma' = \Gamma[x \mapsto \alpha] \quad \alpha \text{ fresh}}{C; \Gamma; \Omega \vdash x = e_0.m(e_1, \dots, e_n)b; \Gamma'}$	
<p>(YIELD)</p> $\frac{C; \Gamma; \Omega \vdash_e e_i : \tau_i \quad C \vdash \Omega.\text{blk} \leq (\tau_1 \dots \times \tau_n) \rightarrow \alpha \quad \Gamma' = \Gamma[x \mapsto \alpha] \quad i \in 1..n \quad \alpha \text{ fresh}}{C; \Gamma; \Omega \vdash x = \text{yield}(e_1, \dots, e_n); \Gamma'}$	
<p>(CLASS)</p> $\frac{Nom_A = \{\text{kind: } Nom_{\text{Class}}; \dots\} \quad C \vdash A : Nom_A \quad C; \{\text{self} \mapsto Nom_A\}; \Omega[\text{cls: } Nom_A] \vdash s; \Gamma'}{C; \Gamma; \Omega \vdash \text{class } A = s, \Gamma}$	
<p>(INHERIT)</p> $\frac{C \vdash A : \alpha \quad \alpha \text{ fresh} \quad C \vdash \alpha \in \Omega.\text{cls.par}}{C; \Gamma; \Omega \vdash \text{inherit } A; \Gamma}$	
<p>(DEF)</p> $mt = (\alpha_0 \times \dots \times ?\alpha_n \times \dots \times * \alpha_m \times \beta \rightarrow \alpha_{m+1})$ $C \vdash (m : mt) \in \Omega.\text{cls.mths}$ $\frac{\Gamma' = \{\text{self} \mapsto \alpha_0, x_i : \tau_i\} \quad \Omega' = \Omega[\text{ret: } \alpha_{m+1}, \text{blk: } \beta] \quad C; \Gamma'; \Omega' \vdash s; \Gamma'' \quad C; \Gamma'; \Omega' \vdash_e e_k : \tau_k \quad C \vdash \tau_k \leq \alpha_k \quad C \vdash Nom_{\text{Array}} \langle \tau_m \rangle \leq \alpha_m \quad k \in n..m-1 \quad i \in 1..m \quad \alpha_0, \alpha_i, \alpha_{m+1}, \beta \text{ fresh}}{C; \Gamma; \Omega \vdash \text{def } m(x_1, \dots, [x_n = e_n, \dots]^?, [*x_m]^?) = s, \Gamma}$	

Figure 8. Type judgments for statements

lated constraints to ensure x has the same type everywhere. (CASSIGN) behaves similarly.

(FASSIGN) types an update to a field. Here $[\@x : \tau; \dots]$ means a structural type with $\@x : \tau$ and the remaining fields and methods are unconstrained. We use fresh *row variables* to represent unconstrained field and method lists (Rémy 1989; Furr and Foster 2006), but we omit them here for simplicity. Notice that field types are flow-insensitive, as they are stored in the type of $\Gamma(\text{self})$, which is the same throughout the method body.

(PASSIGN) types parallel assignment to non-captured local variables. We create a tuple type containing n fresh variables and constrain τ to be a subtype of it. In the output environment, each variable on the left-hand side is bound to the appropriate type. Our implementation also includes an extended version of (PASSIGN) that handles parallel assignment to any possible sequence of *lvals*. We omit this rule since it is messy and provides no new insight. Similarly, the remaining type rules involving assignment only show the case for a non-captured local; the other forms of these rules are straightforward.

(NEW) types object creation. The expression e , which is typically a constant A , is inferred to have type τ . We create a fresh variable α to represent the type of the newly created object and generate an instance constraint $\tau \Rightarrow \alpha$. This constraint, a kind of *instantiation constraint* (Fähndrich et al. 2000), improves our handling of polymorphic classes. In a typical implementation of parametric polymorphism, to type $x=A.\text{new}$ we would need to know the type of A at that point to instantiate A 's type. By representing class instantiation as a constraint, we can instead apply our type rules in a single linear pass. When we encounter $x=A.\text{new}$, if A has some unknown type β , we will generate a constraint $\beta \Rightarrow \alpha$. During constraint resolution we will find a solution for β (i.e., the type it represents) and instantiate it.

