ABSTRACT

Title of dissertation: TYPE CHECKING AND INFERENCE FOR DYNAMIC LANGUAGES

Brianna Ren
Doctor of Philosophy, 2019

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

Object-oriented dynamic languages such as Ruby, Python, and JavaScript provide rapid code development and a high degree of flexibility and agility to the programmer. Some of the their main features include dynamic typing and metaprogramming. In dynamic typing, programmers do not declare or cast types, and types are not known until run time. In addition, an object's suitability is determined by its methods, as opposed to its class. Metaprogramming dynamically generates code as the program executes, which means that methods and classes can be added and modified at run-time. These features are powerful but lead to a major drawback of dynamic languages: the lack of static types means that type errors can remain latent long into the software development process or even into deployment, especially in the presence of metaprogramming. To bring the benefits of static types to dynamic languages, I present three pieces of work.

First, I present the Ruby Type Checker (rtc), a tool that adds type checking to Ruby. Rtc addresses the issue of latent type errors by checking all types

during run time at method entrance and exit. Thus it checks types later than a purely static system, but earlier than a traditional dynamic type system. Rtc is implemented as a Ruby library and supports type annotations on classes, methods, and objects. Rtc provides a rich type language that includes union and intersection types, higher-order (block) types, and parametric polymorphism, among other features. We applied rtc to several apps and found it effective at checking types.

Second, I present Hummingbird, a just-in-time static type checker for dynamic languages. Hummingbird also prevents latent type errors, and type checks Ruby code even in the presence of metaprogramming, which is not handled by rtc. In Hummingbird, method type signatures are gathered dynamically at run-time, as those methods are created. When a method is called, Hummingbird statically type checks the method body against current type signatures. Thus, Hummingbird provides thorough static checks on a per-method basis, while also allowing arbitrarily complex metaprogramming. We applied Hummingbird to six apps, including three that use Ruby on Rails, a powerful framework that relies heavily on metaprogramming. We found that all apps type check successfully using Hummingbird, and that Hummingbird's performance overhead is reasonable.

Lastly, I present a *practical type inference system* for Ruby. Although both rtc and Hummingbird are very effective tools for type checking, the programmer must provide the type annotations on the application methods, which may be a time-consuming and error-prone process. Type inference is a generalization of type checking that automatically infers types while performing checking. However, standard type inference often infers types that are overly permissive compared to what

a programmer might write, or contain no useful information, such as the bottom type. I first present a standard type inference system for Ruby, where constraints on a method is statically gathered as soon as the method is invoked at run-time, and types are resolved after all constraints have been gathered on all methods. I then build a practical type inference system on top of the standard type inference system. The goal of my practical type inference system is to infer types that are concise and include actual classes when appropriate. Finally, I evaluate my practical type inference system on three Ruby apps and show it to be very effective compared to the standard type inference system.

In sum, I believe that rtc, Hummingbird, and the practical type inference system all take strong steps forward in bringing the benefits of static typing to dynamic languages.

Type Checking and Inference for Dynamic Languages

by

Brianna Ren

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
2019

Advisory Committee:
Professor Jeffrey S. Foster, Chair/Advisor
Professor Michael Hicks
Professor David Van Horn
Professor Neil Spring
Professor Donald Yeung

Acknowledgments

I would like to first thank my advisor Jeff Foster for his guidance. I extend special thanks to all other members of my committee for providing me with valuable advice.

I would also like to thank all members of PLUM, the Programming Languages group at the University of Maryland. I was fortunate enough to collaborate with Stevie Strickland, Milod Kazerounian, and talented undergrads John Toman and Alex Yu.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Object-oriented dynamic languages such as Ruby, Python, and JavaScript are popular, with many compelling features. Two of the main features are dynamic typing and metaprogramming. In dynamic typing, programmers do not need to declare types on variables, and variables are not associated with types until run-time. In addition, a variable's type compatibility is determined by the methods defined on it, as opposed to its actual class. Metaprogramming allows methods and classes to be added or modified at run-time. These powerful features help support rapid prototying and provide a high degree of flexibility and agility to the programmer. In static typing, type errors are caught early at compile time, which helps reduce the number of bugs and debugging time. In addition, the annotated types provide documentation about a method, and that documentation is automatically checked against the code for consistency. The lack of static types means that programs can be hard to understand, and subtle errors can remain latent in code for a long time. Morever, metaprogramming-created code complicates this issue. This dissertation addresses these problems with three type systems that we have built: 1) rtc, a purely dynamic Ruby type checker, 2) Hummingbird, a just-in-time static type checking system for dynamic languages, and 3) a *practical type inference system*, a system

that infers concise types.

Thus, I provide the following thesis:

In dynamic languages, types can be effectively checked at an intermediate point between purely static and purely dynamic checking. Further, programs that use metaprogramming generated code can also be effectively type checked at this intermediate point. Finally, practical type inference can be used to reduce the number of manual annotations and infer types that are concise and resemble manual annotations.

To substantiate this thesis, I present three type systems below.

1.1 Rtc: The Ruby Type Checker

Recall that a major disadvantage of dynamic languages is that type errors can remain latent long into the software development process or even into deployment. To address this concern, there have been many proposals for adding static types to dynamic languages [3, 4, 8, 10, 13–15]. While these prior systems are promising, they have two key limitations. First, because they are purely static, they do not deal well with highly dynamic language features such as `eval` or reflective method invocation. Second, since static type systems must be conservative, in practice they can categorize too many programs as erroneous. Adding precision in the form of flow-, path-, and context-sensitivity helps, but also tremendously complicates the type system.

We introduce rtc, the Ruby Type Checker [36], a tool I (along with my col-

leagues) built as an intermediate point between pure static and pure dynamic checking. In rtc, types are checked at run-time—which is later than static typing—but at method entrance and exit—which is earlier than dynamic typing. Because rtc operates at run time, it can handle highly dynamic language features in a natural way. Moreover, as rtc only observes feasible program executions, it automatically includes the sensitivities mentioned above. Rtc is heavily inspired by and builds on the codebase of An et al's Rubydust system [1], which falls at the same design point. However, rtc is a pure type *checking* system, whereas Rubydust performs constraint-based type inference, which results in several technical and implementation differences.

Rtc supports annotations on classes, methods, and objects, and rtc's type system includes nominal types, union and intersection types, block (higher-order method) types, parametric polymorphism, and type casts. A key design principle of rtc is that programmers should only "pay for what they use." That is, programs without annotations should run as usual, and programs with annotations should only perform checking where desired. To achieve this, rtc separates objects into *raw* (untyped) values and *annotated* (typed) values. Type checking only occurs when annotated values are used as receivers. Annotations are introduced either explicitly by the programmer or implicitly when values are passed as arguments to type-checked calls. We think this design strikes the right balance of providing fine enough control over type checking without requiring too much explicit annotation.

Rtc is implemented in a similar fashion to Rubydust. Annotated objects are wrapped by proxy objects that associate types with the underlying object. When a

proxy is invoked, it performs type checking before and after it delegates the method call to the underlying object. The proxy also annotates the incoming arguments and the returned value. Rtc uses some implementation tricks to maintain annotations on self, which would otherwise be lost when the proxy delegates to the underlying object; to handle block type checking; and to allow classes to be declared as auto-annotating, so that all instances of the class are proxied by default.

We evaluated rtc by adding type annotations to several small programs and running the test suites included with those programs. We found that all of the features of rtc were useful in typing our subject programs, and we were able to assign rtc types to most methods. We also found that while the overhead of rtc is substantial in relative terms, in absolute terms the test suites for our subject programs still execute quickly.

In summary, we think that rtc is a practical, useful, and effective tool for increasing the type safety of Ruby programs, and that the ideas of rtc can be ported to other languages.

1.2 Just-in-Time Static Type Checking for Dynamic Languages

Although rtc is an effective tool, it does not work well in the presence of *metaprogramming*, in which code the program relies on is generated as the program executes. The challenge is that purely static systems cannot analyze metaprogramming code, which is often complicated and convoluted; and prior mixed static/dynamic systems are either cumbersome or make certain limiting assumptions.

We introduce Hummingbird [35], a type checking system I (along with my advisor) built for Ruby that solves this problem using a new approach we call *just-in-time static type checking*. In Hummingbird, user-provided type annotations actually execute at *run-time*, adding types to an environment that is maintained during execution. As metaprogramming code creates new methods, it, too, executes type annotations to assign types to dynamically created methods. Then whenever a method m is called, Hummingbird *statically* type checks m's body in the current dynamic type environment. More precisely, Hummingbird checks that m calls methods at their types as annotated, and that m itself matches its annotated type. Moreover, Hummingbird caches the type check so that it need not recheck m at the next call unless the dynamic type environment has changed in a way that affects m.

Just-in-time static type checking provides a highly effective tradeoff between purely dynamic and purely static type checking. On the one hand, metaprogramming code is very challenging to analyze statically, but in our experience it is easy to create type annotations at run time for generated code. On the other hand, by statically analyzing whole method bodies, we catch type errors earlier than a purely dynamic system, and we can soundly reason about all possible execution paths within type checked methods.

We evaluated Hummingbird by applying it to six Ruby apps. Three use Ruby on Rails (just "Rails" below), a popular, sophisticated web app framework that uses metaprogramming heavily both to make Rails code more compact and expressive and to support "convention over configuration." We should emphasize that Rails's use of metaprogramming makes static analysis of it very challenging [25]. Two apps

use other styles of metaprogramming, and the last app does not use metaprogramming, as a baseline.

We found that all of our subject apps type check successfully using Hummingbird, and that dynamically generated types are essential for the apps that use metaprogramming. We also found that Hummingbird's performance overhead ranges from 19% to 469%, which is much better than prior approaches [1, 36], and that caching is essential to achieving this performance. For one Rails app, we ran type checking on many prior versions, and we found a total of six type errors that had been introduced and then later fixed. We also ran the app in Rails *development mode*, which reloads files as they are edited, to demonstrate how Hummingbird type check caching behaves in the presence of modified methods.

In summary, we believe Hummingbird is an important step forward in our ability to bring the benefits of static typing to dynamic languages while still supporting flexible and powerful metaprogramming features.

1.3 Practical Type Inference

Rtc and Hummingbird are important steps forward in bringing the benefits of static typing to dynamic languages. However, in both type systems, the programmer must provide annotations on the application methods, which may be a time-consuming and error prone process.

To address this issue, we introduce a standard type inference system for Ruby and then build a *practical type inference system* on top it. In both systems, we wrap

each method with subtype constraint generation code that executes as soon as the method is invoked. After all constraints have been gathered, we perform constraint resolution on the original constraints to introduce new constraints. Finally, we use a solution extraction algorithm to read off the solutions based on the final set of constraints.

Our standard type inference system has some improvements over existing standard type inference systems, mostly to resolve each intersection method type to a single method type. An intersection type is a list of valid method types for a particular method, only one of which is valid for a particular call. We introduce two new types in our system, possible and implication types. A possible type is specificies a list of types in which only one of the type is valid. An implication type specifies a list of implications in which only one conclusion is valid. We use these new types along with variable delay operations and other features in an attempt to resolve each intersection method type to a single method type.

We then introduce a practical type inference system that builds on our standard inference system to infer types that are practical, meaning that the types are concise, easy to understand, and resemble what a prorgrammer would write. The practical constraint resolution rules are mostly a super set of the standard constraint resolution rules. We also modify the solution extraction algorithm to consider certain constraints that are not used in conventional solution extraction.

We applied the standard and practical type inference systems to three Ruby apps, and found the practical type inference system much more successful than the standard system.

Chapter 2: The Dynamic Ruby Type Checker

This chapter introduces rtc, a purely dynamic Ruby type checker. Rtc is implemented as a Ruby library in which a method's arguments are type checked at method entrance and return value is type checked at method exit. Rtc is designed so programmers can control exactly where type checking occurs: type-annotated objects serve as the "roots" of the type checking process, and unannotated objects are not type checked.

Attribution and Acknowledgements

The work described in this chapter was previously published in the 2013 Object-Oriented Program Languages and Systems (OOPS) Track at ACM Symposium on Applied Computing, with co-authors John Toman, T. Stephen Strickland, and Jeffrey Foster. The design, implementation, and evaluation was shared roughly equally across the authors.

2.1 Using rtc

Figure 2.1 illustrates the basic use of rtc with excerpts from a payroll program with three classes: Person, the base class for describing employees of the company;

```
require ' rtc_lib '

class Person
  rtc_annotated        ...
  typesig " personnel_id : () → Fixnum"
  def personnel_id  ... end
  typesig " self . from_id : (Fixnum) → Person"
  def self . from_id( id )  ... end
  typesig "manager: () → Manager or %false"
  def manager ... end
end

class Manager < Person
  rtc_annotated
  def employees
    # ... find all managed employees in the database
  end  ...
  typesig ("employees: () → Array<Person>")
end

class Payroll
  rtc_annotated     ...
  typesig " self . give_raise :( Fixnum,Fixnum,Fixnum)→Fixnum"
  typesig " self . give_raise :( Person,Manager,Fixnum)→Fixnum"
  def self . give_raise (emp, okayed_by, incr )
     ... # ensure okayed_by is in charge of emp
    curr = fetch_salary_from_database (emp)
     set_salary (emp, curr + incr)
  end
end

ids_1 = [1141,1231,3142] # raw, untyped value
ids_1 . push "foo"            # allowed for raw value
ids_2 = [1141,1231,3142]. rtc_annotate ("Array<Fixnum>")
ids_2 . push "foo"            # type error
m = Person.from_id(1141)  # Assuming employee number 1141 is a Manager
m.employees # type error
m_1 = m.rtc_annotate "Manager"  # type error
m_2 = m.rtc_cast "Manager"  # ok
m_2.employees # ok
sm = m.manager # sm: Manager or %false
unless sm
  ssm_1 = sm.manager # type error
  ssm_2 = sm.rtc_cast("Manager").employees # ok
end
```

Figure 2.1: Basic usage of rtc

Manager, a subclass of Person that includes extra information for managers; and Payroll, a class for modifying the company's payroll.

The program begins by calling require to load the rtc_lib library, which contains rtc's implementation. Next are the class definitions for Person, Manager, and Payroll. All three definitions start with a call to rtc_annotated, which makes annotation methods, such as typesig, available locally. The programmer declares types for methods by calling typesig with a string that contains the method name and its type. Annotating a method with typesig tells rtc to intercept calls to the method to perform typechecking (more on this in Section 2.2). For example, personnel_id (line 5) is an instance method that takes no arguments and returns the employee's id number as a Fixnum.[1] Class method from_id (line 7) takes an id number and returns the appropriate instance of Person. Finally, the manager instance method returns either the Manager of the employee or false if the employee has no manager; note the use of a union type on line 9 to denote these possibilities. Here, %false is shorthand for the class FalseClass, of which the value false is the only inhabitant. This and other type aliases like %true, %bool, and %any are used to make types both clear and concise. Rtc allows programmers to define type aliases with typesig "type %type_name= t", where t is some valid rtc type. After the above call %type_name may be used within the defining class wherever a type is expected.

Class Manager includes a method employees that returns an array of employees managed by the receiver. Notice that we provide the type annotation for employees on line 18 *after* its definition. We use this ability to add types to the Ruby core

[1] Fixnum is the Ruby type for fixed-size integers.

library without modifying its code—instead we simply reopen the core library classes as allowed by Ruby and add appropriate typesig calls. Although we do not show it here, rtc_annotated can also appear late in a class definition, but it must occur before any other rtc forms like typesig are used.

Often in Ruby, methods are called in several different ways. One such example is Payroll#give_raise[2] on line 25. The first two arguments to the method are the employee receiving the raise and the manager that signs off on the raise. Either id numbers or objects are allowed in both positions; however, callers may not mix the two in a given call. Thus we use an intersection type: we write multiple annotations for the same method (lines 23–24), and the resulting method type is the intersection of all such annotations. When the method is called, the arguments are checked to ensure they conform to one of the allowed patterns.

In rtc, type checking happens eagerly when a method is called, which may detect errors earlier than standard dynamic typing. For example, suppose our program passes a type-incorrect final argument to give_raise. In standard Ruby, we would need to wait until the program reaches line 28 to see the error—but this may take a relatively long time if the preceding database operation is slow. In contrast, rtc detects and reports the type error on entry to give_raise. In our experience with writing Ruby programs, we are often frustrated with exactly this problem: while programs can be quick to write, they often contain small, frustrating mistakes that manifest late.

[2]Following the convention in Ruby documentation, the notation C#m refers to class C's instance method m.

One design goal of rtc is allowing programmers to use types where desired and eschewing type checking elsewhere. Thus, rtc employs a finer grained strategy than Rubydust [1], in which developers decide on a per-class basis whether to use types. In rtc, newly created objects, dubbed *raw* objects, are untyped by default, so invoking their instance methods does not involve type checking. For example, even though rtc contains type annotations for the Array class, a newly created array is initially untyped (lines 32–33). There are two ways to enable type checking for a given value. First, the programmer can use rtc_annotate to create an *annotated* version of a value that carries a type (line 34). When an annotated value is the receiver of a call to a type-annotated method, rtc performs type checking (line 35). Second, when any value, raw or annotated, is passed as an argument or returned from a method for which rtc performs type checking, then rtc checks that the value is consistent with the declared type. If the value is already typed, then rtc checks that the current type is a subtype of the desired type, raising a type error if it is not, and then rewraps the contained value with the desired type. If the value is not typed, then rtc checks first-order properties of the value, such as its class, to determine whether the value is consistent with the desired type. If it is not, then rtc raises a type error. If it is consistent, then within the method body rtc annotates the value with the declared type. For example, when our program calls Person.from_id[3] on line 36, the value 1141 is annotated with the type Fixnum within the body of Person.from_id.

Unlike instance methods, rtc checks every call to annotated class methods,

[3]Class method m of class C is referred to as C.m.

such as the call to Person.from_id on line 36. We choose to have rtc automatically check class methods to reduce the annotation burden; forcing the programmer to write C.rtc_annotate(...).m to get type-checked class methods would be a large change to existing programs.

In addition, rtc assumes a subclass is a subtype of its superclass by default; the programmer can annotate a class with no_subtype if this is not the case. Thus, it is possible for objects to become annotated with proper supertypes of their actual, run-time type. For example, on line 36 we use Person.from_id to get employee 1141 on line 36. While we may know this employee is a manager, Person.from_id is annotated to return a Person. Thus, our program cannot call the employees method directly on the result (line 37).

One design choice would be to allow rtc_annotate to perform a downcast. However, since this operation is conceptually different than an upcast, we prefer to use a distinct method call. Thus, we restrict rtc_annotate (line 38), and similarly the re-annotation that occurs at method entry, to only safe upcasts; and rtc provides rtc_cast for cases where the programmer desires a downcast during reannotation (lines 39–40). The method rtc_cast is particularly useful when working with union types. For example, on line 41 the variable sm may contain either a Manager or false. Our program uses unless to test for falsity, so on lines 43–44 we know that sm is a Manager. However, since rtc's implementation cannot automatically reassign types based on conditions (see Section 2.2), we must add an explicit use of rtc_cast to reflect this knowledge in the program.

Next, we discuss some of the key features of rtc, particularly places where type checking differs from inference significantly, or features that are lacking in Rubydust.

Blocks and procedures Ruby supports higher-order programming through the use of *code blocks*, which are anonymous methods passed in using a special syntax. Code blocks are not first-order objects (they can only be called using the special yield expression), but blocks can be freely converted to Proc objects, which are first class.

As an example, the String class defines method each_char, which calls its block argument on each character of the receiver as a string of length 1. Rtc includes the following type annotation for each_char: Here, since the return value of the block is

```
46 class String
47   rtc_annotated
48   typesig "each_char: () { (String) → %any } →String"
49 end
```

not used by each_char, we use the type %any to signify that the block may return any value.