(RETURN) constrains the current return type ret —either the current block or current method—to be a supertype of the returned expression type. (Note we are simplifying here, since in Ruby, return only exits methods, not blocks.)

(CALL) types the receiver object, actual parameters, and block parameter. (Note we assume here that a block has been specified; a simple variation handles the case with no block.) It then constrains the receiver type τ_0 to contain a method of the right type. Notice that τ_0 , the type of self , appears as the first argument type. During constraint resolution, if τ_0 is a polymorphic type we will instantiate it when resolving this constraint. (YIELD) is similar to (CALL), except it constrains the block from the current context.

(CLASS) creates a fresh nominal type Nom_A that is itself an instance of Nom_{Class} and is otherwise unconstrained (indicated by \dots). We then add a constraint $A : Nom_A$. If the same class is opened multiple times, we will generate several of these constraints, and constraint resolution will merge the information from all the class openings together. We type the

body s of the class in an environment with self bound to the class type and no other bindings; this conforms to the scoping rules of Ruby. When typing s , we also update the context so that `cls` is the newly created class type.

(INHERIT) is surprisingly simple. We generate constraints that the inherited class A have some type α that is contained in the parents of the current class. During constraint resolution we traverse all the parents implied by such constraints when searching for methods.

Finally, (DEF) types the definition of a new method, similarly to (BLOCK). We create fresh variables for the method parameters and return value, and require the current class have a method of the appropriate type. We then type the method body in a fresh environment with self bound to the first type in the parameter list, and the x_i bound to the remaining types. Notice that unlike blocks, methods cannot access locals from the outer scope. When typing the method body, we also update the context so the return and block types are bound appropriately. We discard the output environment Γ'' as expected. The optional arguments are typed with the same environment as the method body (following Ruby’s scoping rules), and we add constraints to relate them to the formal parameter types. For the vararg parameter, we add a constraint that α_m is an instance of an array type instantiated to contain the specified parameter type.

4.5 Constraint Resolution

After applying the type inference rules, we *resolve* the generated set of constraints by exhaustively applying a set of rewrite rules. For example, given $\tau \leq \alpha$ and $\alpha \leq \beta$, we add the $\tau \leq \beta$. During this process, we issue a warning if any inconsistencies arise. For example, given $Nom \leq [m : mt]$, if the class described by Nom has no method m , 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.

Our constraint resolution process is a variation on fairly standard techniques (Aiken et al. 1998; Furr and Foster 2006). Rather than present an exhaustive set of rewriting rules, we briefly discuss a few key features of our algorithm.

We begin by solving the constraints $\tau' \in \tau.par$ and $(m : mt) \in \tau.mths$ by simply adding the appropriate entries into $\tau.par$ or $\tau.mths$, respectively. In the first case, we require that τ be a Nom and produce an error message if it is not. Constraints $A : \tau$ and $x : \tau$ are also straightforward: We gather all such constraints with the same left-hand side and unify the corresponding types. E.g., given constraints $A : \tau_1, \dots, A : \tau_n$, we require $\tau_1 = \dots = \tau_n$. (Here $\tau = \tau'$ is shorthand for $\tau \leq \tau'$ and $\tau' \leq \tau$.)

As alluded to above, given a constraint $Nom \leq [m : mt]$, we derive a new constraint $Nom(m) \leq mt$, where $Nom(m)$ looks up m in Nom , first checking $Nom.mths$ and then, since methods may be inherited, recursively checking in $Nom.par$ if needed. Handling field constraints $Nom \leq [@x : \tau]$ and $[@x : \tau; \dots] \leq Nom$ is similar. When working with tuple

types $(\tau_1 \times \dots \times \tau_n)$, we allow them to be treated as array types as outlined in Section 3.

There are a number of special cases in resolving method subtyping $mt_1 \leq mt_2$. We allow the number of arguments to differ between mt_1 and mt_2 according to the expected rules for optional and varargs. If mt_1 is universally quantified, we instantiate it with fresh variables before performing subtyping. More details about type inference with object types and universals can be found elsewhere (Furr and Foster 2006). If mt_1 is an intersection type $mt_1^1 \cap \dots \cap mt_1^n$, there are three cases for solving the constraint. If all of mt_2 ’s parameters are known to include some nominal type (directly or transitively), then for every mt_1^i that is compatible with all mt_2 ’s parameter types, we add the constraint $mt_1^i \leq mt_2$. If there are no matches but we can exclude all but one mt_1^j as a possibility, we add the constraint $mt_1^j \leq mt_2$. Otherwise, we leave the constraint unresolved until we have further information.

Lastly, given a constraint $\tau_1 \Rightarrow \tau_2$, we instantiate τ_1 to τ_3 and add the constraint $\{\text{kind} = \tau_3\} \leq \tau_2$. When instantiating τ_1 , there are two cases to consider. If τ_1 is universally quantified, we create a set of fresh variables $\vec{\alpha}$ and produce $\tau_3 = \tau_1(\vec{\alpha})$. Otherwise, we have set $\tau_3 = \tau_1$.

5. Implementation and Experiments

We implemented our type inference system as a tool we call DRuby, which is comprised of approximately 13,800 lines of OCaml and 1,760 lines for the lexer and parser.

DRuby supports type inference for all of RIL, including the language constructs discussed in Section 2.4. As mentioned in that section, the operations `new` and `include`, which appear to be primitives, are actually method calls. DRuby handles these calls as described by the type rules in Section 4. For `new`, we also add the appropriate constraints to model the call to `initialize`. 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.

As mentioned in Section 2.4, Ruby objects have a virtual class (a.k.a. *eigenclass*) that holds so-called *eigenmethods* and *eigenfields* that are specific to a single instance of an object. DRuby models typical use of these constructs, and apart from slightly complicating the name resolution order for a method call, adding typing constraints for an object’s *eigenclass* is straightforward.

Ruby programs can load code from other files by invoking either `require`, which reads a file at most once no matter how many times it is required, or `load`, which always reads a file. 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.

Currently, DRuby flattens the namespace for classes and constants into a single level. For example, DRuby treats $A :: C$, $B :: C$, and $:: C$ as referring to the same symbol. This is sound but somewhat conservative. For example,

Program	LOC	Changes	Tm (s)	FPos
<i>merge-bibtex</i>	103	None	2.0	0
<i>sudokusolver-1.4</i>	152	None	3.3	35
<i>ObjectGraph-1.0.1</i>	153	None	2.3	1
<i>itcf-1.0.0</i>	178	S-2	4.3	0
<i>text-highlight-1.0.2</i>	283	S-1, M-2	2.7	1
<i>rhotoalbum-0.4</i>	313	S-2	3.5	0
<i>gs_phone-0.0.4</i>	542	S-1	2.6	0
<i>StreetAddress-1.0.1</i>	892	(S, R)-1	36.1	0

S—added stub file; M—manually expanded meta-programming code;
R—replaced require argument with constant String

Figure 9. Experimental Results

we initially tried analyzing portions of the Ruby standard library, but found that parts of it, particularly REXML, an XML processing library, reuse the same class name in different parts of the namespace, and so flattening causes us to lose significant precision. We expect this issue can be addressed fairly easily in the future.

5.1 Benchmarks

We evaluated DRuby by applying it to a suite of small programs gathered from our colleagues and RubyForge. The left portion of Figure 9 lists the benchmark names, their size as computed by SLOCCount (Wheeler 2008), and the number and kinds of changes required to be analyzable by DRuby. The analyzed code includes both the application and any test suite shipped with the software.