Blocks types were not supported in Rubydust. Rtc implements block typing by wrapping block arguments in a Proc object that does type checking on entry to and exit from the block; more details are in Section 2.2.

Parametric Polymorphism Rtc supports parametric polymorphism for classes and methods. For example, here are some of the annotations already included in rtc on the built-in Array class: On line 51, we call rtc_annotated and include a two-element list argument to indicate the Array class should be parameterized by its

```
50 class Array
51   rtc_annotated [:t, :each]
52
53   typesig " '[]': (Range) → Array<t>"
54   typesig " '[]': (Fixnum, Fixnum) →Array<t>"
55   typesig " '[]': (Fixnum) → t"
56
57   typesig "map<u>: () {(t) → u } → Array<u>"
58 end
```

contents type. The first element of the list, :t, names the type parameter. The second element, :each, indicates how to find the contents type of :t for a raw Array. More specifically, when checking whether a raw array can be annotated with a type Array⟨u⟩, rtc will call the each method to iterate over all the array elements and check whether they are compatible with type u. Classes with multiple type parameters can be specified by passing multiple two-element list arguments to rtc_annotated.

Note that iterating through raw arrays is potentially very expensive, and so rtc includes a *non-strict mode* that omits it (see Section 2.2 for details). Additionally, in some cases, classes may have type parameters that are cannot be inferred by iterating over the contents. For these cases, the programmer can omit the iterator method name when calling rtc_annotated; rtc signals a type error if a raw instance of such a class is passed to a typed position.

Lines 53–55 give the type for one commonly used method, the array getter []. Note the type parameter t is in scope inside the class, so it can be used in these annotations.

Line 57 illustrates method polymorphism with the type for map. For this method, rtc attempts to infer the instantiation of u at a method call. For many

polymorphic methods we can infer the right instantiation by examining the arguments when the method is invoked. For map, however, it is slightly trickier, as rtc cannot know the return type of the block until it is called. In this case, rtc assigns u to the type of the value returned by the first call to the block, or to %none (the bottom type) if the block is never called. Further returns from the block are checked using this inferred type, and when map returns, rtc checks that the returned array is type Array⟨u⟩.

This approach to type inference may not choose the correct types for instantiation, however. Consider the following use of Array#map, where the block returns numbers for even inputs and strings for odd inputs:

```
59 a = [1,2,3]. rtc_annotate("Array<Fixnum>")
60 a.map() { |n| if (n % 2 == 0) then n else n.to_s end }
```

In the above example the call will fail because our type checker infers u to be the type String from the first use of the block, but the block returns a Fixnum from its second use. To address this issue, rtc includes a method rtc_instantiate to explicitly instantiate type parameters. In this case, the instantiation returns a method object of the correct type:

```
61 m = a. rtc_instantiate (:map,:u⇒"Fixnum or String")
62 m.call() { |n| if (n % 2 == 0) then n else n.to_s end }
```

Ambiguity in union and intersection types While union and intersection types are heavily used in type annotations, rtc must forbid some uses that are problematic from a type checking perspective. For example, recall the Person and

Manager classes from figure 2.1 and consider the following intersection type:

```
63 typesig "seat: (Person) → Cubicle"
64 typesig "seat: (Manager) → Office"
```

If we pass in a Manager, both arms of the intersection are valid since Manager is a subtype of Person. We could choose various disambiguation rules, but to keep rtc simple and predictable we opt to report an error when such an ambiguously typed method is called.

Type variables can also introduce ambiguity. For example:

```
65 typesig "m1<t,u>: (t or u) → Array<t> or Hash<String, u>"
66 typesig "m2<t>: (t) →Array<t>"
67 typesig "m2<u>: (u) →Hash<String, u>"
```

The uses of t and u above are ambiguous because they appear in the same place in a union or intersection type. Thus, rtc forbids such type annotations by reporting an error when an ambiguously typed method is called.

Similarly, having a concrete type and a type variable at the same level causes ambiguity:

```
68 typesig "m3<t>: (t or Fixnum) →Array<t>"
```

If a value of type Fixnum is provided, then we cannot determine whether type variable t should be assigned Fixnum or whether we are using the other branch of the union and t should be some other type. (Here we see that or is regular union, rather than disjoint union.)

Note that not all uses of type variables create ambiguity:

```
69 typesig "m4<t,u>: (Array<t> or Hash<String, u>) →t or u"
70
71 typesig "m5<t>: (Array<t>) →t"
72 typesig "m5<u>: (Hash<String, u>) →u"
```

In these annotations, we can determine the bindings of the type variables depending on whether the argument is a Array or Hash. In the case of m4, the type variable in the unused part of the union gets assigned the empty type %none.

Tuple types Ruby programmers often use Arrays both homogeneously for unbounded lists, and heterogeneously for fixed-size tuples. Like DRuby, rtc includes a special type Tuple⟨t1, ..., tn⟩ representing an array whose ith element has type ti [8]. Values of type Tuple can be manipulated using a subset of the Array methods that do not change the size of the array or the order of array elements. For example, Array#[] (element access) is allowed, but Array#push is not. Note that this is different than DRuby, which performs inference and thus begins by assuming every array literal is a Tuple and then promotes it to an Array if non-Tuple methods are used on it.

Instantiating proxy objects automatically Sometimes a programmer may want all instances of a given class to be annotated without having to explicitly call rtc_annotate. To achieve this, the programmer adds a call to rtc_autowrap in an annotated class definition. Classes that are subtypes of an auto-wrapping class are also auto-wrapping. Currently, auto-wrapping works only with non-parameterized classes.

```
a = [1,2,3]
b = a.rtc_annotate("Array<Object>")
# equiv. to b = Proxy.new(a, "Array<Object>")
n = 4.rtc_annotate("Fixnum")
b.push(n)
# the array now contains 1, 2, 3, and Proxy(4, "Object")
m = b[1]
# m is bound to the value Proxy(2, "Object")
```

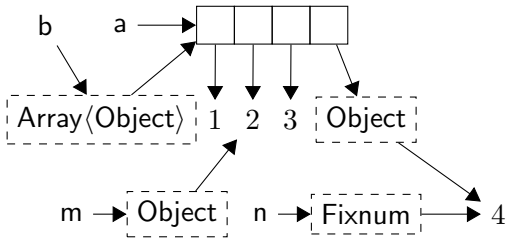

Figure 2.2: Illustration of proxy implementation.

2.2 Implementation

We have implemented rtc as a Ruby library. Rtc adds rtc_annotate, rtc_cast, and other key methods to the base Object and Class classes as appropriate, so they are available everywhere. When the programmer uses rtc_annotate to add a type to an existing object, rtc wraps the original object in a *proxy* that also contains the type. The proxy defines a method_missing method, which in Ruby receives a call when calling an undefined method on the object.[4] Calls to the proxy first ensure the arguments are of the appropriate type, then delegate to the original object, and finally check the return value's type before returning to the callee. The general idea of proxy wrapping is borrowed from Rubydust, though rtc does not perform constraint generation [1].

[4]Since calling methods via method_missing is slower than direct dispatch, we explicitly delegate some Object operations such as ==, class, or nil? due to their prevalence.

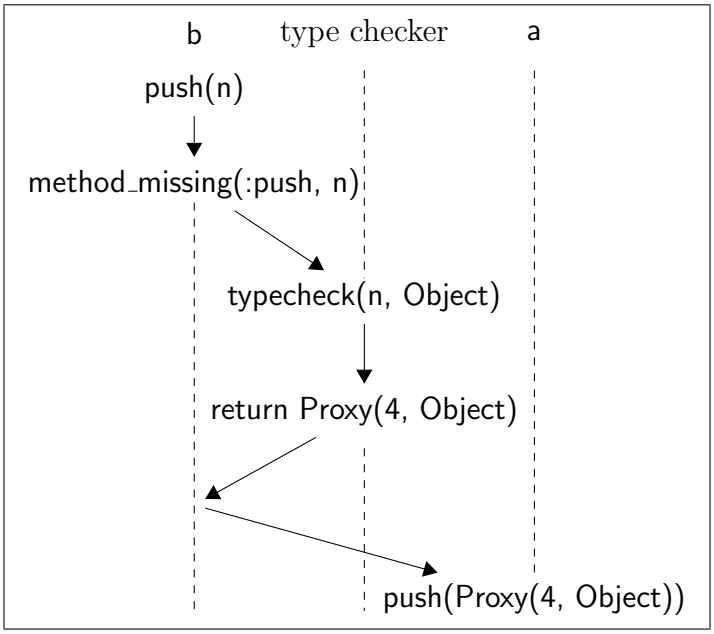

Figure 2.3: Illustration of proxy implementation - Sequence diagram for line 77

In more depth, consider the code at the top of Figure 2.2; the bottom part of the figure shows the objects resulting from this code. On line 74, we annotate the array from line 73 with the type Array⟨Object⟩. This annotation returns a new instance of the internal rtc class Proxy that holds both the underlying object and its type. Similarly, line 76 assigns a new Proxy to n.

Next consider the call on line 77; the sequence of events triggered by this call is shown in Figure 2.3. When push is invoked on the proxy object b, Proxy#method_missing is called with two arguments: :push, the name of the method, and n. Then method_missing checks the type of the argument by retrieving the type of the push method and comparing the type of the argument against the expected type. Rtc then rewraps the underlying object in a new Proxy with the formal argument type and returns the new proxy to method_missing. This ensures that the method must use the value according to the method's type signature (here, Object) instead of its possibly more specific

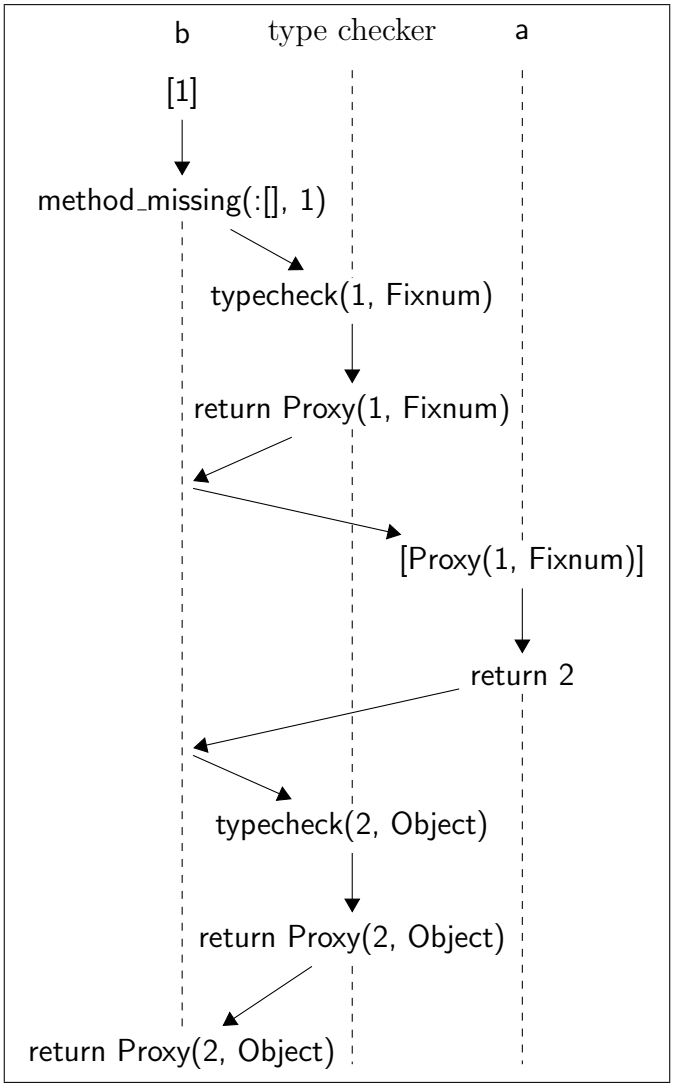

Figure 2.4: Illustration of proxy implementation - Sequence diagram for line 79

type (here, Fixnum). Finally, the rewrapped argument is passed to the underlying array's push method, which adds it to the end of the array a.

Next consider line 79, which sets m to b[1]; Figure 2.4 illustrates this call. As before, b's method_missing receives the call to []. This time, the argument 1 is a raw value, so rtc retrieves the value's class to derive its type. Since there is only one argument of class Fixnum, rtc infers that this call uses [] with the type (Fixnum) → Object. The type checking algorithm then wraps 1 in a Proxy with the type Fixnum before it is passed to the underlying object's [] method. Similarly, the unannotated value 2 in the array is wrapped in a Proxy with type Object (the inferred return type of []) before it is returned from the call.

As we have just seen, adding annotations to objects means that annotations get added to method arguments and results, even if those values were originally unannotated. Operations on those newly annotated values can add further annotations. Thus, the user need not annotate all objects explicitly to get wide type checking coverage, but rather can annotate just a few key objects to get the ball rolling.

Type checking blocks and procedures Thanks to Ruby's support for higher-order procedures, type checking blocks and procedures is straightforward. If the value to type check is a block, rtc first converts it to a procedure, and otherwise we use the procedure value directly. Next, rtc creates a new procedure that first checks the arguments, then calls the original procedure, and finally checks the return value from that call. This conversion is similar to the conversion performed by Findler and

Felleisen [6] to protect higher-order functions with contracts. If the original value is a procedure, then rtc uses the new procedure as the new value, and otherwise, rtc then converts the resulting procedure back into a block before use.

Type checking in method wrappers As we will explain shortly, we need to add another layer of interposition to track proxies on self and to support calls to native methods. Thus, rtc alters annotated classes to add a *method wrapper* layer within annotated classes themselves. To implement this alteration, rtc uses some low-level features of Ruby to rename methods in the original object to a mangled name. It then inserts a new method with the original name that delegates to the original method. Rtc adds a method instead of inserting a generic method_missing for improved performance. It is in these method wrappers that rtc performs type checking if a Proxy received the previous call.

In developing this implementation, we discovered one interesting quirk of Ruby. There are two ways to define new methods: using define_method, which takes a Proc object as an argument, or using eval. We found that methods created by the former mechanism are much slower to call than methods created by the latter. Thus, we use eval to create new methods although it is less elegant.

Due to this design, calling an annotated method in rtc entails two method interceptions: one in the Proxy and one in the method wrapper layer. To improve performance in the method wrapper layer, we directly call the (name-mangled) original methods of underlying objects that rtc uses internally in its type checking process.

Tracking proxies on self When a Proxy finally delegates to the underlying object's method, self will be bound to the underlying object rather than the Proxy. Thus, if that method in turn invokes other methods on self, without further work we will fail to type check those calls, since the receiver will not be a Proxy.

We solve this problem using the method wrapper layer. Internally, rtc maintains a stack of Proxys associated with each object. The method_missing of a Proxy pushes self (that is, the proxy) onto the stack associated with the wrapped object before delegating and pops the stack after normal or exceptional exit of the delegated method. When an annotated method is intercepted by the method wrapper layer, it also checks whether there is a Proxy on the stack. If so, it performs type checking using the type information contained in the topmost Proxy. This ensures type checking continues to occur for calls targeting self in annotated objects.

Handling methods that expect native values Certain methods of built-in types—particularly those implemented in native code—expect their arguments to be objects of an appropriate class, and passing in Proxys instead causes those operations to fail. Thus, typesig optionally takes an :unwrap argument that is an array of argument positions from which the proxy must be removed before calling the method. For example, we can annotate the + operation on Fixnum to unwrap its argument:

```
81  typesig "'+':(Fixnum)→Fixnum", :unwrap⇒[0]
```

In the method wrapper layer, we remove proxies as specified by :unwrap before calling the original method.

Handling false and nil A related problem is that boolean comparisons and conditional expressions, which cannot be intercepted in Ruby, treat false and nil as false and all other values as true. Thus, wrapping false or nil in a proxy would cause them to be treated as true, yielding incorrect results. As a result, we do not wrap either these values in proxies during type checking.

Non-strict mode As discussed in Section 2.1, rtc checks that raw objects have the correct type whenever they are annotated; for container classes like Array, this check involves iterating over the contents, which can be quite expensive.

Thus, rtc includes a *non-strict mode* in which this iteration is omitted. That is, in non-strict mode, when raw values are annotated only the type constructor is checked for compatibility, but not the type parameters. For example:

```
82 # non−strict mode
83 [1,2,3]. rtc_annotate("Fixnum")          # error
84 [1,2,3]. rtc_annotate ("Array<String>") # ok
```

While non-strict mode does not catch errors as soon as possible, errors are caught on uses of the contained values. For example, consider the following code:

In non-strict mode, the argument to sum is accepted although the contents do not match the expected type. However, rtc deduces from its annotation that the block argument to each accepts type Fixnum. When the first element of the array, of type String, is passed to the block, the block wrapper checks it against type Fixnum and reports an error.

In Section 2.3, we compare the performance of strict and non-strict modes. As the latter is significantly faster that than former, non-strict mode is enabled by

```
85 # non-strict mode
86 class Statistics
87   rtc_annotated
88   typesig "sum: (Array<Fixnum>) →Fixnum"
89   def sum(input)
90     total = 0
91     input.each { |elem|
92       total += elem
93     }
94     total
95   end
96 end
97 Statistics.new.rtc_annotate("Statistics").sum(["1", "2", "3"])
```

default. However, the programmer may opt-in to stricter type checking by setting the global variable $RTC_STRICT to true.

2.3 Evaluation

We performed an initial evaluation of rtc on a set of Ruby programs and libraries that we retrofitted with rtc types. Figure 2.5 summarizes the results. The subject programs are as follows:

- Sudoku: an implementation of Norvig's algorithm for solving Sudoku puzzles.
- Ascii85: a program for encoding/decoding Adobe's binary-to-text encodings of the same name.
- ministat: a library that computes statistical info such as mode, median, mean, variance, etc.
- finitefield: an implementation of finite field arithmetic.
- hebruby: a Hebrew data conversion program.
- set: Ruby's set library and its associated test cases.

program	time (s) unann	n-strict	strict	annot. meths	unann meths	annot vals	Features Tuple	{·}	(τ)	∪	∩	∀
Sudoku-1.4	0.04	5.34	7.58	8	1	10	0	0	2	5	0	0
Ascii85-1.0.2	0.02	0.05	0.05	2	1	0	0	0	0	0	0	0
ministat-1.0.0	<0.01	0.30	0.56	13	1	0	0	0	0	0	0	0
finitefield-0.1.0	<0.01	0.02	0.02	10	1	0	0	0	0	0	0	0
hebruby-2.0.2	<0.01	0.12	0.12	19	1	0	1	0	0	1	0	0
RDS-1.0.0	<0.01	0.01	0.01	7	2	3	0	0	0	3	0	7
library												
Array	–	–	–	71	4	–	0	28	–	7	35	18
Hash	–	–	–	38	2	–	4	12	–	5	9	8
Set	–	–	–	21	13	–	0	7	–	0	2	7

Tuple = Tuple types, {·} = block types, (τ) = rtc_cast
∪ = union types, ∩ = intersection types, ∀ = polymorphic types

Figure 2.5: Summary of evaluation results

- Ruby Data Structures (abbreviated RDS): a library of common data structures. We annotated two classes, SinglyLinkedList and SinglyLinkedListElement.

The first five of these programs come from the Rubydust benchmark suite [1]. We also tried to annotate the other three Rubydust benchmarks, but those programs fail to run under the latest version of Ruby, which rtc requires.

In addition to annotating the subject programs, we also annotated the built-in Array, Hash, and Set libraries. These particular libraries were chosen because they are the basis for most user-defined data structures and they saw the heaviest use in the programs we used for our evaluation.

Next we report on the overhead of rtc, which rtc features were used for the subject programs, and the ease of the conversion process.