We made three kinds of changes to the benchmarks so that DRuby could analyze them. First, we added stub files (changes labeled “S”) to assign types to loaded code that was written in Ruby, but that DRuby could not analyze. In particular, the programs *itcf* (twice), *rhotoalbum* (twice), *StreetAddress*, and *gs_phone* load files from the standard library. Those files in turn load more code transitively, and as a result, even though the programs are small, DRuby can end up analyzing tens of thousands of lines of code. Moreover, the standard library employs many of the reflective and highly dynamic constructs that DRuby does not model, resulting in a multitude of false positives and greatly hurt its performance (because of cascading error messages). When pulling in standard library code, DRuby does not terminate within 5 minutes for these programs and reports hundreds of errors. With annotations for the necessary parts of the library, DRuby terminates in seconds and issues no warnings. We also add a stub file for *text-highlight*. In this case the stub is for a logging module *text-highlight* refers to but does not include, and that we could not locate anywhere.

Second, we expanded two uses of metaprogramming (labeled “M”) in *text-highlight*, which twice used the code shown at the end of Section 3 to add method definitions. DRuby produces 5 extra false positives without this change.

Finally, one program, *StreetAddress*, included the call “require File.dirname(__FILE__) + ‘ ../ lib /street_address’”

We replaced this with the appropriate constant string (change labeled “R”) to accurately analyze the program.

5.2 Experimental Results

The right portion of Figure 9 shows the running times for DRuby and the number of reported warnings. Times were the average of 5 runs on an AMD Athlon 4600 processor with 4GB of memory. For these programs, all of the warnings were false positives, meaning that no type errors would occur at run time.

For these small examples, DRuby runs quite quickly, though as mentioned before it does not currently scale without annotations for the standard library. Most of the benchmarks yielded no warnings, i.e., DRuby’s type system is precise enough to model all the constructs in these programs.

The *sudokusolver* benchmark produced 35 false positives, all due to a union type that is discriminated by run-time tests. In particular, 3 methods return either false or an array. Clients of these methods check the return values against false before using them as arrays, but DRuby does not model these tests. This is a place where occurrence types (Tobin-Hochstadt and Felleisen 2008) could improve precision.

The one false positive in *ObjectGraph* occurs when the code tests whether StringIO is defined before instantiating it. Interestingly, if the class is undefined, the program creates a simulation of StringIO by extending the eigenclass of String:

```

if defined? StringIO
  s = StringIO.new
else
  s = String.new
  class << s # extend eigenclass of s
    def puts(str) ... end
  ...
end end

```

Because StringIO is not loaded by default, DRuby produces a warning that StringIO is used without being defined. Out of curiosity, we manually added a require to load StringIO, and found DRuby was able to type a later call to `s.puts`.

The last false positive, in *text-highlight*, is due to calling the `extend` method, which DRuby currently does not model. The method `extend` is similar to `include` but instead adds methods to an object’s eigenclass rather than its class. Modeling this behavior should be easy to add to DRuby.

6. Related Work

Many researchers have previously studied the problem of applying static typing to dynamic languages. We group these prior results by language.

Scheme 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 (Cartwright and Fagan 1991; Aiken et al. 1994; Wright and Cartwright 1997; Flanagan et al. 1996). Soft typing aims to detect potential program errors and remove unneeded run

time checks. Our type inference system is similar in spirit to soft typing, but is significantly more sophisticated, including support for object types, intersection types, and other features needed for Ruby.

Typed Scheme adds type annotations and type checking to Scheme (Tobin-Hochstadt and Felleisen 2008). 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 in Section 5, one of our benchmarks could benefit from adding occurrence types, but we have not explored this option yet. Also, in contrast to our system, typed Scheme includes only intraprocedural type inference, whereas our inference system is interprocedural.