Efficiency The first three columns of the Figure 2.5 report the running times for the program's test suite on the original program; on the annotated program under non-strict mode; and on the annotated program under strict mode. While the performance overhead of rtc is relatively large, the test suites run quite rapidly in

most cases, suggesting that rtc is practical in many testing scenarios. As mentioned in section 2.2, rtc creates a wrapper layer that all calls must go through whether there is an active proxy or not. This extra level of indirection is the main source of the overhead in rtc due to the inefficiency of method calls in Ruby.

The program with the most substantial overhead is Sudoku. This program makes extensive use of large arrays and hashes, and so the overhead of rtc's method interception has a large cost. To partially address this issue, rtc can be disabled in production by setting by the RTC_DISABLE environment variable to a non-empty value. When rtc is disabled, no wrappers are created by calls to typesig. In addition, annotations on objects via rtc_annotate and rtc_cast become no-ops. That is, instead of returning a new proxy object, they simply return self. This enables the programmer to use rtc in a test environment where some overhead may be acceptable and then disable rtc in the field.

Rtc features The middle three columns of Figure 2.5 count the number of unannotated methods, annotated methods, and explicit annotations of values we added. In the subject programs, the only unannotated method was initialize, the constructor. Since the receiver of initialize is always a newly created, and hence raw, object, rtc will never type check an initialize call. In the future, we plan to investigate other policies for constructors.

The only subject program for which we needed explicit rtc_annotate calls was Sudoku. These annotations were for several large arrays built during initialization, and they improved the performance of rtc in strict mode since otherwise rtc would

repeatedly iterate over those arrays to infer their types at each use.

In the library classes, we were able to annotate almost all of the methods for Array and Hash. There were several methods, however, that have no reasonable type annotation in rtc. For example, Array#flatten returns a new array in which arbitrary depth nestings of array have been removed from the receiver. Even worse, Array#flatten! does the same, but mutates the receiver object. We leave these as unannotated, so they may be used but are not type checked. Unannotated methods in the Set class include Set#flatten and methods that use the Enumerable class, which is a mixin for collection classes; rtc currently does not support mixins.

The rightmost columns of Figure 2.5 list how often various typing features of rtc were used. To count uses of polymorphic types, we counted how many classes or methods have annotations that bind type variables. The most commonly used features in the subject programs are union and polymorphic types, and the most commonly used features in the Ruby standard libraries are intersection and block types. There were very few uses overall of tuple types and rtc_cast.

The annotation process We found the process of annotating the subject programs to be relatively straightforward: we examined their code, looked for program invariants assumed by the original authors, and turned those invariants into type annotations. Although this was somewhat time consuming for us, we expect the original authors of the methods would be faster at this process.

We often had to iterate the annotation process as we found mistakes in our typesigs. The most common errors we made fall into a few groups. For some pro-

grams, we missed edge cases in methods, such as sometimes returning false, and so would initially annotate a method with a subset of its possible return values. Similarly, we sometimes missed an arm of an intersection type. Finally, we sometimes forgot to cast a value typed with a union type to a more specific type after the value was tested. In all cases, we found our errors immediately upon running the test suites under rtc.

Not all type errors were due to our mistakes, however. The Sudoku solver contains the following (correct) annotations:

```
98  typesig "search: (...) → %false or Hash<String,String>)")
99  typesig " string_solution : (Hash<String,String>) → String")
```

The search method returns false if the given puzzle is impossible to solve, while string_solution assumes that it is given a valid puzzle solution. In the test suite, the return of search is fed directly into string_solution without checking for false. While an error due to this mismatch never happens in the test suite because all its puzzles are solvable, rtc appropriately raises a type error.

2.4 Related Work

As discussed in the introduction, rtc builds on the Rubydust system of An et al. [1]; we even reuse some of the same code base, specifically the type language parser and some of the proxy-related code. The key difference is that rtc is a type checking system, whereas Rubydust performs type inference. The addition of checking introduces several new concerns: adding explicit annotations (rtc_annotate) and

type casts (rtc_cast); inferring types of raw values passed to annotated positions; and making control of type checking finer grained, that is, driven by annotated objects, rather than by annotations on classes as in Rubydust. Rtc also supports some features that Rubydust does not, including lock types and tracking type annotations on self. Rubydust does not do the latter because its coarse-grained distinction between typed and untyped code means it does not matter whether self is proxied. Finally, perhaps the most important difference from a usability perspective is that rtc type errors are reported as soon as they occur, whereas Rubydust generates constraints and only solves them at the end of execution. Thus, it may be harder in Rubydust to understand reported errors.

Several researchers have proposed adding static types and static type inference to various dynamically typed languages, including Ruby [7,8], Python [4,10,15], and JavaScript [3, 13] among others. Similarly, gradual type systems [11] like those for Scheme [14] and Thorn [5] pair a dynamically typed language with a sister, statically typed language. The typed and untyped parts of a program are allowed to interact without breaking the invariants of the typed language. All of these systems perform static analysis, whereas rtc is a library that operates purely at run time. One advantage of rtc's approach is that it does not require maintaining a Ruby frontend, which the Rubydust authors have pointed out as problematic [2]. Another advantage of rtc is that because it operates at run time, rtc only observes realizable execution paths through the target program and can easily operate in the presence of dynamic features such as eval, reflective method invocation, and method_missing.

Rtc's dynamic implementation is inspired by research into contract systems.

Existing contract systems for Ruby are limited to "design by contract" [9] systems, which annotate classes with preconditions, postconditions, and invariants that are simple assertions checked only on method entry and method exit. Rtc's dynamic checks are closer to those provided by higher-order contracts [6, 12]. Like higher-order contract systems, rtc wraps method arguments and results with proxies that stay with those objects as they flow through the program. This enables rtc not just to enforce preconditions and postconditions, but also to check that the type of a parameter is adhered to within the body of a method and that the type of a return value is respected long after the method has returned.

2.5 Conclusion

We present rtc, a Ruby library that adds type checking at method call boundaries. Rtc uses proxy objects to wrap regular objects with annotated types and only type checks annotated methods on classes and proxied objects. Our experimental results suggest that rtc is a practical, useful system. In the future, we plan to apply rtc to Ruby on Rails programs, and explore extending its type checking capability to reason about some of the complex invariants of the Rails framework.

Chapter 3: Just-in-Time Static Type Checking for Dynamic Languages

The last chapter introduced rtc, a purely static type checking tool for Ruby that brings the benefits of static typing to a dynamic language. Although rtc is an effective tool for increasing type safety, it has trouble dealing with metaprogramming. In this chapter, we introduce Hummingbird [35], a run-time static type checking tool for Ruby that addresses the metaprogramming issue.

Attribution and Acknowledgements

The work described in this chapter was previously published in the 2016 Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, with co-author Jeffrey S. Foster. I implemented and evaluated Hummingbird, and the design and proof of correctness was shared across the authors.

3.1 Overview

We begin our presentation by showing some uses of metaprogramming in Ruby and the corresponding Hummingbird type checking process. The examples below

```
1  class Talk < ActiveRecord::Base
2    belongs_to :owner, :class_name => "User"
3    ....
4    type :owner?, "(User) -> %bool"
5    def owner?(user)
6      return owner == user
7  end end
8
9  module ActiveRecord::Associations::ClassMethods
10   pre(:belongs_to) do |*args|
11     hmi = args[0]
12     options = args[1]
13     hm = hmi.to_s
14     cn = options[:class_name] if options
15     hmu = cn ? cn : hm.singularize.camelize
16     type hm.singularize, "() -> #{hmu}"
17     type "#{hm.singularize}=", "(#{hmu}) ->#{hmu}"
18     true
19 end end
```

Figure 3.1: Ruby on Rails Metaprogramming.

are from the experiments in Section 3.4.

Rails Associations. The top of Figure 3.1 shows an excerpt from the *Talks* Rails app. This code defines a class Talk that is a *model* in Rails, meaning an instance of Talk represents a row in the talks database table. The change in case and pluralization here is not an accident—Rails favors "convention over configuration," meaning many relationships that would otherwise be specified via configuration are instead implicitly expressed by using similar or the same name for things.

In this app, every talk is owned by a user, which in implementation terms means a Talk instance has a foreign key owner_id indicating the owner, which is an instance of class User (not shown). The existence of that relationship is defined on line 2. Here it may look like belongs_to is a keyword, but in fact it is simply a method

call. The call passes the *symbol* (an interned string) :owner as the first argument, and the second argument is a hash that maps symbol :class_name to string "User".

Now consider the owner? method, defined on line 5. Just before the method, we introduce a type annotation indicating the method takes a User and returns a boolean. Given such an annotation, Hummingbird's goal is to check whether the method body has the indicated type.[1] This should be quite simple in this case, as the body of owner? just calls no-argument method owner and checks whether the result is equal to user.

However, if we examine the remaining code of Talk (not shown), we discover that owner is not defined anywhere in the class! Instead, this method is created at run-time by belongs_to. More specifically, when belongs_to is called, it defines several convenience methods that perform appropriate SQL queries for the relationship [30], in this case to get the User instance associated with the Talk's owner. Thus, as we can see, it is critical for Hummingbird to handle such dynamically created methods even to type check simple Rails code.

Our solution is to instrument belongs_to so that, just as it creates a method dynamically, it also creates method type signatures dynamically. The code on lines 9–19 of Figure 3.1 accomplishes this. Hummingbird is built on RDL, a Ruby contract system for specifying pre- and postconditions [33, 41]. The precondition is specified via a *code block*—an anonymous function (i.e., a lambda) delimited by do...end—passed to pre. Here the code block trivially returns true so the precondition is

[1] In practice type takes another argument to tell Hummingbird to type check the body, in contrast to library and framework methods whose types are trusted. We elide this detail for simplicity.

always satisfied (last line) and, as a side effect, creates method type annotations for belongs_to.

In more detail, hmi is set to the first argument to belongs_to, and options is either nil or the hash argument, if present. (Here hm is shorthand for "has many," i.e., since the Talk belongs to a User, the User has many Talks.) Then hmu is set to either the class_name argument, if present, or hmi after singularizing and camel-casing it. Then type is called twice, once to give a type to a getter method created by belongs_to, and once for a setter method (whose name ends with =). Notation #{e} inside a string evaluates the expression e and inserts the result in the string. In this particular case, these two calls to type evaluate to

```
type "owner", "() → User"
type "owner=", "(User) → User"
```

Now consider executing this code. When Talk is loaded, belongs_to will be invoked, adding those type signatures to the class. Then when owner? is called, Hummingbird will perform type checking using currently available type information, and so it will be able to successfully type check the body. Moreover, notice this approach is very flexible. Rails does not require that belongs_to be used at the beginning of a class or even during this particular class definition. (In Ruby, it is possible to "re-open" a class later on and add more methods to it.) But no matter where the call occurs, it must be before owner? is called so that owner is defined. Thus in this case, Hummingbird's typing strategy matches well with Ruby's semantics.

```
1  module Rolify::Dynamic
2    def define_dynamic_method(role_name, resource)
3      class_eval do
4        define_method("is_#{role_name}?".to_sym) do
5          has_role?("#{role_name}")
6        end if !method_defined?("is_#{role_name}?".to_sym)
7        ...
8    end end
9
10   pre :define_dynamic_method do |role_name, resource|
11     type "is_#{role_name}?", "() → %bool"
12     true
13 end end
14
15 class User; include Rolify::Dynamic end
16 user = User.first
17 user.define_dynamic_method("professor", ...)
18 user.define_dynamic_method("student", ...)
19 user.is_professor?
20 user.is_student?
```

Figure 3.2: Methods Dynamically Created by User Code.

Type Checking Dynamically Created Methods. In the previous example, we trusted Rails to dynamically generate code matching the given type signature. Figure 3.2 shows an example, extracted from *Rolify*, in which user code dynamically generates a method. The first part of the figure defines a module (aka mixin) with a two-argument method define_dynamic_method. The method body calls define_method to create a method named using the first argument, as long as that method does not exist (note the postfix if on line 6). Similarly to earlier, line 10 adds a precondition to define_dynamic_method that provides an appropriate method type. (We do not check for a previous type definition since adding the same type again is harmless.)

The code starting at line 15 uses the module. This particular code is not from our experiment but is merely for expository purposes. Here we (re)open class User

```
1  Transaction = Struct.new(:type, :account_name, :amount)
2  class ApplicationRunner
3    def process_transactions
4      @transactions.each do |t|
5        name = t.account_name
6        ...
7    end ... end
8     field_type :@transactions, "Array<Transaction>"
9  end
10
11 class Struct
12   def self.add_types(*types)
13     members.zip(types).each {|name, t|
14       self.class_eval do
15         type name, "() → #{t}"
16         type "#{name}=", "(t) →#{t}"
17       end
18     }
19   end
20 end
21 Transaction.add_types("String", "String", "String")
```

Figure 3.3: Type Signatures for Struct.

and mix in the module. Then we create a user; call define_dynamic_method twice; and then call the generated methods is_professor? and is_student?.

In this case, since the generated methods have type annotations and are in user code, Hummingbird type checks their bodies when they are called, just like any other user-defined method with a type. For example, consider the call to is_professor?, which is given type () → %bool. At the call, Hummingbird type checks the code block at line 4 and determines that it has no arguments and that its body returns a boolean, i.e., it type checks.

User-provided Type Signatures. In the examples so far, the types for dynamically created methods could be determined automatically. However, consider

Figure 3.3, which shows an excerpt from *CCT* that uses Struct from the Ruby core library. Line 1 creates a new class, instances of which are defined to have getters type, account_name, and amount, and setters type=, account_name=, and amount=. The process_transactions method iterates through instance field @transactions (whose type is provided on line 8), and calls the account_name method of each one.

From line 1 we know the account_name method exists, but we do not know its type. Indeed, a "struct field" can hold any type by default. Thus, to fully type check the body of process_transactions, we need more information from the programmer to specify the type of account_name.

The bottom part of Figure 3.3 defines a new method, add_types, that the programmer can call to indicate desired struct field types. The types are given in the same order as the constructor arguments, and the body of add_types uses zip to pair up the constructor arguments (retrieved via members) and the types, and then iterates through the pairs, creating the appropriate type signatures for the getters and setters. The last line of the figure uses add_types to create type signatures for this example, allowing us to type check process_transactions when it is called.

In this particular case, we could have individually specified type signatures for the methods of Transaction. However, because Hummingbird lets programmers write arbitrary Ruby programs to generate types, we were able to develop this much more elegant solution.

$$
\begin{array}{lrcl}
\text{values} & v & ::= & \mathsf{nil} \mid [A] \\
\text{expressions} & e & ::= & v \mid x \mid \mathsf{self} \mid x = e \mid e; e \mid A.\mathsf{new} \\
 & & \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid e.m(e) \\
 & & \mid & \mathsf{def}\ A.m = b \mid \mathsf{type}\ A.m : \tau_m \\
\text{premths} & b & ::= & \lambda x.e \\
\text{val typs} & \tau & ::= & A \mid \mathsf{nil} \\
\text{mth typs} & \tau_m & ::= & \tau \to \tau
\end{array}
$$

$$x \in \text{var ids} \quad m \in \text{mth ids} \quad A \in \text{cls ids}$$

$$
\begin{array}{lrcl}
\text{dyn env} & E & : & \text{var ids} \to \text{vals} \\
\text{dyn cls tab} & DT & : & \text{cls ids} \to \text{mth ids} \to \text{premths} \\
\text{contexts} & C & ::= & \Box \mid x = C \mid C.m(e) \mid v.m(C) \\
 & & \mid & C; e \mid \mathsf{if}\ C\ \mathsf{then}\ e\ \mathsf{else}\ e \\
\text{stack} & S & ::= & \cdot \mid (E, C) :: S \\
\text{type env} & \Gamma, \Delta & : & \text{var ids} \to \text{val typs} \\
\text{type tab} & TT & : & \text{cls ids} \to \text{mth ids} \to \text{mth typs} \\
\text{cache} & X & ::= & \text{cls ids} \to \text{mth ids} \to \mathcal{D}_M \times \mathcal{D}_\leq \\
\text{typ chk deriv} & \mathcal{D}_M & ::= & TT \vdash \langle \Gamma, e \rangle \Rightarrow \langle \Gamma', \tau \rangle \\
\text{subtyp deriv} & \mathcal{D}_\leq & ::= & \tau_1 \leq \tau_2
\end{array}
$$

Figure 3.4: Source Language and Auxiliary Definitions.

3.2 Formalism

We formalize Hummingbird using the core, Ruby-like language shown at the top of Figure 3.4. *Values* v include nil, which can be treated as if it has any type, and $[A]$, which is an instance of class A. Note that we omit both fields and inheritance from our formalism for simplicity, but they are handled by our implementation.

Expressions e include values, variables x, the special variable self, assignments $x = e$, and sequencing $e; e$. Objects are created with $A.\mathsf{new}$. Conditional $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$ evaluates to e_2 unless e_1 evaluates to nil, in which case it evaluates to e_3. Method invocation $e_1.m(e_2)$ is standard, invoking the m method based on the run-time type of e_1.

$\boxed{TT \vdash \langle \Gamma, e \rangle \Rightarrow \langle \Gamma', \tau \rangle}$

$$\text{(TNil)}\ \frac{}{TT \vdash \langle \Gamma, \mathsf{nil} \rangle \Rightarrow \langle \Gamma, \mathsf{nil} \rangle} \qquad \text{(TObject)}\ \frac{}{TT \vdash \langle \Gamma, [A] \rangle \Rightarrow \langle \Gamma, A \rangle} \qquad \text{(TSelf)}\ \frac{}{TT \vdash \langle \Gamma, \mathsf{self} \rangle \Rightarrow \langle \Gamma, \Gamma(\mathsf{self}) \rangle}$$

$$\text{(TVar)}\ \frac{}{TT \vdash \langle \Gamma, x \rangle \Rightarrow \langle \Gamma, \Gamma(x) \rangle} \qquad \text{(TSeq)}\ \frac{TT \vdash \langle \Gamma, e_1 \rangle \Rightarrow \langle \Gamma_1, \tau_1 \rangle \quad TT \vdash \langle \Gamma_1, e_2 \rangle \Rightarrow \langle \Gamma_2, \tau_2 \rangle}{TT \vdash \langle \Gamma, e_1; e_2 \rangle \Rightarrow \langle \Gamma_2, \tau_2 \rangle}$$

$$\text{(TAssn)}\ \frac{TT \vdash \langle \Gamma, e \rangle \Rightarrow \langle \Gamma', \tau \rangle}{TT \vdash \langle \Gamma, x = e \rangle \Rightarrow \langle \Gamma'[x \mapsto \tau], \tau \rangle} \qquad \text{(TNew)}\ \frac{}{TT \vdash \langle \Gamma, A.\mathsf{new} \rangle \Rightarrow \langle \Gamma, A \rangle}$$

$$\text{(TDef)}\ \frac{}{TT \vdash \langle \Gamma, \mathsf{def}\ A.m = \lambda x.e \rangle \Rightarrow \langle \Gamma, \mathsf{nil} \rangle} \qquad \text{(TType)}\ \frac{}{TT \vdash \langle \Gamma, \mathsf{type}\ A.m : \tau_m \rangle \Rightarrow \langle \Gamma, \mathsf{nil} \rangle}$$

$$\text{(TApp)}\ \frac{TT \vdash \langle \Gamma, e_0 \rangle \Rightarrow \langle \Gamma_0, A \rangle \quad TT \vdash \langle \Gamma_0, e_1 \rangle \Rightarrow \langle \Gamma_1, \tau \rangle \quad TT(A.m) = \tau_1 \to \tau_2 \quad \tau \leq \tau_1}{TT \vdash \langle \Gamma, e_0.m(e_1) \rangle \Rightarrow \langle \Gamma_1, \tau_2 \rangle}$$

$$\text{(TIf)}\ \frac{TT \vdash \langle \Gamma, e_0 \rangle \Rightarrow \langle \Gamma', \tau \rangle \quad TT \vdash \langle \Gamma', e_1 \rangle \Rightarrow \langle \Gamma_1, \tau_1 \rangle \quad TT \vdash \langle \Gamma', e_2 \rangle \Rightarrow \langle \Gamma_2, \tau_2 \rangle}{TT \vdash \langle \Gamma, \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rangle \Rightarrow \langle \Gamma_1 \sqcup \Gamma_2, \tau_1 \sqcup \tau_2 \rangle}$$

Figure 3.5: Type Checking System.