Smalltalk and Self Several researchers investigated static typing for Smalltalk, a close relation of Ruby. Graver and Johnson (1990) propose a type checking system for Smalltalk that includes class, object, union, and block types. Strongtalk, a variant of Smalltalk extended with static types, has a similar system with additional support for mixins (Bak et al. 2002). Agesen et al. (1993) and Agesen and Hölzle (1995) explored type inference for Self, which is an object-based (rather than class-based) language. The focus of all of these systems is on performance improvement, whereas our focus is error detection. Moreover, our system integrates both type annotations and type inference, and includes support for more language features.

Ruby The idea of type inference for Ruby has been proposed before. Kristensen (2007) claims to have developed a Ruby type inference system, but his thesis is not available on-line, and emails requesting it were not returned. Morrison (2006) developed a type inference algorithm that has been integrated into RadRails, an authoring environment 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.

Python There have been several type systems proposed for Python: Aggressive type inference (Aycock 2000), Starkiller (Salib 2004), and a system proposed by Cannon (2005) all infer types for Python code. These systems are focused on performance, and seem to be based purely on nominal typing with no intersection types. RPython is a statically-typed subset of Python designed to compile to JVM and CLI bytecode (Ancona et al. 2007). RPython includes a 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. We plan to incorporate a similar technique into DRuby.

Javascript There are several proposals for type inference for Javascript (Anderson et al. 2005; Thiemann 2005). The

proposed ECMAScript 4 language, which will succeed Javascript, includes a rich type annotation language, including object types, tuple types, parametric polymorphism, and union types (Hansen 2007). The challenges in typing Ruby are somewhat different than for Javascript, which has no explicit classes but rather builds objects by assigning them sets of pre-methods. Javascript also includes a range of silent conversions between different types, which further complicates type checking and inference.

Erlang Marlow and Wadler (1997) developed a soft typing system for Erlang, and used it to type check several Erlang programs. Success typing for Erlang (Lindahl and Sagonas 2006) is similar to soft typing, but instead of warning about any programs that *may* produce run-time errors, success typing tries to flag code that *definitely* produces run-time errors. These systems share a few characteristics with our approach, such as subtyping and union types, but are otherwise fairly different due to language differences.

Other Related Work Aside from work on particular dynamic languages, the question of statically typing dynamic language constructs has been studied more generally. Abadi et al. (1991) 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 (Thatte 1990). Gradual type systems improve on this idea further (Siek and Taha 2006; Herman et al. 2007), and have been proposed for object-oriented type systems (Siek and Taha 2007). Sage mixes a very rich static type system with the type Dynamic (Gronski et al. 2006). Tobin-Hochstadt and Felleisen (2006) present a framework for gradually changing program components from untyped to typed. We expect to incorporate ideas from all of these systems into DRuby to increase flexibility and better handle values that cannot be statically typed.

7. Conclusions

We have presented a type inference system for checking the static type safety of Ruby programs. In order to analyze real-world Ruby programs, we developed a GLR parser and intermediate language to make a type analysis tractable. Our current benchmark suite contains small but interesting programs, and our powerful type language was necessary to analyze them precisely. Indeed, nearly every feature of our system was motivated in some way by code we tried to analyze, particularly in our initial attempts to analyze the standard library. We believe that DRuby is a major step toward the goal of developing an integrated static and dynamic type system for Ruby in particular, and object-oriented scripting languages in general.