Expression def $A.m = \lambda x.e$, defines method m of class A as taking argument x and returning e. (We refer to $\lambda x.e$ as a *premethod*.) This form allows methods to be defined anywhere during execution, thus it combines the features of Ruby's def and define_method. As in Ruby, if $A.m$ is already defined, def overwrites the previous definition. The def expression itself evaluates to nil.

Finally, expression type $A.m : \tau \to \tau'$ asserts that method m of class A has domain type τ and range type τ'. Types may be either classes A or nil, the type of expression nil. The type expression overwrites the previous type of $A.m$, if any. Like Hummingbird, there is no ordering dependency between def and type—the only requirement is that a method's type must be declared by the time the method is

called. The type expression itself evaluates to nil.

Type Checking. Figure 3.5 gives the static type checking rules. As in Hummingbird, static type checking is performed at run time at method entry—thus these rules will be invoked as a subroutine by the dynamic semantics (below). The bottom part of Figure 3.4 defines the sets and maps used in this figure and in the dynamic semantics.

In these rules, TT is a *type table* mapping class and method ids $A.m$ to their corresponding types, as declared by type, and Γ is a *type environment* mapping local variables to their types. These rules prove judgments of the form $TT \vdash \langle \Gamma, e \rangle \Rightarrow \langle \Gamma', \tau \rangle$, meaning with type table TT, in type environment Γ, expression e has type τ, and after evaluating e, the new type environment is Γ'. Using an "output" type environment Γ' allows us to build a flow-sensitive type system, in which variables' types can change at assignments. Note there is no output TT because the type table does not change during static type checking—it only changes as the program is executed by the dynamic semantics.

The type rules are largely standard. (TNil) and (TObject) give nil and instances the obvious types. (TSelf) and (TVar) give self and local variables their types according to the type environment. Since none of these four expressions updates the state, the output type environment is the same as the input environment.

(TSeq) types sequencing, threading the type environment from the output of e_1 to the input of e_2. (TAssn) types an assignment, updating the output type environment to bind the assigned variable x to the type of the right-hand side. (TNew)

types object creation in the obvious way. (TDef) trivially type checks method definitions. Notice we do not type check the method body; that will happen at run time when the method is actually called. (TType) type checks a type expression, which has no effect during type checking. Such expressions are only evaluated at run-time, when they update the type table (see below).

One consequence of (TType) is that our type system forbids typing a method and then immediately calling it in the same method body. For example, the following method body would fail to type check:

```
1 def A.m = λx.
2   def B.m = ...;    # define B.m
3   type B.m : ...;   # give B.m a type
4   B.new.m           # type error! B.m not in type table
```

Here we type check A.m's body at the first call to it, so the type expression has not been run—and hence has not bound a type to B.m—yet. Thus it is a type error to invoke B.m in the method body.

While we could potentially solve this problem with a more complex type system, in our experience (Section 3.4) we have not needed such a feature.

Next, (TApp) types method invocation $e_0.m(e_1)$, where we look up the method's type in TT based on the compile-time type of e_0. (Note that since there is no inheritance, we need not search the inheritance hierarchy to find the type of $A.m$.) Here subtyping is defined as nil $\leq A$ and $A \leq A$ for all A. Thus, as is standard in languages with nil, the type system may accept a program that invokes a non-existent method of nil even though this is a run-time error. However, notice that if e_0 evaluates to a non-nil value, then (TApp) guarantees e_0 has method m.

$\boxed{\langle X, TT, DT, E, e, S\rangle \to \langle X', TT', DT', E', e', S'\rangle}$

(ESelf)
$\langle X, TT, DT, E, \mathsf{self}, S\rangle \quad \to \quad \langle X, TT, DT, E, E(\mathsf{self}), S\rangle$
(EVar)
$\langle X, TT, DT, E, x, S\rangle \quad \to \quad \langle X, TT, DT, E, E(x), S\rangle$
(EAssn)
$\langle X, TT, DT, E, x = v, S\rangle \quad \to \quad \langle X, TT, DT, E[x \mapsto v], v, S\rangle$
(ENew)
$\langle X, TT, DT, E, A.\mathsf{new}, S\rangle \quad \to \quad \langle X, TT, DT, E, [A], S\rangle$
(ESeq)
$\langle X, TT, DT, E, (v; e_2), S\rangle \quad \to \quad \langle X, TT, DT, E, e_2, S\rangle$
(EIfTrue)
$\langle X, TT, DT, E, \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, S\rangle \quad \to \quad \langle X, TT, DT, E, e_1, S\rangle$ if $v \neq \mathsf{nil}$
(EIfFalse)
$\langle X, TT, DT, E, \mathsf{if\ nil\ then}\ e_1\ \mathsf{else}\ e_2, S\rangle \quad \to \quad \langle X, TT, DT, E, e_2, S\rangle$
(EDef)
$\langle X, TT, DT, E, \mathsf{def}\ A.m = \lambda x.e, S\rangle \quad \to \quad \langle X \backslash A.m, TT, DT[A.m \mapsto \lambda x.e], E, \mathsf{nil}, S\rangle$
(EType)
$\langle X, TT, DT, E, \mathsf{type}\ A.m : \tau_m, S\rangle \quad \to \quad \langle (X \backslash A.m)[TT'], TT', DT, E, \mathsf{nil}, S\rangle$
$TT' = TT[A.m \mapsto \tau_m]$ and $A.m \notin \mathrm{TApp}(S)$

(EAppMiss)
$\langle X, TT, DT, E, C[v_1.m(v_2)], S\rangle \quad \to \quad \langle X', TT, DT, [\mathsf{self} \mapsto v_1, x \mapsto v_2], e, (E, C) :: S\rangle$
if $A.m \notin dom(X)$ and $v_1 = [A]$ and $DT(A.m) = \lambda x.e$
and $TT(A.m) = \tau_1 \to \tau_2$ and $\mathsf{type\_of}(v_2) \leq \tau_1$ and
$\mathcal{D}_M = (TT \vdash \langle [x \mapsto \tau_1, \mathsf{self} \mapsto A], e\rangle \Rightarrow \langle \Gamma', \tau\rangle)$ holds and $\mathcal{D}_{\leq} = (\tau \leq \tau_2)$ holds and
$X' = X[A.m \mapsto (\mathcal{D}_M, \mathcal{D}_{\leq})]$

(EAppHit)
$\langle X, TT, DT, E, C[v_1.m(v_2)], S\rangle \quad \to \quad \langle X, TT, DT, [\mathsf{self} \mapsto v_1, x \mapsto v_2], e, (E, C) :: S\rangle$
if $A.m \in dom(X)$ and $v_1 = [A]$ and $DT(A.m) = \lambda x.e$
and $TT(A.m) = \tau_1 \to \tau_2$ and $\mathsf{type\_of}(v_2) \leq \tau_1$

(ERet)
$\langle X, TT, DT, E', v, (E, C) :: S\rangle \quad \to \quad \langle X, TT, DT, E, C[v], S\rangle$

(EContext)
$$\frac{\langle X, TT, DT, E, e, S\rangle \to \langle X', TT', DT', E', e', S'\rangle \quad \nexists v_1, v_2, e' \,.\, e = (v_1.m(v_2)) \vee e = v_1 \vee e = C[e']}{\langle X, TT, DT, E, C[e], S\rangle \to \langle X', TT', DT', E', C[e'], S'\rangle}$$

Figure 3.6: Dynamic Semantics

Finally, (TIf) types conditionals. Like Ruby, the guard e_0 may have any type. The type of the conditional is the least upper bound of the types of the two branches, defined as $A \sqcup A = A$ and $\mathsf{nil} \sqcup \tau = \tau \sqcup \mathsf{nil} = \tau$. The output environment of the conditional is the least upper bound of the output environments of the branches, defined as $(\Gamma_1 \sqcup \Gamma_2)(x) = \Gamma_1(x) \sqcup \Gamma_2(x)$ if $x \in dom(\Gamma_1) \wedge x \in dom(\Gamma_2)$ and $(\Gamma_1 \sqcup \Gamma_2)(x)$ is undefined otherwise.

Dynamic Semantics. Figure 3.6 gives a small-step dynamic semantics for our language. The semantics operates on *dynamic configurations* of the form $\langle X, TT, DT, E, e, S \rangle$. The first two components are the key novelties to support run-time static type checking. X is a *cache* mapping $A.m$ to the type checking proofs for its method body (more details below). TT is the *type table*, which is updated at run time by calls to type. The last four components are standard. DT is a *dynamic class table* mapping $A.m$ to its premethod. E is the *dynamic environment* mapping local variables to values. e is the expression being reduced. Lastly, S is a *stack* of pairs (E, C), where E is the dynamic environment and C is the *evaluation context* (defined in the usual way) at a call site. The semantics pushes onto the stack at calls and pops off the stack at returns.

The first seven rules in the semantics are standard. (ESelf) and (EVar) evaluate self and variables by looking them up in the environment. (EAssn) binds a variable to a value in the environment. Notice that, like Ruby, variables can be written without first declaring them, but it is an error to try to read a variable that has not been written. (ENew) creates a new instance. Note that since objects do

not have fields, we do not need a separate heap. (ESeq) discards the left-hand side of a sequence if it has been fully evaluated. (EIfTrue) reduces to the true branch if the guard is non-nil, and (EIfFalse) reduces to the false branch otherwise.

The next four rules are the heart of just-in-time static type checking. Our goal is to statically type check methods once at the first call, and then avoid rechecking them unless something has changed. To formalize this notion, we define the cache X as a map from $A.m$ to a pair of typing derivations $(\mathcal{D}_M, \mathcal{D}_{\leq})$. Here $\mathcal{D}_M$ is a type checking derivation from Figure 3.5 for the body of $A.m$, and $\mathcal{D}_{\leq}$ is a subtyping judgment showing that the type of e is a subtype of the declared return type. We need $\mathcal{D}_{\leq}$ because our type system is syntax-directed and hence does not include a standalone subsumption rule. (EDef) reduces to nil, updating the dynamic class table to bind $A.m$ to the given premethod along the way. Recall that we allow a method to be redefined with def. Hence we need to *invalidate* anything in the cache relating to $A.m$ so that $A.m$ will be checked the next time it is called. More precisely:

Definition 1 (Cache invalidation)**.** *We write $X \backslash A.m$ to indicate a new cache that is the same as X, except $A.m$ has been invalidated, meaning:*

1. *Any entries with $A.m$ as the key are removed.*

2. *Any entries with a $\mathcal{D}_M$ that apply* (TApp) with $A.m$ are removed.

Thus, in (EDef), the output cache is the same as the input cache but with $A.m$ invalidated.

(EType) also reduces to nil, updating the type table to be TT', which is the same as TT but with new type information for $A.m$. As with (EDef), we invalidate $A.m$ in the cache. However, there is a another subtlety. Recall that cached typing derivations $\mathcal{D}_M$ include the type table TT. This is potentially problematic, because we are changing the type table to TT'. However, cache invalidation removes any derivations that refer to $A.m$. Hence, cached type derivations that use TT can safely use TT'. Formally, we define:

Definition 2 (Cache upgrading)**.** *We write $X[TT']$ to indicate a new cache that is the same as X, except the type table in every derivation is replaced by TT'.*

Thus, in (EType), the output cache is upgraded to the new type table after invalidation.

The next two rules use the type cache. Both rules evaluate a method call in a context, written $C[v_1.m(v_2)]$; we will discuss the other rule for contexts shortly. In both rules, the receiver v_1 is a run-time object $[A]$. (EAppMiss) applies when $A.m$ is not in the cache. In this case, we look up the type of $A.m$ in TT, yielding some type $\tau_1 \to \tau_2$. We type check the method body e in an environment in which formal variable x is bound to τ_1 and self is bound to A, yielding a derivation $\mathcal{D}_M$. We check that the resulting type τ of e is a subtype of the declared type e_2, with subtyping derivation $\mathcal{D}_{\leq}$. Finally, we check that the run-time type of v_2—defined as $\mathsf{type\_of}(\mathsf{nil}) = \mathsf{nil}$ and $\mathsf{type\_of}([A]) = A$—is a subtype of τ_1. If all this holds, then it is type-safe to call the method. Hence we update the cache with the method's typing derivations and start evaluating the method body, pushing the context C and the

environment E on the stack.

(EAppHit) is similar but far simpler. This rule applies when $A.m$ is in the cache. In this case we know its method body has been successfully type checked, so we need only check that the run-time type of v_2 is a subtype of the declared domain type of v_1. If so, we allow the method call to proceed.

However a method is called, the return, handled by (ERet), is the same. This rule applies when an expression has been fully evaluated and is at the top level. In this case, we pop the stack, replacing E' with E from the stack and plugging the value v into the context C from the stack.

Finally, (EContext) takes a step in an subexpression inside a context C. This rule only applies if the subexpression is not a method call (since that case is handled by (EApp*), which must push the context on the stack) and not a fully evaluated value (which is handled by (ERet), which must pop the context from the stack). We also do not allow the subexpression to itself be a context, since that could cause (EApp*) and (ERet) to misbehave.

Soundness. Our type system forbids invoking non-existent methods of objects. However, there are three kinds of errors the type system does not prevent: invoking a method on nil; calling a method whose body does not type check at run time; and calling a method that has a type signature but is itself undefined. (We could prevent the latter error by adding a side condition to (TApp) that requires the method to be defined, but we opt not to to keep the formalism slightly simpler.) To state a soundness theorem, we need to account for these cases, which we do by extending

the dynamic semantics with rules that reduce to *blame* in these three cases. After doing so, we can state soundness:

Theorem 1 (Soundness)**.** *If* $\emptyset \vdash \langle \emptyset, e \rangle \Rightarrow \langle \Gamma', \tau \rangle$ *then either* e *reduces to a value,* e *reduces to blame, or* e *diverges.*

We show soundness using a standard progress and preservation approach. The key technical challenge is preservation, in which we need to show that not only are expression types preserved, but also the validity of the cache and types of contexts pushed on the stack. The proof [34] can be found in Appendix A.

3.3 Implementation

Hummingbird is implemented using a combination of Ruby and OCaml. On the OCaml side, we use the Ruby Intermediate Language (RIL) [22] to parse input Ruby files and translate them to control-flow graphs (CFG) on which we perform type checking. On the Ruby side, we extend RDL [33], a contract system for Ruby, to perform static type checking. We next discuss the major challenges of implementing Hummingbird.

RIL. RIL is essentially the front-end of Diamondback Ruby (DRuby) [8,19]. Given an input Ruby program, RIL produces a CFG that simplifies away many of the tedious features of Ruby, e.g., multiple forms of conditionals. We modified DRuby so it emits the RIL CFG as a JSON file and then exits. When loading each application file at run-time, we read the corresponding JSON file and store a mapping from

class and method names and positions (file and line number) to the JSON CFG. At run-time we look up CFGs in this map to perform static type checking.

RDL and Type Checking. Like standard RDL, Hummingbird's type annotation stores type information in a map and wraps the associated method to intercept calls to it. We should emphasize that RDL does not perform any static checking on its own—rather, it solely enforces contracts dynamically. In Hummingbird, when a wrapped method is called, Hummingbird first checks to see if it has already been type checked. If not, Hummingbird retrieves the method's CFG and type and then statically checks that the CFG matches the given type.

Hummingbird uses RDL's type language, which includes nominal types, intersection types, union types, optional and variable length arguments, block (higher-order method) types, singleton types, structural types, a self type, generics, and types for heterogenous arrays and hashes. Hummingbird supports all of these kinds of types except structural types, self types, heterogeneous collections, and some variable length arguments. In addition, Hummingbird adds support for both instance field types (as seen in Figure 3.3) and class field types.

There is one slight subtlety in handling union types: If in a method call the receiver has a union type, Hummingbird performs type checking once for each arm of the union and the unions the possible return types. For example if in call $e.m(\ldots)$ the receiver has type $A \cup B$, then Hummingbird checks the call assuming e has type of $A.m$, yielding a return type τ_A; checks the call assuming $B.m$, yielding return type τ_B; and then sets the call's return type to $\tau_A \cup \tau_B$.

Eliminating Dynamic Checks. Recall the (EApp*) rules dynamically check that a method's actual arguments have the expected types before calling a statically typed method. This check ensures that an untrusted caller cannot violate the assumptions of a method. However, observe that if the immediate caller is itself statically checked, then we know the arguments are type-safe. Thus, as a performance optimization, Hummingbird only dynamically checks arguments of statically typed methods if the caller is itself not statically checked. As a further optimization, Hummingbird also does not dynamically check calls from Ruby standard library methods or the Rails framework, which are assumed to be type-safe. The one exception is that Hummingbird does dynamically check types for the Rails params hash, since those values come from the user's browser and hence are untrusted.

Numeric Hierarchy. Ruby has a Numeric tower that provides several related types for numbers. For example, Fixnum $<$ Integer $<$ Numeric and Bignum $<$ Integer $<$ Numeric. Adding two Fixnums normally results in another Fixnum, but adding two large Fixnums could result in a Bignum in the case of numeric overflow. To keep the type checking system simple, Hummingbird omits the special overflow case and does not take Bignum into consideration. (This could be addressed by enriching the type system [40].) Numeric overflow does not occur in our experiments.

Code Blocks. As mentioned earlier, Ruby code blocks are anonymous functions delimited by do...end. Hummingbird allows methods that take code block arguments to be annotated with the block's type. For example:

```
type :m, "() { (T) → U } → nil"
```

indicates that m takes no regular arguments; one code block argument where the block takes type T and returns type U; and m itself returns type nil.

There are two cases involving code blocks that we need to consider. First, suppose Hummingbird is statically type checking a call m() do |x| body end, and m has the type given just above. Then at the call, Hummingbird statically checks that the code block argument matches the expected type, i.e., assuming x has type T, then body must produce a value of type U. Second, when statically type checking m itself, Hummingbird should check that calls to the block are type correct. Currently this second case is unimplemented as it does not arise in our experiments.

Recall from above that Hummingbird sometimes needs to dynamically check the arguments to a statically typed method. While this test is easy to do for objects, it is hard to do for code blocks, which would require higher-order contracts [6]. Currently Hummingbird does not implement this higher order check, and simply assumes code block arguments are type safe. Also, Hummingbird currently assumes the self inside a code block is the same as in the enclosing method body. This assumption holds in our experiments, but it can be violated using instance_eval and instance_exec [41]. In the future, we plan to address this limitation by allowing the programmer to annotate the self type of code blocks.

Type Casts. While Hummingbird's type system is quite powerful, it cannot type check every Ruby program, and thus in some cases we need to insert type casts.

Hummingbird includes a method o.rdl_cast(t) that casts o's type to t. After such a call, Hummingbird assumes that o has type t. At run-time, the call dynamically checks that o has the given type.

In our experience, type casts have two main uses. First, sometimes program logic dictates that we can safely downcast an object. For example, consider the following method from one of our experiments:

```
def self.load_cache
  f = datafile_path (['cache', 'countries'])
  t = Marshal.load(File.binread(f))
  @@cache ||= t.rdl_cast ('Hash<String, %any>')
end
```

Marshal.load returns the result of converting its serialized data argument into a Ruby object of arbitray type. However, in our example, the argument passed to Marshal.load is always an application data file that will be converted to the annotated Hash.

Second, by default Hummingbird gives instances of generic classes their "raw" type with no type parameters. To add parameters, we use type casts, as in the following code:

```
a = []                          # a has type Array
a.rdl_cast ("Array<Fixnum>") # cast to Array<Fixnum>
a.push(0)                       # ok
a.push("str")                   # type error due to cast
```

Here without the type annotation the last line would succeed; with the annotation it triggers a type error. Note that when casting an array or hash to a generic type, rdl_cast iterates through the elements to ensure they have the given type.

Modules. Ruby supports mixins via *modules*, which are collections of methods that can be added to a class. Recall that Hummingbird caches which methods have been statically type checked. Because a module can be mixed in to multiple different classes—and can actually have different types in those different classes—we need to be careful that module method type checks are cached via where they are mixed in rather than via the module name.

For example, consider the following code, where the method foo defined in module M calls bar, which may vary depending on where M is mixed in:

```
1 module M def foo(x) bar(x) end end
2 class C; include M; def bar(x) x + 1 end end
3 class D; include M; def bar(x) x.to_s end end
```

Here method foo returns Fixnum when mixed into C and String when mixed into D. Thus, rather that track the type checking status of M#foo, Hummingbird separately tracks the statuses of C#foo and D#foo.

Cache Invalidation. Recall from Section 3.2 that Hummingbird needs to invalidate portions of the cache when certain typing assumptions change. While Hummingbird currently does not support cache invalidation in general, it does support one important case. In Rails *development mode*, Rails automatically reloads modified files without restarting, thus redefining the methods in those files but leaving other methods intact [31]. In Rails development mode, Hummingbird intercepts the Rails reloading process and performs appropriate cache invalidation. More specifically, when a method is called, if there is a difference between its new and old method body (which we check using the RIL CFGs), we invalidate the method and any

methods that depend on it. We also maintain a list of methods defined in each class, and when a class is reloaded we invalidate dependencies of any method that has been removed. In the next section, we report on an experiment running a Rails app under Hummingbird as it is updated.

We plan to add more general support for cache invalidation in future work. There are two main cases to consider. The first is when a method is redefined or is removed (which never happens in our experiments except in Rails development mode). Ruby provides two methods, method_added and method_removed, that can be used to register callbacks when the corresponding actions occur, which could be used for cache invalidation.

The second case of cache invalidation is method's type changes. However, in RDL and Hummingbird, multiple calls to type for the same method are used to create intersection types. For example, the core library Array#[] method is given its type with the following code:

```
1  type Array, :[], '(Fixnum or Float) → t'
2  type Array, :[], '(Fixnum, Fixnum) →Array<t>'
3  type Array, :[], '(Range<Fixnum>) →Array<t>'
```

meaning if given a Fixnum or Float, method Array#[] returns the array contents type; and, if given a pair of Fixnums or a Range<Fixnum>, it returns an array.

In this setting, we cannot easily distinguish adding a new arm of an intersection type from replacing a method type. Moreover, adding a new arm to an intersection type should not invalidate the cache, since the other arms are still in effect. Thus, full support of cache invalidation will likely require an explicit mechanism for replacing

App	LoC	Static types			Dynamic types		Casts	Phs	Running time (s)			
		Chk'd	App	All	Gen'd	Used			Orig	No$	Hum	Or. Ratio
Talks-1/4/2013	1,055	111	201	363	990	45	31	1	162	1,590	256	1.6×
Boxroom-1.7.1	854	127	221	306	534	93	17	1	263	705	327	1.2×
Pubs-1/12/2015	620	47	86	171	445	33	13	1	72.0	4,470	217	3.0×
Rolify-4.0.0	84	14	24	71	26	2	15	12	5.63	7.79	6.71	1.2×
CCT-3/23/2014	172	23	27	75	6	3	6	1	3.06	78.2	17.4	5.7×
Countries-1.1.0	227	33	40	111	0	0	22	1	1.02	18.1	4.62	4.5×

Figure 3.7: Type checking results.

earlier type definitions.

3.4 Experiments

We evaluated Hummingbird by applying it to six Ruby apps:

- *Talks*[2] is a Rails app, written by the second author, for publicizing talk announcements. Talks has been in use in the UMD CS department since February 2012.
- *Boxroom*[3] is a Rails implementation of a simple file sharing interface.
- *Pubs* is a Rails app, developed several years ago by the second author, for managing lists of publications.
- *Rolify*[4] is a role management library for Rails. For this evaluation, we integrated *Rolify* with *Talks* on the User resource.
- Credit Card Transactions (*CCT*)[5] is a library that performs simple credit card processing tasks.

[2] `https://github.com/jeffrey-s-foster/talks`
[3] `http://boxroomapp.com`
[4] `https://github.com/RolifyCommunity/rolify`
[5] `https://github.com/daino3/credit_card_transactions`

- *Countries*[6] is an app that provides useful data about each country.

We selected these apps for variety rather than for being representative. We chose these apps because their source code is publicly available (except *Pubs*); they work with the latest versions of Ruby and RDL; and they do not rely heavily on other packages. Moreover, the first three apps use Rails, which is an industrial strength web app framework that is widely deployed; the next two use various metaprogramming styles in different ways than Rails; and the last one does not use metaprogramming, as a baseline. Table 3.7 summarizes the results of applying Hummingbird to these apps. On the left we list the app name, version number or date (if no version number is available), and lines of code as measured with sloccount [46]. For the Rails apps, we ran sloccount on all ruby files in the model, controller, helper, and mailer directories. We do not include lines of code for views, as we do not type check views. For *Countries* and *CCT*, we ran sloccount on all files in the lib directory. For *Rolify*, we only statically checked several methods, comprising 84 lines of code, that use define_method in an interesting way.

Type Annotations. For all apps, we used common type annotations from RDL for the Ruby core and standard libraries. For several apps, we also added type annotations for third-party libraries and for the Rails framework. We trusted the annotations for all these libraries, i.e., we did not statically type check the library methods' bodies.

We also added code to dynamically generate types for metaprogramming code.

[6] `https://github.com/hexorx/countries`

For Rails, we added code to dynamically generate types for model getters and setters based on the database schema; for finder methods such as find_by_name and find_all_by_password (the method name indicates which field is being searched); and for Rails associations such as belongs_to.

In Figure 3.2, we showed code we added to *Rolify* to generate types for a method created by calling define_dynamic_method. Calling define_dynamic_method also dynamically creates another method, is_#{role_name}_of(arg)?, which we also provide types for in the pre block.

In *CCT*, we used the code in Figure 3.3 to generate types for Struct getters and setters.

Finally, we wrote type annotations for the app's own methods that were included in the lines of code count in Table 3.7. We marked those methods to indicate Hummingbird should statically type check their bodies. Developing these annotations was fairly straightforward, especially since we could quickly detect incorrect annotations by running Hummingbird.

Type Checking Results. For each program, we performed type checking while running unit tests that exercised all the type-annotated app methods. For *Talks* and *Pubs*, we wrote unit tests with the goal of covering all application methods. For *Boxroom*, we used its unit tests on models but wrote our own unit tests on controllers, since it did not have controller tests. For *Rolify*, we wrote a small set of unit tests for the dynamic method definition feature. For *CCT* and *Countries*, we used the unit tests that came with those apps.

In all cases, the app methods type check correctly in Hummingbird; there were no type errors. The middle group of columns summarizes more detailed type checking data.

The "Static types" columns report data on static type annotations. The count under "Chk'd" is the number of type annotations for the app's methods whose bodies we statically type checked. The count under "App" is that number plus the number of types for app-specific methods with (trusted) static type annotations, e.g., some Rails helper functions have types that we do not currently dynamically generate. The count under "All" reports the total number of static type annotations we used in type checking each app. This includes the "App" count plus standard, core, and third-party library type annotations for methods referred to in the app.

The "Dynamic types" columns report the number of types that were dynamically generated ("Gen'd") and the number of those that were actually used during type checking ("Used"). These numbers differ because we tried to make the dynamic type information general rather than app-specific, e.g., we generate both the getter and setter for belongs_to even if only one is used by the app.

These results show that having types for methods generated by metaprogramming is critical for successfully typing these programs—every app except *Countries* requires at least a few, and sometimes many, such types.

The "Casts" column reports the number of type casts we needed to make these programs type check; essentially this measures how often Hummingbird's type system is overly conservative. The results show we needed a non-trivial but relatively small number of casts. All casts were for the reasons discussed in Section 3.3:

downcasting and generics.

The "Phs" column in Table 3.7 shows the number of type checking phases under Hummingbird. Here a *phase* is defined as a sequence of type annotation calls with no intervening static type checks, followed by a sequence of static type checks with no intervening annotations. We can see that almost all apps have only a single phase, where the type annotations are executed before any static type checks. Investigating further, we found this is due to the way we added annotations. For example, we set up our Rails apps so the first loaded application file in turn loads all type annotation files. In practice the type annotations would likely be spread throughout the app's files, thus increasing the number of phases.

Rolify is the only application with multiple phases. Most of the phases come from calling define_dynamic_method, which dynamically defines other methods and adds their type annotations. The other phases come from the order in which the type annotation files are required—unlike the Rails apps, the *Rolify* type annotation files are loaded piecemeal as the application loads.

Performance. The last four columns of Table 3.7 report the overhead of using Hummingbird. The "Orig" column shows the running time without Hummingbird. The next two columns report the running time with Hummingbird, with caching disabled ("No$") and enabled ("Hum"). The last column lists the ratio of the "Hum" and "Orig" column.

For *Talks*, *Boxroom*, and *Pubs*, we measured the running time of a client script that uses curl to connect to the web server and exercise a wide range of

functionality. For *CCT*, we measured the time for running its unit tests 100 times. For *Countries* and *Rolify*, we measured the time for running the unit tests once (since these take take much more time than *CCT*'s tests). For all apps, we performed each measurement three times and took the arithmetic mean.

These results show that for the Rails apps, where IO is significant, Hummingbird slows down performance from 24% to 201% (with caching enabled). We think these are reasonable results for an early prototype that we have not spent much effort optimizing. Moreover, across all apps, the ratios are significantly better than prior systems that mix static and dynamic typing for Ruby [1, 36], which report orders of magnitude slowdowns.

Investigating further, we found that the main Hummingbird overhead arises from intercepting method calls to statically type checked methods. (Note the interception happens regardless of the cache state.) The higher slowdowns for *CCT* and *Countries* occur because those applications spend much of their time in code with intercepted calls, while the other applications spend most of their time in framework code, whose methods are not intercepted. We expect performance can be improved with further engineering effort.

We can also see from the results that caching is an important performance optimization: without caching, performance slows down 1.4× to 62×. We investigated *pubs*, the app with the highest no-caching slowdown, and found that while running the application with large array inputs, certain application methods are called more than 13,000 times while iterating through the large arrays. This means that each of these application methods are statically type checked more than 13,000 times when

caching is disabled.

Type Errors in *Talks*. We downloaded many earlier versions of *Talks* from its github repository and ran Hummingbird on them using mostly the same type annotations as for the latest version, changed as necessary due to program changes. Cumulatively, we found six type errors that were introduced and later removed as *Talks* evolved. Below the number after the date indicates which checkin it was, with 1 for the first checkin of the day, 2 for the second, etc.

- 1/8/12-4: This version misspells compute_edit_fields as copute_edit_fields. Hummingbird reported this error because the latter was an unbound local variable and was also not a valid method.

- 1/7/12-5: Instead of calling @list.talks.upcoming.sort{| a, b | ...}, this version calls @list.talks.upcoming{| a, b | ...} (leaving off the sort). Hummingbird detects this error because upcoming's type indicates it does not take a block. Interestingly, this error would not be detected at run-time by Ruby, which simply ignores unused block arguments.

- 1/26/12-3: This version calls user.subscribed_talks(true), but subscribed_talks's argument is a Symbol.

- 1/28/12: This version calls @job.handler.object, but @job.handler returns a String, which does not have an object method.

- 2/6/12-2: This version uses undefined variable old_talk. Thus, Hummingbird assumes old_talk is a no-argument method and attempts to look up its type,

which does not exist.

- 2/6/12-3: This version uses undefined variable new_talk

We should emphasize that although we expected there would be type errors in *Talks*, we did not know exactly what they were or what versions they were in. While the second author did write *Talks*, the errors were made a long time ago, and the second author rediscovered them independently by running Hummingbird.

Updates to Talks Finally, we performed an experiment in which we launched one version of *Talks* in Rails development mode and then updated the code to the next six consecutive versions of the app. (We skipped versions in which none of the Ruby application files changed) Notice that cache invalidation is particular useful here, since in typical usage only a small number of methods are changed by each update.

In more detail, after launching the initial version of the app, we repeated the following sequence six times: Reset the database (so that we run all versions with the same initial data); run a sequence of curl commands that access the same *Talks* functionalities as the ones used to measure the running time of *Talks* in Table 3.7 ; update the code to the next version; and repeat.

Table 3.1 shows the results of our experiment. The "Δ Meth" column lists the number of methods whose bodies or types were changed compared to the previous version. Note there are no removed methods in any of these versions. The "Added" column lists the number of methods added; such methods will be checked when they are called for the first time but do not cause any cache invalidations. The

"Deps" column counts the number of *dependent* methods that call one or more of the changed methods. These methods plus the changed methods are those whose previous static type check are invalidated by the update. The last column, "Chk'd," reports how many methods are newly or re-type checked after the update. Currently, Hummingbird always rechecks Rails helper methods, due to a quirk in the Rails implementation—the helper methods' classes get a new name each time the helper file is reloaded, causing Hummingbird to treat their methods as new. Thus (except for the first line, since this issue does not arise on the first run), we list two numbers in the column: the first with all rechecks, including the helper methods, and the second excluding the helper methods.

These results show that in almost all cases, the second number in "Chk'd" is equal to the sum of the three previous columns. There is one exception: in 8/24/12/-1, there 14 rechecked methods but 18 changed/added/dependent methods. We investigated and found that the 14 rechecks are composed of six changed methods that are rechecked once; two changed methods that are rechecked twice because they have dependencies whose updates are interleaved with calls to those methods; one added method that is checked; and three dependent methods that are rechecked. The remaining added method is not called by the curl scripts, and the remaining dependent methods are also changed methods (this is the only version where there is overlap between the changed and dependent columns).

Finally, as there are no type errors in this sequence of updates, we confirmed that this streak of updates type checks under Hummingbird.

Version	Δ Meth	Added	Deps	Chk'd
5/14/12	N/A	N/A	N/A	77
7/24/12	1	-	4	15 / 5
8/24/12-1	8	2	8	24 / 14
8/24/12-2	-	1	-	11 / 1
8/24/12-3	1	1	-	12 / 2
9/14/12	1	-	-	15 / 1
1/4/13	4	-	-	13 / 4

Table 3.1: Talks Update Results

3.5 Related Work

There are several threads of related work.

Type Systems for Ruby. We have developed several prior type systems for Ruby. Diamondback Ruby (DRuby) [8] is the first comprehensive type system for Ruby that we are aware of. Because Hummingbird checks types at run-time, we opted to implement our own type checker rather than reuse DRuby for type checking, which would have required some awkward shuffling of the type table between Ruby and OCaml. Another reason to reimplement type checking was to keep the type system a little easier to understand—DRuby performs type inference, which is quite complex for this type language, in contrast to Hummingbird, which implements much simpler type checking.

DRuby was effective but did not handle highly dynamic language constructs well. $\mathcal{P}$Ruby [7] solves this problem using profile-based type inference. To use $\mathcal{P}$Ruby, the developer runs the program once to record dynamic behavior, e.g., what methods are invoked via `send`, what strings are passed to `eval`, etc. $\mathcal{P}$Ruby then

applies DRuby to the original program text plus the profiled strings, e.g., any string that was passed to eval is parsed and analyzed like any other code. While $\mathcal{P}$Ruby can be effective, we think that Hummingbird's approach is ultimately more practical because Hummingbird does not require a separate, potentially cumbersome, profiling phase. We note that Hummingbird does not currently handle eval, because it was not used in our subject apps' code, but it could be supported in a straightforward way.

We also developed DRails [25], which type checks Rails apps by applying DRuby to translated Rails code. For example, if DRails sees a call to belongs_to, it outputs Ruby code that explicitly contains the methods generated from the call, which DRuby can then analyze. While DRails was applied to a range of programs, its analysis is quite brittle. Supporting each additional Rails feature in DRails requires implementing, in OCaml, a source-to-source transformation that mimics that feature. This is a huge effort and is hard to sustain as Rails evolves. In contrast, Hummingbird types are generated in Ruby, which is far easier. DRails is also complex to use: The program is combined into one file, then run to gather profile information, then transformed and type checked. Using Hummingbird is far simpler. Finally, DRails is Rails-specific, whereas Hummingbird applies readily to other Ruby frameworks. Due to all these issues, we feel Hummingbird is much more lightweight, agile, scalable, and maintainable than DRails. Finally, RubyDust [1] implements type inference for Ruby at run time. RubyDust works by wrapping objects to annotate them with type variables. More precisely, consider a method def m(x) ... end, and let α be the type variable for x. RubyDust's wrapping is

approximately equal to adding x = Wrap.new(x, α) to the beginning of m. Uses of the wrapped x generate type constraints on α and then delegate to the underlying object. The Ruby Type Checker [36] (rtc) is similar but implements type checking rather than type inference.

Hummingbird has several important advantages over RubyDust and rtc. First, RubyDust and rtc can only report errors on program paths they observe. In contrast, Hummingbird type checks all paths through methods it analyzes. Second, wrapping every object with a type annotation is extremely expensive. By doing static analysis, Hummingbird avoids this overhead. Finally, RubyDust and rtc have no special support for metaprogramming. In RubyDust, dynamically created methods could have their types inferred in a standard way, though RubyDust would likely not infer useful types for Rails-created methods. In rtc, dynamically created methods would lack types, so their use would not be checked. (Note that it would certainly be possible to add Hummingbird-style support for metaprogramming-generated type annotations to either RubyDust or rtc.) In sum, we think that Hummingbird strikes the right compromise between the purely static DRuby approach and the purely dynamic RubyDust/rtc approach.

Type Systems for Other Dynamic Languages. Many researchers have proposed type systems for dynamic languages, including Python [4], JavaScript [**?**, 13, 28], Racket [14,40,44], and Lua [29], or developed new dynamic languages or dialects with special type systems, such as Thorn [5], TypeScript [16,32], and Dart [18]. To our knowledge, these type systems are focused on checking the core language and

can have difficulty in the face of metaprogramming.

One exception is RPython [15], which introduces a notion of load time, during which highly dynamic features may be used, and run time, when they may not be. In contrast, Hummingbird does not need such a separation.

Lerner et al [27] propose a system for type checking programs that use JQuery, a very sophisticated Javascript framework. The proposed type system has special support for JQuery's abstractions, making it quite effective in that domain. On the other hand, it does not easily apply to other frameworks.

Feldthaus et al's TSCHECK [21] is a tool to check the correctness of TypeScript interfaces for JavaScript libraries. TSCHECK discovers a library's API by taking a snapshot after executing the library's top-level code. It then performs checking using a separate static analysis. This is similar to Hummingbird's tracking of type information at run-time and then performing static checking based on it. However, Hummingbird allows type information to be generated at any time and not just in top-level code.

Related Uses of Caching. Several researchers have proposed systems that use caching in a way related to Hummingbird. Koukoutos et al [26] reduce the overhead of checking data structure contracts (e.g., "this is a binary search tree") at run time by modifying nodes to hold key verification properties. This essentially caches those properties. However, because the properties are complex, the process of caching them is not automated.

Stulova et al [42] propose memoizing run-time assertion checking to improve

performance. This is similar to Hummingbird's type check caching, but much more sophisticated because the cached assertions arise from a rich logic.

Hermenegildo et al [24] proposed a method to incrementally update analysis results at run-time as code is added, deleted, or changed. Their analysis algorithms are designed for constraint logic programming languages, and are much more complicated than Hummingbird's type checking.