References

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

- Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.
- O. Agesen, J. Palsberg, and M.I. Schwartzbach. Type Inference of SELF. *ECOOP*, 1993.
- Ole Agesen and Urs Hölzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *OOPSLA*, pages 91–107, 1995.
- Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft Typing with Conditional Types. In *POPL*, pages 163–173, 1994.
- Alexander Aiken, Manuel Fähndrich, Jeffrey S. Foster, and Zhen-dong Su. A Toolkit for Constructing Type- and Constraint-Based Program Analyses. In *TIC*, pages 78–96, 1998.
- Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas Matsakis. Rpython: Reconciling dynamically and statically typed oo languages. In *DLS*, 2007.
- Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards Type Inference for JavaScript. In *ECOOP*, pages 428–452, 2005.
- John Aycock. Aggressive Type Inference. In *Proceedings of the 8th International Python Conference*, pages 11–20, 2000.
- L. Bak, G. Bracha, S. Grarup, R. Griesemer, D. Griswold, and U. Holzle. Mixins in Strongtalk. *Inheritance Workshop at ECOOP*, 2002.
- Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP*, pages 303–311, 1990.
- Brett Cannon. Localized Type Inference of Atomic Types in Python. Master’s thesis, California Polytechnic State University, San Luis Obispo, 2005.
- Robert Cartwright and Mike Fagan. Soft typing. In *PLDI*, pages 278–292, 1991.
- Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable Context-Sensitive Flow Analysis using Instantiation Constraints. In *PLDI*, pages 253–263, 2000.
- Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching Bugs in the Web of Program Invariants. In *PLDI*, pages 23–32, 1996.
- David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O’Reilly Media, Inc, 2008.
- Michael Furr and Jeffrey S. Foster. Polymorphic Type Inference for the JNI. In *ESOP*, pages 309–324, 2006.
- Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 136–150, White Plains, New York, June 1990.
- J. Gronski, K. Knowles, A. Tomb, S.N. Freund, and C. Flanagan. Sage: Hybrid Checking for Flexible Specifications. *Scheme and Functional Programming*, 2006.
- Lars T Hansen. Evolutionary Programming and Gradual Typing in ECMAScript 4 (Tutorial), November 2007.
- D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Trends in Functional Programming*, 2007.
- Kristian Kristensen. Ecstatic – Type Inference for Ruby Using the Cartesian Product Algorithm. Master’s thesis, Aalborg University, 2007.
- Xavier Leroy. The Objective Caml system, August 2004.
- Tobias Lindahl and Konstantinos Sagonas. Practical type inference based on success typings. In *PPDP*, pages 167–178, 2006.
- Simon Marlow and Philip Wadler. A practical subtyping system for erlang. In *ICFP*, pages 136–149, 1997.
- John C. Mitchell. Type inference with simple subtypes. *JFP*, 1(3): 245–285, July 1991.
- Jason Morrison. Type Inference in Ruby. Google Summer of Code Project, 2006.
- George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC*, pages 213–228, 2002.
- Emmanuel Onzon. *dypgen User’s Manual*, January 2008.
- B.C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, 1991.
- Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
- Didier Rémy. Typechecking records and variants in a natural extension of ML. In *POPL*, pages 77–88, 1989.
- Michael Salib. Starkiller: A Static Type Inferencer and Compiler for Python. Master’s thesis, Massachusetts Institute of Technology, 2004.
- Jeremy Siek and Walid Taha. Gradual typing for objects. In *ECOOP*, pages 2–27, 2007.
- Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.
- Bruce Stewart. An Interview with the Creator of Ruby, November 2001. <http://www.linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html>.
- Strongtalk. Strongtalk: Smalltalk...with a need for speed, 2008. <http://www.strongtalk.org/>.
- Satish Thatte. Quasi-static typing. In *POPL*, pages 367–381, 1990.
- Peter Thiemann. Towards a type system for analyzing javascript programs. In *ESOP*, pages 408–422, 2005.
- Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmers’ Guide*. Pragmatic Bookshelf, 2nd edition, 2004.
- Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *POPL*, pages 395–406, 2008.
- Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *OOPSLA*, pages 964–974, 2006.
- Bill Venners. The Philosophy of Ruby: A Conversation with Yukihiro Matsumoto, Part I, September 2003. <http://www.artima.com/intv/rubyP.html>.
- David A. Wheeler. Sloccount, 2008. <http://www.dwheeler.com/sloccount/>.
- A.K. Wright and R. Cartwright. A practical soft type system for scheme. *ACM TOPLAS*, 19(1):87–152, 1997.