Staged Analysis. MetaOCaml [43] is a multi-stage extension of OCaml in which code is compiled in one stage and executed in a later stage. The MetaOCaml compiler performs static type checking on any such delayed code, which is similar to Hummingbird's just-in-time type checking. A key difference between MetaOCaml and Hummingbird is that Ruby programs do not have clearly delineated stages.

Chugh et al's staged program analysis [17] performs static analysis on as much code as is possible at compile time, and then computes a set of remaining checks to be performed at run time. Hummingbird uses a related idea in which no static analysis is performed at compile time, but type checking is always done when methods are called. Hummingbird is simpler because it need not compute which checks are necessary, as it always does the same kind of checking.

Other. Several researchers have explored other ways to bring the benefits of static typing to dynamic languages. Contracts [**?**] check assertions at function or method entry and exit. In contrast, Hummingbird performs static analysis of method bodies, which can find bugs on paths before they are run. At the same time, contracts can

encode richer properties than types.

Gradual typing [11] lets developers add types gradually as programs evolve; Vitousek et al recently implemented gradual typing for Python [45]. Like types [47] bring some of the flexibility of dynamic typing to statically typed languages. The goal of these systems is to allow mixing of typed and untyped code. This is orthogonal to Hummingbird, which focuses on checking code with type annotations.

Richards et al [37,38] have explored how highly dynamic language features are used in JavaScript. They find such features, including `eval`, are used extensively in a wide variety of ways, including supporting metaprogramming.

The GHC Haskell compiler lets developers defer type errors until run-time to suppress type errors on code that is never actually executed [23]. Hummingbird provides related behavior in that a method that is never called will never be type checked by Hummingbird. Template Haskell [39] can be used for compile-time metaprogramming. Since Haskell programs contain types, template Haskell is often used to generate type annotations, analogously to the type annotations generated using Hummingbird. Similarly, F# type providers [20] allow users to create compile time types, properties and methods. A key difference between these Haskell/F# features and Hummingbird is that Ruby does not have a separate compile time.

3.6 Conclusion

We presented Hummingbird, a novel tool that type checks Ruby apps using an approach we call just-in-time static type checking. Hummingbird works by tracking

type information *dynamically*, but then checking method bodies *statically* at run time as each method is called. As long as any metaprogramming code is extended to generate types as it creates methods, Hummingbird will, in a very natural way, be able to check code that uses the generated methods. Furthermore, Hummingbird can cache type checking so it need not be unnecessarily repeated at later calls to the same method.

We formalized Hummingbird using a core, Ruby-like language that allows methods and their types to be defined at arbitrary (and arbitrarily separate) points during execution, and we proved type soundness. We implemented Hummingbird on top of RIL, for parsing Ruby source code, and RDL, for intercepting method calls and storing type information. We applied Hummingbird to six Ruby apps, some of which use Rails. We found that Hummingbird's approach is effective, allowing it to successfully type check all the apps even in the presence of metaprogramming. We ran Hummingbird on earlier versions of one app and found several type errors. Furthermore, we ran Hummingbird while applying a sequence of updates to a Rails app in development mode to demonstrate cache invalidation under Hummingbird. Finally, we measured Hummingbird's run-time overhead and found it is reasonable.

In sum, we think that Hummingbird takes a strong step forward in bringing static typing to dynamic languages.

Chapter 4: Practical Type Inference

In the previous chapters, we introduced two type checking systems, rtc and Hummingbird. Although both systems are useful, they may require a substantial amount of manual annotations. Type inference automatically gathers subtyping constraints from code with no type annotations and finds the solution that satisfies all constraints. Thus, type inference reduces the programmer's burden of writing type annotations and improves the code's readability code while still performing type checking. Unfortunately, exisiting type inference approaches often generate types that are impractical to use. Some of the main issues include 1) structural types that have too many method requirements, and thus are too complicated to understand, and 2) Types such as $\top$ that are not meaningful in certain situations. In addition, prior Ruby type inference systems have limited support for intersection types and parameterized types. In this chapter, we will first introduce a standard type inference system for Ruby, and then describe a practical type inference sytem that overcomes the above limitations. The standard inference system tries to find the most general solution that satisfies all subtyping constraints. On the other hand, our practical inference system includes more constraint resolution rules in addition to unconventional solution extraction rules. Thus, the practical solution is a subtype

```
1   class C
2     def add(x)
3       x + 1
4     end
5     infer :add
6   end
7
8   s = Solver.new()
9   C.new.add(0)
10  s.solve()
```

Figure 4.1: Basic Type Inference Usage

of the standard solution that may be more specific and constrained. For example, suppose standard type inference infers the structural type solution x = [foo: ..., bar: ...]. Practical type inference may infer x = A where A is a class with foo and bar as instance methods.

The work described in this chapter has not been previously published.

4.1 Introduction

Like Hummingbird, our type inference system also uses method wrapping and interception to perform program analysis, and relies on a set of existing type annotations. The programmer specifies the method for which to infer types by calling infer with the method name. The type inference system then wraps these methods for run-time interception. We assume that type annotations already exist on all methods that are not infer-wrapped. The programmer manually calls solve to invoke type inference. We support the inference of method argument/return types, field types, and Rails-specific session and params hashes, which we will discuss later.

Figure 4.1 illustrates the basic use of the type inference system. Line 8 creates

the Solver object. Then, line 9 invokes add defined on line 2. Since infer is invoked with add on line 5, the system statically analyzes the body of add for constraint generation before invoking add. Finally, the solve call on line 10 infers the type of add.

We now briefly describe the three steps of type inference. We will detail the process in later sections.

1) **Constraint generation.** When wrapped methods are intercepted at run-time, the type inference system statically walks through the method bodies to generate subtyping constraints of the form $\tau_1 \leq \tau_2$. At the same time, the system performs type checking whenever possible. If τ_1 and τ_2 are not type variables and $\tau_1 \not\leq \tau_2$, then the system reports a type error. In this paper, we refer to τ_1 as the precedessor of τ_2, and τ_2 as the successor of τ_1. This first step is the same for both standard and practical inference.

2) **Constraint resolution.** Apply a set of constraint resolution rules on the original constraints to possibly introduce new constraints. The simplest example is conventional transitive closure, where if $\tau_1 \leq v, v \leq \tau_2$, then $\tau_1 \leq \tau_2$. In later sections, we will describe many other cases of constraint manipulation. Most of the standard constraint resolution rules are a subset of the practical resolution rules.

3) **Solution extraction.** Merge all relevant constraints on each type variable. The relevant constraints are chosen differently between standard and practical inference. In standard type inference, we merge types based on the position of

each type variable. Method arguments are in negative position, and method returns are in positive position. For a negative variable, the final merged type is the intersection of all of its successors. For a positive variable, the merged type is the union of all of its predecessors. As a standard type inference example, suppose that the final constraints involving α are:

$\mathsf{A} \leq \alpha$, $\mathsf{B} \leq \alpha$, $\alpha \leq [\mathsf{foo}\colon\ \tau_1 \to \tau_2\ ]$, $\alpha \leq [\mathsf{bar}\colon\ \tau_3 \to \tau_4\ ]$, where the types in the last two constraints are structural types specifying methods that must be defined on α. If α is a positive variable, or return type, then we union its predecessors, so we infer $\alpha = \mathsf{A} \cup \mathsf{B}$. If α is a negative variable, then we take the intersection of its successors, which is $[\mathsf{foo}\colon\ \tau_1 \to \tau_2, \mathsf{bar}\colon\ \tau_3 \to \tau_4\ ]$. This new structural type indicates that α must have both foo and bar defined on it. The system will check $A \cup B \leq [\mathsf{foo}\colon\ \tau_1 \to \tau_2, \mathsf{bar}\colon\ \tau_3 \to \tau_4\ ]$. We relax these rules in practical inference, which we will discuss later.

4.2 Motivating Examples

We begin with a simple example of standard type inference, followed by a few examples of practical type inference to show where it generates better types.

4.2.1 Standard Type Inference Example

Figure 4.2 is a simplified version of a method from pubs, an app used in our evaluation. This method first sets tmp to an empty string. Then, it calls various instance methods of the argument, possibly modifies tmp, and finally returns tmp.

```
1   type String, :≪, '([to_s: () → String]) → String'
2
3   def flags(paper)
4     tmp = ''                    # String ≤ tmp
5     if paper.invited then       # paper ≤[invited: (v1) → v2]
6       tmp = 'Invited'           # String ≤ tmp
7     end
8     if paper.draft then         # paper ≤[draft: (v3) → v4]
9       tmp ≪'Draft'              # String ≤ tmp
10    end
11    if paper.hidden then        # paper ≤[hidden: (v5) → v6]
12      tmp ≪'Hidden'             # String ≤ tmp
13    end
14    return tmp                  # tmp ≤ret
15  end
```

Figure 4.2: Simple Ruby Method

The constraint generated by each line is shown in the corresponding comment. This first constraint is trivial. Then, line 5 generates a structural constraint paper $\leq$ [invited: (v1) $\rightarrow$ v2], which indicates that paper is a subtype of an object that has an invited method with argument v1 and return v2. Then, the structural constraint on line 5 indicates that paper has invited defined as an instance method. Note that since we do not know the actual type of paper, we cannot look up the annotated type of invited. Thus, we simply assume invited takes one variable argument v1, and returns variable v2. The count of one argument comes from the actual call. Next, the conditional paper.invited generates a trivial duplicate constraint on tmp from the assignment. On line 8, we know that paper has draft as an instance method. In the paper.draft true branch, we again generate the same subtyping relationship between String and tmp. Note that in the true branch of paper.invited, tmp is used in an assignment, so we do not perform any type annotation lookups. However, in the true branch of paper.draft, tmp is the method's caller, and tmp is stored

as String in the current type environment. So we look at the annotated type of String#<<, and type check the argument type. In the third conditional, we generate similar constraints. Finally, we know that tmp must be a subtype of the method's return. In this particular example, constraint resolution yields no new constraints, because there are no new constraints that we could introduce based on the original constraints to help with inference.

During solution extraction, we infer paper, a negative variable (or argument type), as [invited: (v1) → v2, draft: (v3) → v4, hidden: (v5) → v6]. which is the intersection of all of its successors. On the other hand, tmp is a positive variable, so we take the union of all of its predecessors, which is simply String. Note the vi's do not have any constraints on them. So their types are simply their initial types, where v1, v3, v5 are ⊤, and v2, v4, v6 are ⊥. We would like to point out that the vi's are simply intermediate local variables, and we do not consider their inferred types in our evalution in Section 4.5.

We now use this example to describe the reason for taking the intersection of successors for each negative variable. Suppose that we instead take the union of the successors, then the type of paper becomes [invited: (v1) → v2] ∪ [draft: (v3) → v4] ∪ [hidden: (v5) → v6]. Since the actual argument only needs to be subtype of this structural type, [invited: (v1) → v2] is a valid actual argument. Next, in transitive closure, we generate constraints:

[invited: (v1) → v2] ≤ [invited: (v1) → v2],

[invited: (v1) → v2] ≤ [draft: (v3) → v4],

[invited: (v1) → v2] ≤ [hidden: (v5) → v6].

But notice the last two constraints are invalid.

In comparison, a negative variable must account for all types that flow into it. Suppose that we modified flags so that its last statement becomes if ... return tmp else return 0. If we instead take the intersection of the predecessors, then the return type would be Fixnum ∩ String, which is an invalid type. Thus, the only valid precise type is the union of the predecessors, or Fixnum ∪ String.

However, we will demonstrate later in this chapter that sometimes it helps to consider variables from the reversed direction.

4.2.2 Practical Type Inference Examples

Since standard type inference computes the most general solution, the resulting types may be too long or too complicated to understand. On the other hand, practical type inference aims at inferring types that are concise and resemble how a programmer would write types.

4.2.2.1 Structural Type to Actual Class Conversion

We now revisit Figure 4.2 to demonstrate structural type to actual class conversion, a key feature of practical type inference. Recall that standard type inference produces [invited: (v1) → v2, draft: (v3) → v4, hidden: (v5) → v6] as the type of paper. Although this type is correct, we could use practical type inference to reduce this type to a concise type, or an actual class. In particuar, if we search through all run-time classes, we find that Paper is the only class that has all methods in the

```
1  type :Transaction, :account_name, 'String'
2
3  class Card
4    def process_transactions
5      ...
6      name = t.account_name    # String ≤ name
7      card = find_card(name)   # name ≤ n
8      ...
9    end
10
11   def find_card(n)
12     card = ...
13     if card.account_name == n
14       ...
15     end
16     ...
17   end
```

Figure 4.3: Reversed Solution Extraction

structural type as instance methods. Thus, we add a new constraint paper = Paper. This also allows us to eventually infer Paper as the final type of paper In this case, Paper is a much more concise solution than [invited: (v1) → v2, draft: (v3) → v4, hidden: (v5) → v6]. In the future, we plan to conduct a user study on the types generated from practical inference.

In the evaluation section, we will show this conversion feature very helpful in practice, as it reduces many much longer structural types to single classes.

4.2.2.2 Reversed Solution Extraction

Consider Figure 4.3, a simplified code snippet from the CCT app. We use the example to demonstrate how practical solution infers a much better type for find_card's argument n through reversed solution extraction.

In process_transactions, we know from line 1 that t.account_name returns String.

Also, in find_card, suppose n only appears on line 13. During constraint resolution, we take the trivial constraints shown in comments, and generate one transitive constraint String $\leq$ n.

Now consider the standard type inference of n. Since it is a negative variable, we infer its type by merging its successors. First, assume that on line 13, we do not know the actual class of card to look up the type annotation of account_name, so we generate no constraint from this call. Thus, the result of n is $\top$, an impractical type. Next, consider the case where we know the type of card. We are then able to look up the return type of its instance method account_name as String. In all conventional classes including String, the annotated type of == is theoretically $(\top) \to$ Bool. Although we get a new constraint n $\leq \top$ from this annotation, it doesn't change the inferred type of n. However, in practice, the argument type of comparison methods like == is usually the same as the caller's type. We will discuss how this phenomenon helps us to infer more better types with practical inference in Section 4.2.2.3.

Ideally, we want to infer String as the type of n. Notice that when we reach the solution extraction phase, the only constraint involving n is the transitive constraint String $\leq$ n. In standard solution extraction, we essentially ignore this helpful constraint because standard inference comptues the most general solution. On the other hand, in practical solution extraction, we read solutions from the reversed side if the conventional direction does not offer useful type information. We find this approach helpful as the reverse side often also provides the right type information. This feature allows us to easily infer String as the type of n.

```
1  def empty(a)
2    return a == ""                    # a ≤[ ==: (String) → Bool]
3  end
4
5  def format_author(s)
6    return "[no authors]" if empty s  # s ≤a
7    authors = s. split (' and ')      # s ≤[ split : (String) → ...]
8    ...
9  end
```

Figure 4.4: Method name-based inference

4.2.2.3 Method Name-based Inference

Another way to infer practical types is to generate constraints based on method names. As we mentioned earlier, in a comparison method such as ==, the type of the argument is theoretically $\top$. However, the type of a variable with instance methods such as == or != is very likely the same as the method argument's type in practice.

Consider Figure 4.4, a simplified code snippet taken from the pubs app. The original constraints are again shown in the comments. During constraint resolution, we add a transitive constraint $\mathsf{s} \leq [\ ==: (\mathsf{String}) \to \mathsf{Bool}]$. We can easily see that standard approach infers $\mathsf{a} = [\ ==: (\mathsf{String}) \to \mathsf{Bool}]$, and $\mathsf{s} = [\ ==: (\mathsf{String}) \to \mathsf{Bool}, \mathsf{split}: (\mathsf{String}) \to ...]$.

Again, although these types are correct, practical inference generates much better types. Using the above technique on inferring the caller of == based on the type of argument, we infer both String as the type of both a and s. Furthermore, we would like to point out that the technique in Section 4.2.2.1 to convert structural types to actual classes is not feasible in this == case. During run-time, we find too

values	v	::=	$\mathsf{nil} \mid [A] \mid \tau_g$
names	n	::=	$A.\mathsf{m} \mid @f$
expressions	e	::=	$v \mid x \mid \mathsf{self} \mid x = e \mid e; e \mid A.\mathsf{new}$
		$\mid$	$\mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$
		$\mid$	$\mathsf{case}\ \alpha\ (\mathsf{when}\ A\ \mathsf{do}\ e)^+\ ?(\mathsf{else\ do}\ e)$
		$\mid$	$e.m(e) \mid @f$
		$\mid$	$\mathsf{type}\ A.m : \tau_m \mid \mathsf{infer}\ n$
mth defs	d	::=	$\mathsf{def}\ \mathsf{m}(x : \alpha^-) : \alpha^+ = e$
classes	c	::=	$\mathsf{class}\ A < B = d^*$
programs	p	::=	$c; c$
simpl val typs	τ_{si}	::=	$A \mid \mathsf{nil} \mid \tau \cup \tau \mid \tau_g$
val typs	τ	::=	$\tau_{si} \mid \alpha \mid \tau_s \mid \tau_i \mid \tau_\iota \mid \tau_\rho$
generic typs	τ_g	::=	$\mathrm{Array}\langle\tau\rangle \mid \mathrm{Hash}\langle\tau, \tau\rangle$
mth typs	τ_m	::=	$\tau \to \tau$
inter typs	τ_i	::=	$\tau_{m1} \cap \tau_{m2} \cap ... \cap \tau_{mn}$
struct typs	τ_s	::=	$[\mathsf{m}_1 : \tau_m, \mathsf{m}_2 : \tau_m, ..., \mathsf{m}_n : \tau_m]$
imp typs	τ_ι	::=	$< (\alpha \le \tau_{si1}) \to \tau_{si2}, (\alpha \le \tau_{si3}) \to \tau_{si4}, ... >^I$
possible typs	τ_ρ	::=	$< \tau_{si1}, \tau_{si2}, ... >^p$
constraints/delayed constraints	C, C^D	::=	$\tau \le \tau \mid C \cup C$

$x \in$ var ids $\quad m \in$ mth ids $\quad A \in$ cls ids

Figure 4.5: Source Language

many actual clases that match this structural type. In fact, there are 1620 classes that have == as instance methods, and 23 classes that have both == and split as instance methods.

4.3 Formalism

We formalize the source language using the core, Ruby-like langugage show in Figure 4.5. Values v include nil and $[A]$, like Hummingbird. In addition, values includes τ_g for generic types. Next, names n include method names and field names. Expressions e include every expression from Hummingbird, except we redefine method defintions d separately and add case statements. The formalism in

Hummingbird allows a method to be defined inside an expression, which then allows a method to be defined anywhere during execution. In this chapter, we focus on type inference only and do not consider dynamic method creations. Next, A case statement on a type variable has a sequence of condtionals guarded by classes, followed by an optional else branch. We also add the infer expression, which simply tells the type inferencer to infer the type of the associated method or field.

We write method definitions d as a method with argument and returns that are both type variables that we will attempt to solve, and the body is an expression. Throughout this chapter, we will use $^-$ to indicate that the associated variable is an argument, and $^+$ to indicate the variable is a return. In addition, we introduce classes c in this language, where each class is a sequence of 0 or more method definitions, and we allow each class to have an ancestor. Then, each program p is simply a sequence is class definitions.

Next, we define simple value types τ_{si} to include class instances A, nil, union types, and generic types. The definition of value types τ becomes much more complex than in Hummingbird. In addition to including τ_{si}, τ includes α for type variables, τ_s for structural types, τ_i for intersection types, and two new types τ_ι and τ_ρ for possible and implication types

We define generic types τ_g as either an Array with one parameter, or a Hash with two parameters. For simplity, we only include these two generic types. The next three types are very straightforward. Method types τ_m have the exact same definition as in Hummingbird. An intersection type τ_i is a list of method types. A structural type τ_s is a list of method names along with their types.

```
1 type :Tag, 'self.find', '(String or Fixnum) → Tag'
2 type :Tag, 'self.find', '(Array<String or Fixnum>) →Array<Tag>'
3
4 def tagged
5   a = ...
6   @tag = Tag.find(a)
7   @tag.papers          # @tag ≤[ papers:  ...  ]
8    ...
9 end
```

Figure 4.6: Possible and Implication Type Example

We now introduce the two novel types in our system, possible types τ_ρ and implication types τ_ι. Both are created using information from intersection type annotations to help us choose the right arm of the intersection type.

An implication type is simply a list of implications of the form $< (\alpha \leq \tau_{si1}) \rightarrow \tau_{si2}, (\alpha \leq \tau_{si3}) \rightarrow \tau_{si4}, ... >^I$, where $\alpha \leq \tau_{si1}, \alpha \leq \tau_{si3}, ...$ are hypotheses, and $\tau_{si2}, \tau_{si4}, ...$ are conclusions. Each hypothesis is a subtyping relationship between an argument variable α^- and a simple value type. The conclusion of each implication is also a simple value type. During constraint resolution, we apply a set of rules to each implication type in an attempt to resolve it to a single valid implication. The conclusion of the single valid implication is the resolved implication type. A possible type is a much simpler type that is used in conjunction with an intersection type. It is defined as simply a list of simple types, only one of which is valid.

We now give a simple example to demonstrate how implication and possible types can prevent a false positive type error during type inference. Consider the method in Figure 4.6, taken from the pubs app, and assume that we do not have any type information on a. First, suppose our type system does not support implication

and possible types. This means that on line 6, $\mathsf{find\_ret} \leq \mathsf{@tag}$, where $\mathsf{find\_ret}$ is the return type of $\mathsf{Tag.find(a)}$. We show find's annotated type as an intersection type on lines 1 and 2, which means that $\mathsf{find\_ret} \leq \mathsf{Tag} \cup \mathsf{Array}\langle\mathsf{Tag}\rangle$, or the union of the intersection's two return types. We combine this constraint and the constraint shown on line 7 to create a transitive constraint $\mathsf{Tag} \cup \mathsf{Array}\langle\mathsf{Tag}\rangle \leq [\mathsf{papers}: \ldots]$. Since the predecessor is a union type, every type in the union must be a subtype of the successor, which that both $\mathsf{Tag} \leq [\mathsf{papers}: \ldots]$ and $\mathsf{Array}\langle\mathsf{Tag}\rangle \leq [\mathsf{papers}: \ldots]$ must hold. But notice the only the first constraint holds, and the later is a type error because Array does not have papers as an instance method.

We rectify this issue with implication and possible types. In this case, we get two constraints from line 6: $\mathsf{a} \leq \langle\ \mathsf{Fixnum} \cup \mathsf{String}, \mathsf{Array}\langle\mathsf{Fixnum} \cup \mathsf{String}\rangle\rangle^p$ and $\langle\ \mathsf{a} \leq \mathsf{Fixnum} \cup \mathsf{String} \rightarrow \mathsf{Tag}, \mathsf{a} \leq \mathsf{Array}\langle\mathsf{Fixnum} \cup \mathsf{String}\rangle \rightarrow \mathsf{Array}\langle\mathsf{Tag}\rangle\ \rangle^I \leq \mathsf{@tag}$. Next, we combine this constraint with the structural constraint shown on line 7, and conclude that the only valid implication is $\mathsf{a} \leq \mathsf{Fixnum} \cup \mathsf{String} \rightarrow \mathsf{Tag}$ because Tag is the only conclusion with papers as an instance method. Thus, instead of getting a type error, we infer $\mathsf{@tag} = \mathsf{Tag}$.

Finally, each constraint C in the source language is simply a list of subtyping relationships. During constraint resolution, we may encounter a point where we do not have enough information on a method argument variable to choose a valid arm from an intersection type. Thus, it may help to delay the consideration of this constraint, as we may later gather more information on this variable through various constraint resolution rules. We indicate this delay of constraints as C^D, and we will show details about this delay operation in the constraint resolution section.

$$\frac{}{\Gamma \vdash \mathsf{nil} : \mathsf{nil}}\ (\text{TNil}) \qquad \frac{}{\Gamma \vdash [A] : A}\ (\text{TObject}) \qquad \frac{}{\Gamma \vdash \mathsf{self} : \Gamma(\mathsf{self})}\ (\text{TSelf}) \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\ (\text{TVar})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1; e_2 : \tau_2}\ (\text{TSeq}) \qquad \frac{\Gamma \vdash e : \tau \qquad \tau \leq \Gamma(x)}{\Gamma \vdash x = e : \tau}\ (\text{TAssn}) \qquad \frac{}{\Gamma \vdash A.\mathsf{new} : A}\ (\text{TNew})$$

$$\frac{\Gamma \vdash e_0 : \tau_0 \qquad \Gamma \vdash e_1 : \tau_1 \qquad \tau_0 \leq [\mathsf{m} : \tau_1 \to \beta] \qquad \beta \text{ fresh}}{\Gamma \vdash e_0.\mathsf{m}(e_1) : \beta}\ (\text{TApp})$$

$$\frac{\Gamma \vdash e_0 : \tau \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau_1 \cup \tau_2}\ (\text{TIf})$$

$$\frac{\Gamma \vdash \alpha : A \cup B \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{case}\ \alpha\ \mathsf{when}\ A\ \mathsf{do}\ e_1\ \mathsf{when}\ B\ \mathsf{do}\ e_2 : \tau_1 \cup \tau_2}\ (\text{TCase})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \Gamma \vdash e_3 : \tau_3}{\Gamma \vdash \mathsf{case}\ \alpha\ \mathsf{when}\ A\ \mathsf{do}\ e_1\ \mathsf{when}\ B\ \mathsf{do}\ e_2\ \mathsf{else\ do}\ e_3 : \tau_1 \cup \tau_2 \cup \tau_3}\ (\text{TCaseElse})$$

$$\frac{\mathsf{class}\ A < B\ d* \in p}{p \vdash A < B}\ (\text{TClass})$$

Figure 4.7: Constraint Generation

4.3.1 Constraint Generation

Our constraint generation process is mostly straightforward. During constraint generation, we statically through the AST and assign a type for each expression based on the the formalism in Figure 4.7. This formalism is a simplification of the implementation.

In the this figure, Γ is the type environment that maps expressions to their corresponding types. (TNil) and (TObject) simply give nil and instances the obvious

types. (TSelf) and (TVar) give self and local variables their types according to the current environment. (TSeq) states that a sequence of two expressions has the type of the second expression. (TAssn) is the flow-insensitive assignment rule that requires the successor to be a subtype of the predecessor. (TNew) types object creation in the obvious way. (TApp) is method invocation rule. We look up the annotated type of $e_0.\mathsf{m}$, and 1) ensure the actual method argument type is a subtype of the annotated argument type, and 2) the return has the annotated return type. (TIf) says that the return type of a conditional is the union of the branch types. (TCase) is a special rule about case statements without the else branch. The rule is similar to TIf where the result is the union of the branch types. In addition, we assume that the type variable α has the types of the guards. (TCaseElse) is similar to (TIf), where we simply union the type of each branch. Since we cannot easily infer α's type in the else branch, we do not make any assumptions about α's type in the whole statement. Finally, (TClass) defines sub-class relationships in the obvious way.

4.3.2 Standard Constraint Resolution

(top)		
$\{\alpha \leq \top\} \cup C$	$\Rightarrow$	C
(bot)		
$\{\bot \leq \alpha\} \cup C$	$\Rightarrow$	C
(trans)		
$\{\tau_1 \leq \tau, \tau \leq \tau_2\} \cup C$	$\Rightarrow$	$\{\tau_1 \leq \tau, \tau \leq \tau_2, \tau_1 \leq \tau_2\} \cup C$ if $\tau \in \{\alpha, \tau_\iota, \tau_\rho\}$
(n-nil)		
$\{\alpha \leq A, \alpha \leq B\} \cup C$	$\Rightarrow$	$\{\alpha \leq \mathsf{nil}, \mathsf{nil} \leq \alpha\} \cup C$ if $A \not\leq B \wedge B \not\leq A$

Table 4.1 Standard Constraint Resolution Rules (continued on next page)

(continued from previous page)

(g-g) $\{A\langle\tau_1\rangle \leq A\langle\tau_2\rangle\} \cup C$	$\Rightarrow$	$\{\tau_1 \leq \tau_2, \tau_2 \leq \tau_1\} \cup C$
(trans-struct-0-err) $\{[\mathsf{m} : (\tau_a) \to \tau_r] \leq [\mathsf{n} : (\tau'_a) \to \tau'_r]\} \cup C$	$\Rightarrow$	Err
(trans-struct-1-err) $\{[\mathsf{m} : (\tau_a) \to \tau_r] \leq [\mathsf{m} : (\tau'_a) \to \tau'_r]\} \cup C$	$\Rightarrow$	Err if $(\mathsf{cmp}(\tau'_a, \tau_a) = \mathsf{Err} \vee \mathsf{cmp}(\tau_r, \tau_{r'}) = \mathsf{Err}$
(trans-struct-1) $\{[\mathsf{m} : (\tau_a) \to \tau_r] \leq [\mathsf{m} : (\tau'_a) \to \tau'_r]\} \cup C$	$\Rightarrow$	$\{\mathsf{cmp}(\tau'_a, \tau_a), \mathsf{cmp}(\tau_r, \tau'_r)\} \cup C$
(trans-struct-2-err) $\{\tau_s \leq A\} \cup C$	$\Rightarrow$	Err if $\exists \mathsf{m}.\mathsf{m} \in A \wedge \mathsf{m} \notin \tau_s$
(trans-struct-2) $\{[\mathsf{m} : (\tau_a) \to \tau_r] \leq A\} \cup C$	$\Rightarrow$	$\{[\mathsf{m} : (\tau_a) \to \tau_r] \leq [\mathsf{m} : (\tau'_a) \to \tau'_r]\} \cup C$ if $TT[A.m_i] = [(\tau'_a) \to \tau'_r]$
(trans-struct-err-3) $\{A \leq \tau_s\} \cup C$	$\Rightarrow$	Err if $\exists \mathsf{m}.\mathsf{m} \in \tau_s \wedge \mathsf{m} \notin A$
(trans-struct-3) $\{A \leq [\mathsf{m} : (\tau_a) \to \tau_r]\} \cup C$	$\Rightarrow$	$\{[\mathsf{m} : (\tau'_a) \to \tau'_r] \leq [\mathsf{m} : (\tau_a) \to \tau_r]\} \cup C$ if $TT'[A.m, (\tau_a) \to \tau_r] = (\tau'_a) \to \tau'_r$
(trans-struct-3-delay) $\{A \leq [\mathsf{m} : (\tau_a) \to \tau_r]\} \cup C$	$\Rightarrow$	$\{A \leq [\mathsf{m} : (\tau_a) \to \tau_r]\}^D \cup C$ if $TT'[A.m, (\tau_a) \to \tau_r] = (\tau'_{ia}) \to \tau'_{ir} \cap ...$
(union-t) $\{\tau_1 \cup \tau_2 \leq \tau_3\} \cup C$	$\Rightarrow$	$\{\tau_1 \leq \tau_3, \tau_2 \leq \tau_3\} \cup C$
(possible) $\{\tau \leq \langle\tau_1, ..., \tau_n\rangle^P\} \cup C$	$\Rightarrow$	$\{\tau = \mathrm{match}(\tau, \langle\tau_1, ..., \tau_n\rangle^P\} \cup C$ if $\tau \notin \alpha_i$
(implication-ret-delay) $\{\tau_\iota \leq \tau_1, ...\tau_\iota \leq \tau_n\} \cup C$	$\Rightarrow$	$\{\tau_\iota \leq \tau_1, ...\tau_\iota \leq \tau_n\}^D \cup C$ if $\lvert\mathrm{imatch}(\tau_\iota, [\tau_1, ..., \tau_n]).\mathsf{conclusions}\rvert > 1$ $\wedge \forall k \in [1..n].\tau_k \notin \alpha_i \wedge \nexists \tau_j > n.(\tau_j \notin \alpha_i \wedge \tau_\iota \leq \tau_j)$
(implication-ret) $\{\tau_\iota \leq \tau_1, ...\tau_\iota \leq \tau_n\} \cup C$	$\Rightarrow$	$\{\tau_\iota = \tau_r, \alpha \leq \tau_{a1}, ..., \alpha \leq \tau_{an}, \tau_\iota \leq \tau_1, ...\tau_\iota \leq \tau_n\} \cup C$ if $\mathrm{imatch}(\tau_\iota, [\tau_1, ..., \tau_n]) =$ $[\alpha \leq \tau_{a1} \to \tau_r, ..., \alpha \leq \tau_{an} \to \tau_r]$ $\wedge \forall k \in [1..n].\tau_k \notin \alpha_i \wedge \nexists \tau_j > n.(\tau_j \notin \alpha_i \wedge \tau_\iota \leq \tau_j)$
(implication-arg-delay) $\{\tau_\iota(\leq \vee \geq)\tau\} \cup C$	$\Rightarrow$	$\{\tau_\iota(\leq \vee \geq)\tau\}^D \cup C$ if $\tau_\iota = \langle\alpha \leq \tau_{a1} \to \tau_{r1}, ..., \alpha \leq \tau_{an} \to \tau_{rn}\rangle^I$ $\wedge \mathsf{sol}(\alpha) = \bot$
(implication-arg) $\{\tau_\iota(\leq \vee \geq)\tau\} \cup C$	$\Rightarrow$	$\{\tau_\iota = \tau_{rj}, \tau_\iota(\leq \vee \geq)\tau\} \cup C$ if $\tau_\iota = \langle\alpha \leq \tau_{a1} \to \tau_{r1}, ..., \alpha \leq \tau_{an} \to \tau_{rn}\rangle^I$ $\wedge \mathsf{sol}(\alpha) = \tau' \wedge \tau' \leq \tau_{aj}$

Table 4.1: Standard Constraint Resolution Rules

Table 4.1 shows the standard constraint resolution rules. During constraint

resolution, we apply these rules exhaustivey in order until there are no new constraints.

top $\{\alpha \leq \top\} \cup C \Rightarrow C$

Eliminate all constraints with $\top$ on the right side.

bot $\{\bot \leq \alpha\} \cup C \Rightarrow C$

Eliminate all constraints with $\bot$ on the left side.

trans $\{\tau_1 \leq \tau, \tau \leq \tau_2\} \cup C \Rightarrow \{\tau_1 \leq \tau, \tau \leq \tau_2, \tau_1 \leq \tau_2\} \cup C$ if $\tau \in \{\alpha, \tau_\iota, \tau_\rho\}$

If τ is a type variable, implication type, or possible type, then transitively connect τ's predecessors with successors, and keep the original constraint. The reason we add the new constraint if $\tau \in \{\alpha, \tau_\iota, \tau_\rho\}$ is that α, τ_ι, and τ_ρ are all unknown types, even though τ_ι, and τ_ρ carry some type information themselves.

n-nil $\{\alpha \leq A, \alpha \leq B\} \cup C \Rightarrow \{\alpha \leq \mathsf{nil}, \mathsf{nil} \leq \alpha\} \cup C$ if $A \not\leq B \wedge B \not\leq A$

If α has successors that are different classes, then $\alpha = \mathsf{nil}$ is the only constraint that satisfies α. Hence, we add $v = \mathsf{nil}$, and delete constraints connecting α to these successors. However, we have not encountered a case in our apps where it is necessary to apply this rule.

g-g $\{A\langle\tau_1\rangle \leq A\langle\tau_2\rangle\} \cup C \Rightarrow \{\tau_1 \leq \tau_2, \tau_2 \leq \tau_1\} \cup C$

If there is a subtyping relationship between two generic types with the same base, then assume their parameters are equal.

trans-struct-err-0 $\{[\mathsf{m} : (\tau_a) \to \tau_r] \leq [\mathsf{n} : (\tau'_a) \to \tau'_r]\} \cup C \Rightarrow \mathsf{Err}$

If two structural types have a subtyping relationship, but do not share the same method name, then we get a type error. Note that in our inference system, each structural type has only one method prior to solution extraction. This reason for this is that there are two ways to get structural types during constraint generation. 1) method calls where we do not know the type of the caller. Consider the method call be x.foo(y). Suppose we do not know the type of x. Then we generate $\mathsf{x} \leq [\mathsf{foo}\colon (\mathsf{v1}) \rightarrow \mathsf{v2})\ ]$. It is obvious here that it is only possible to have one method in the structural type. 2) structural type annotations. This is where the structural type comes from a prior anntotation. There is only one method in every structural type annotation that we used in our experiments.

trans-struct-err-1 $\{[\mathsf{m} : (\tau_a) \rightarrow \tau_r] \leq [\mathsf{m} : (\tau'_a) \rightarrow \tau'_r]\} \cup C \Rightarrow \mathsf{Err}$

if $(\mathsf{cmp}(\tau'_a, \tau_a) = \mathsf{Err} \vee \mathsf{cmp}(\tau_r, \tau_{r'}) = \mathsf{Err}$

Error if both sides of the constraint are structural types that share the same method name, but their argument or return types are not compatible. We define cmp to check for compatibility between two types.

$$\mathrm{cmp}(\tau_1, \tau_2) = \begin{cases} \tau_1 \leq \tau_2 & \text{if } \tau_2 \in \alpha_i \vee \tau_1 \in \alpha_i \\ \tau_1 \leq \tau_2, & \text{if } \tau_1 \leq \tau_2 \\ \mathsf{Err}, & \text{otherwise} \end{cases} \tag{4.1}$$

In other words, if $\tau_1 \leq \tau_2$ and one of the types is a variable, then assume this is

a valid constraint. Otherwise, perform normal type checking on this subtyping relationship and either return the same relationship if valid, or return error.

trans-struct-1 $\{[\mathsf{m} : (\tau_a) \to \tau_r] \leq [\mathsf{m} : (\tau'_a) \to \tau'_r]\} \cup C \Rightarrow$

$\{\mathsf{cmp}(\tau'_a, \tau_a), \mathsf{cmp}(\tau_r, \tau'_r)\} \cup C$

If both sides are structural types with the same method name, then add contravariant constraints between the argument types and covariant constraints between the return types. Also delete the original constraint.

trans-struct-err-2 $\{\tau_s \leq A\} \cup C \Rightarrow \mathsf{Err}$ if $\exists \mathsf{m}.\mathsf{m} \in A \wedge \mathsf{m} \notin \tau_s$

Error if a structural type is a subtype of a class that has more methods defined. Note that we use $\mathsf{m} \notin \tau_s$ to indicate that m is not included the methods of τ_s.

trans-struct-2 $\{[\mathsf{m} : (\tau_a) \to \tau_r] \leq A\} \cup C \Rightarrow \{[\mathsf{m} : (\tau_a) \to \tau_r] \leq [\mathsf{m} : (\tau'_a) \to \tau'_r]\} \cup C$ if $TT[A.m_i] = [(\tau'_a) \to \tau'_r]$

First, recall from Chapter 3 that TT is a type table mapping class and method ids $\mathsf{A.m}$ to their corresponding types. If a structural type with method m is a subtype of a class, then replace the class with its annotated type on m. We assume the annotated type is not an intersection type.

trans-struct-err-3 $\{A \leq \tau_s\} \cup C \Rightarrow \mathsf{Err}$ if $\exists \mathsf{m}.\mathsf{m} \in \tau_s \wedge \mathsf{m} \notin A$

Error if a class is a subtype of a structural type with method m, but m is undefined on the class. Note that we use $\mathsf{m} \notin A$ to indicate that m is not an instance method of A.

trans-struct-3 $\{A \leq [\mathsf{m} : (\tau_a) \to \tau_r]\} \cup C \Rightarrow \{[\mathsf{m} : (\tau'_a) \to \tau'_r] \leq [\mathsf{m} : (\tau_a) \to \tau_r]\} \cup C$

if $TT'[A.m, (\tau_a) \to \tau_r] = (\tau'_a) \to \tau'_r$

If A is a subtype of a structural type $[\mathsf{m} : \tau_a \to \tau_r]$, then look up the annotated type of A.m with relevance to this structural type in the special type table TT'. In comparison to TT, a type lookup in TT' requires an extra structural type parameter. The reason is that a type lookup in TT may return an intersection type, and this structural type parameter may help us to reduce the intersection type to a single method type. If this lookup in TT' returns a a single method type, then replace A with this type. We define type lookup in TT' formally below. The first and last cases do not involve intersection types, thus the lookup results resemble that of in TT. In the second case where the lookup in TT returns an intersection type, we apply the selection operator sel' in an attempt to reduce the intersection to a single method type.

$$C \vdash TT'[A.\mathsf{m}, \tau_a \to \tau_r] = \begin{cases} TT[A.\mathsf{m}] & \text{if } TT[A.\mathsf{m}] = \tau_m \\ \mathsf{sel}'(TT[A.\mathsf{m}], \tau_a \to \tau_r), & \text{if } TT[A.\mathsf{m}] = \tau_{m1} \cap \tau_{m2} \cap ... \\ \text{Err} & \text{otherwise} \end{cases} \tag{4.2}$$

We now define the select operator sel'

$$\mathsf{sel}'(\tau_i, \tau_a \to \tau_r) = \begin{cases} \mathsf{sel}(\tau_i, \tau_a \to \tau_r) & \text{if } \tau_a \notin \alpha_i \\ \mathsf{sel}(\tau_i, \mathsf{sol}(\tau_a) \to \tau_r) & \text{otherwise} \end{cases} \tag{4.3}$$

In this definition, sel represents the conventional intersection type selection based on the method argument. If τ_a is not a type variable, then apply the conventional sel on the types. Otherwise, we solve for τ_a by invoking the solution extraction step on this variable, and then apply sel with this solution. Note that at this stage, τ_a's solution may not be its final solution. Nonetheless, we may be able apply sel with this solution to get a single method type.

trans-struct-3-delay $\{A \leq [\mathsf{m} : (\tau_a) \to \tau_r]\} \cup C \Rightarrow \{A \leq [\mathsf{m} : (\tau_a) \to \tau_r]\}^D \cup C$

if $TT'[A.m, (\tau_a) \to \tau_r] = (\tau'_{ia}) \to \tau'_{ir} \cap ...$

In the above rule, a type lookup in TT' may still be an intersection type. This happens when we are unble to select a valid arm of the intersection because the current constraints do not provide enough information on τ_a. In this case, we delay the constraint in hopes of getting new constraints on τ_a later. It is important to note that in some cases such as incomplete programs, there may never be enough constraints on τ_a to choose the right arm from the intersection type. After the number of delays on a variable exceeds the threshold of 100, we assume that there will be no more new constraints on the variable, and generalize intersection type by converting it to a regular method type. We use 100 as the threshold because it is sufficient in our test apps. To generalize an intersection type into a regular method type, we simply union all arguments and all returns in the original type, e.g., $[(\alpha_1) \to \alpha_2, (\alpha_3) \to \alpha_4]$ becomes $(\alpha_1 \cup \alpha_3) \to \alpha_2 \cup \alpha_4$.

union-t $\{\tau_1 \cup \tau_2 \leq \tau_3\} \cup C \Rightarrow \{\tau_1 \leq \tau_3, \tau_2 \leq \tau_3\} \cup C$

If a union type is a subtype of another type τ_3, then add edges between each type of the union type and τ_3.

possible $\{\tau \leq \langle \tau_1, ..., \tau_n \rangle^P\} \cup C \Rightarrow \{\tau = \text{match}(\tau, \langle \tau_1, ..., \tau_n \rangle^P\} \cup C$ if $\tau \notin \alpha_i$

If a non-variable type τ is a subtype of a possible type τ_ρ, then apply the match operation to find a matching type for τ in τ_ρ. Replace this constraint with an equality between τ and this match.

We formally define $\mathsf{match}(\tau, \tau_\rho)$ below to support the cases where τ is an Array, a nominal, or an implication type.

$$\text{match}(\text{Array}\langle\tau\rangle, \tau_\rho) = \begin{cases} \text{Array}\langle\tau'\rangle, & \text{if Array}\langle\tau'\rangle \in \tau_\rho \wedge \tau \sim \tau' \\ & \wedge \nexists \text{Array}\langle\tau''\rangle \in \tau_\rho . \tau' \neq \tau'' \\ \mathsf{Err}, & \text{otherwise} \end{cases} \tag{4.4}$$

In the above equation, we use the compatibility operator $\sim (\tau_1, \tau_2)$ to simply indicate false if $\text{cmp}(\tau_1, \tau_2) = \mathsf{Err}$, and true otherwise.

If τ is an Array, then the matching type in the possible type is $\text{Array}\langle\tau'\rangle$, where τ' is compatible with τ's parameter. Note that this compatibility means that the Array parameters do not have to match if one of them is a variable. In addition, we assume that there are no other Arrays in the possible type.

$$\text{match}(A, \tau_\rho) = \begin{cases} A, & \text{if } A \in \tau_\rho \\ B, & \text{if } A < B \wedge B \in \tau_\rho \\ \mathsf{Err}, & \text{otherwise} \end{cases} \tag{4.5}$$

If τ is a class, then the matching class is either τ or its superclass.

$$\text{match}(\tau_\iota, \tau_\rho) = \begin{cases} \text{match}(\tau', \tau_\rho), & \text{if } \tau_\iota = [(\alpha \leq \tau_{si1}) \to \tau_{si2}, (\alpha \leq \tau_{si3}) \to \tau_{si4}, ...] \wedge \\ & \quad \exists \tau' \in \{\tau_{si2}, \tau_{si4}, ...\}.\text{match}(\tau', \tau_\rho) \neq \mathsf{Err} \\ \mathsf{Err}, & \text{otherwise} \end{cases} \tag{4.6}$$

If τ is an implication type, then we recursively apply the match operator on each conclusion in the implication and τ_ρ to find a match that succeeds. This match is then the matching type for the entire implication type.

implication-ret-delay / implication-ret

$\{\tau_\iota \leq \tau_1, ...\tau_\iota \leq \tau_n\} \cup C \Rightarrow \{\tau_\iota \leq \tau_1, ...\tau_\iota \leq \tau_n\}^D \cup C$

if $|\text{imatch}(\tau_\iota, [\tau_1, ..., \tau_n]).\mathsf{conclusions}| > 1$

$\wedge \forall k \in [1..n].\tau_k \notin \alpha_i \wedge \nexists \tau_j > n.(\tau_j \notin \alpha_i \wedge \tau_\iota \leq \tau_j)$

$\{\tau_\iota \leq \tau_1, ...\tau_\iota \leq \tau_n\} \cup C \Rightarrow \{\tau_\iota = \tau_r, \alpha \leq \tau_{a1}, ..., \alpha \leq \tau_{an}, \tau_\iota \leq \tau_1, ...\tau_\iota \leq \tau_n\} \cup C$

if $\text{imatch}(\tau_\iota, [\tau_1, ..., \tau_n]) = [\alpha \leq \tau_{a1} \to \tau_r, ..., \alpha \leq \tau_{an} \to \tau_r]$

$\wedge \forall k \in [1..n].\tau_k \notin \alpha_i \wedge \nexists \tau_j > n.(\tau_j \notin \alpha_i \wedge \tau_\iota \leq \tau_j)$

If we encounter an implication type τ_ι, then apply imatch on τ_ι and all of its non-variable successors to choose valid implications of τ_ι based on these successors. Recall that in an implication type of the form $< (\alpha \leq \tau_{si1}) \rightarrow \tau_{si2}, (\alpha \leq \tau_{si3}) \rightarrow \tau_{si4}, ... >^I$, the hypotheses are $\alpha \leq \tau_{si1}, \alpha \leq \tau_{si3}, ...$, and the conclusions are $\tau_{si2}, \tau_{si4}, ...$ Since our goal is to resolve the implication type to a single simple type, we ensure that the valid implications share the same conclusion. At the end, we generate a constraint that the implication type is equivalent to the conclusion of the chosen valid implications. In addition, the hypothesis of each valid implication becomes an actual constraint.

We now describe the process of imatch. First, we divide the successors into two groups, structure successors and non-structure successors. Then we combine valid implications from each group and check for inconsistencies.

We define imatch_struct$(\tau_\iota, struct_lst)$ to select a list of implications from τ_ι whose returns match the list of structural types. This means that the conclusions of the implications have the structural type methods as instance methods. Formally, we write

imatch_struct$(\langle imp_1, imp_2, ... \rangle^I, [m_1 : ..., m_2 : ..., m_n : ...]) =$

$\{imp \mid imp \in \{imp_1, imp_2, ...\} \wedge \{m_1, m_2, ..., m_n\} \in \tau_r\}$

where

$imp = (\alpha \leq \tau_a) \rightarrow \tau_r \wedge$

$imp_1 = (\alpha \leq \tau_{si1}) \rightarrow \tau_{si2}, imp_2 = (\alpha \leq \tau_{si3}) \rightarrow \tau_{si4}, ...$

One important point to note is that there are some Ruby methods that are defined on all conventional classes, such as dup. Such methods are unhelpful in the above implication selection because all classes would match. Thus, we elimate structural types with these methods before applying imatch_struct. Next, we define imatch_other to select a list of implications from τ_ι that match a list of non-structural types. Formally, we write

imatch_other$(\langle imp_1, imp_2, ... \rangle^I, [\tau_1, \tau_2, ..., \tau_n]) =$

$\{imp \mid imp \in \{imp_1, imp_2, ...\} \wedge \tau_r \in [\tau_1, \tau_2, ..., \tau_n]\}$ where

$imp = (\alpha \leq \tau_a) \rightarrow \tau_r \wedge$

$imp_1 = (\alpha \leq \tau_{si1}) \rightarrow \tau_{si2}, imp_2 = (\alpha \leq \tau_{si3}) \rightarrow \tau_{si4}, ...$

In other words, we select implications whose returns are elements of the non-structural type list. (In the above equation, we use $\tau_r \in [\tau_1, \tau_2, ..., \tau_n]$ to indicate that τ_r is either equivalent to an element in the non-structural type list, or τ_r is a subtype of a union type in the list.)

Finally, we define imatch to get the combined result of imatch_struct and imatch_other. We use ms to represent the result of match_struct$(\tau_\iota, slst)$, where $slst$ is the list of structural types from τ_ι's successors, and we use mo to represent the result of match_other$(\tau_\iota, olst)$, where olst is the list of non-structural types from τ_ι's successors.

$$\text{imatch}(\tau_\iota, lst) = \begin{cases} \mathsf{Err} & ms = mo = \emptyset \\ \mathsf{Err} & ms \neq \emptyset \wedge mo \neq \emptyset \wedge ms \cap mo = \emptyset \\ \mathsf{Delay} & \text{if } |(ms \cap mo).\mathsf{conclusions}| > 1 \\ ms \cap mo & \text{if } |(ms \cap mo).\mathsf{conclusions}| = 1 \end{cases} \tag{4.7}$$

Furthermore, we define $imp_1 \cap imp_2$, where imp_1 and imp_2 are sets of implications.

$$imp_1 \cap imp_2 = \begin{cases} imp_2 & imp_1 = \emptyset \\ imp_1 & imp_2 = \emptyset \\ imp_1 \wedge imp_2 & \text{otherwise} \end{cases} \tag{4.8}$$

In the two cases of imatch, we report an error if both imatch_struct and imatch_other are unable to find valid implications, or both find implications but their intersection is empty. In the third case, we find implications with multiple conclusions. If ms and mo are both non-empty, then the valid implications is the intersection of the two. Since our goal is to select implications that share the same conclusion, we delay this this implication type as we may later have constraints on the type. In the last case, we have exactly one conclusion in the chosen implications. This is the case where generate constraints based on the valid implications.

Similar to the trans-struct-3 rule, there may never be any new constraint on the delayed type. If the number of delays on the type exceeds the threshold, then

we set the implication type to a union type whose elements are the conclusions of the implication type.

implication-arg / implication-arg-delay

$$\{\tau_\iota(\leq \vee \geq)\tau\} \cup C \Rightarrow \{\tau_\iota(\leq \vee \geq)\tau\}^D \cup C$$

if $\tau_\iota = \langle \alpha \leq \tau_{a1} \to \tau_{r1}, ..., \alpha \leq \tau_{aj} \to \tau_{rj}, ..., \alpha \leq \tau_{an} \to \tau_{rn} \rangle^I \wedge \mathsf{sol}(\alpha) = \bot$

$$\{\tau_\iota(\leq \vee \geq)\tau\} \cup C \Rightarrow \{\tau_\iota = \tau_{rj}, \tau_\iota(\leq \vee \geq)\tau\} \cup C$$

if $\tau_\iota = \langle \alpha \leq \tau_{a1} \to \tau_{r1}, ..., \alpha \leq \tau_{aj} \to \tau_{rj}, ..., \alpha \leq \tau_{an} \to \tau_{rn} \rangle^I \wedge \mathsf{sol}(\alpha) = \tau' \wedge \tau' \leq \tau_{aj}$

This is also a rule where we select a list of valid implications from the implication type τ_ι. However, this selection is only based on the type of the argument α in the implication type's hypotheses. We invoke the solution extraction phase on α to get its current solution. If we are unable to obtain a solution because of the lack of constraints, then we delay resolving the implication type. Otherwise, we simply select a valid solution from τ_ι where the subtyping hypothesis holds, and set it equivalent to the implication type.

4.3.3 Standard Solution Extraction

During solution extraction, we read off the solution to each variable according to the standard solution extraction algorithm in Figure 4.8. We first define the predecessors of a variable to be all of its subtypes that are not type variables,

```
1   preds(α) = ∀τ.τ ≤ α ∧ τ ∉ {α_i, τ_ι, τ_ρ}
2   succs(α) = ∀τ.α ≤ τ ∧ τ ∉ {α_i, τ_ι, τ_ρ}
3
4   def read_sol(α)
5     if α+
6       sol = pred_sol(α)
7     elsif α−
8       sol = succ_sol(α)
9     elsif α+−
10      if preds(α) = ∅
11        sol = succ_sol(α)
12      elsif succs(α) = ∅
13        sol = pred_sol(α)
14      else
15        sol = merge_to_intersection(succ_sol(α), pred_sol(α))
16      end
17    end
18    return sol
19  end
20
21  def pred_sol(α)
22    sol = ⊥
23    preds(α).each {|p| merge_to_union(sol, p) }
24    return sol
25  end
26
27  def succ_sol(α)
28    sol = ⊤
29    succs(α).each {|s| merge_to_intersection(sol, s) }
30    return sol
31  end
```

Figure 4.8: Standard Solution Extraction

```
1   def merge_to_union(a, b)
2     return a if a == b
3     match(a, b) with
4       (⊥, _) ⇒ b
5       (⊤, _) ⇒ a
6       (α_i, _) ⇒ b
7       (b ∩ _, _) ⇒ b
8       (_, _) ⇒ a ∪ b
9     end
10  end
```

Figure 4.9: Merge to Union

implication types and possible types. Then, successors are defined similarly, but for supertypes. In read_sol, we obtain the final solution of each type variable. if α is positive, or a return type, then we merge $\bot$ with all of its precessors to a union. On the other hand, if α is negative, or an argument type, then we merge $\top$ with all of its successors to intersection.

However, if a α is both positive and negative, or a field variable, then the process becomes a little more complicated. If α has no predecessors, we merge its successors to intersection. If α has no successors, then we merge its predecessors to union. Finally, if α has predecessors and successors, then we merge its successor solution and predecessor solution into an intersection.

The details of merging two types into a union are shown in Figure 4.9, and the rules are self-explanatory.

On the other hand, merging types to an intersection is more complicated. We first show merge_to_intersection in Figure 4.10. Many of the rules are self-explanatory, and we will just point out the key ideas. On line 9, merging two distinct classes into intersection results in nil. Finally, line 11 shows a case where a class does not have

```
1   def merge_to_intersection (a, b)
2     match(a, b) with
3       (⊥, _) ⇒ a
4       (⊤, _) ⇒ b
5       (∈ α_i, _) ⇒ b
6       ([m1 : τ_a1 → τ_r1], [m2 : τ_a2 → τ_r2]) ⇒ [m1 : τ_a1 → τ_r1, m2 : τ_a2 → τ_r2]
7       (b ∪_ ⇒, _) ⇒ b
8       (c ∪ d, _) ⇒ merge_to_intersection(c, b) ∪ merge_to_intersection(d, b)
9       (A, B) ⇒ nil
10      (A, [m : τ_a1 → τ_r1]) ⇒ a if m ∈ A
11      (A, [m : τ_a1 → τ_r1]) ⇒ Err if m ∉ A
12      (_, _) ⇒ a ∩ b
13  end
```

Figure 4.10: Merge to Intersection

the structural type method defined on it. This is the only case where an error may occur.

Note that the solution extraction algorithm does not perform variable substitutions. Thus, a solution may contain variables, i.e. $\mathsf{v} = [\mathsf{id}\colon () \to \alpha]$. To make the solutions easier to read, we apply an additional solution simplication step the results. During solution simplification, we recursively take the resolved types that contain no variables and substitute these variables with their solutions in the dependent types.

4.3.4 Practical Constraint Resolution

We define the following constraint resolution rules to aim at inferring practical types that are concise and resemble manual annotations. Practical constraint resolution includes all standard resolution rules, except n-nil is replaced with n-err.

We show the rest of practical constraint resolution rules in Figure 4.11

(n-err)
$\{\alpha \leq A, \alpha \leq B\} \cup C \Rightarrow \mathsf{Err}$ if $A \not\leq B \cap B \not\leq A$

(g-g-v-)
$\{\alpha^- \leq A\langle\tau_1\rangle, \alpha^- \leq A\langle\tau_2\rangle\} \cup C \Rightarrow \{\alpha^- \leq A\langle\tau_1\rangle, \tau_1 \leq \tau_2, \tau_2 \leq \tau_1\} \cup C$

(g-g-v+)
$\{A\langle\tau_1\rangle \leq \alpha^+, A\langle\tau_2\rangle \leq \alpha^+, \tau_1 = \tau_2\} \cup C \Rightarrow \{A\langle\tau_1\rangle \leq \alpha^+, \tau_1 \leq \tau_2, \tau_2 \leq \tau_1\} \cup C$

(n-s-v-)
$\{\alpha^- \leq \tau_s, \alpha^- \leq A\} \cup C \Rightarrow \{\alpha^- \leq A, A \leq \alpha^-, A \leq \tau_s\} \cup C$

(param-read)
$\{\mathsf{params}_{k1} \leq [] : (k_2) \to v_2\} \cup C \Rightarrow \{\mathsf{params}_{k1\_k2} = v_2\} \cup C$

(param-write)
$\{\mathsf{params}_{k1} \leq [] =: (k_2, v_2) \to v_3\} \cup C \Rightarrow \{\mathsf{params}_{k1\_k2} = v_3, v_2 \leq v_3\} \cup C$

(struct-to-nominal)
$\{\alpha^- \leq \tau_{s1}, ..., \alpha^- \leq \tau_{sn}\} \cup C \Rightarrow \{\alpha^- = \mathsf{mc}'(\mathsf{mc}([\tau_{s1}, ..., \tau_{sn}], \mathrm{preds}(\alpha))\} \cup C$
if $|\mathsf{mc}([\tau_{s1}, ..., \tau_{sn})], \mathrm{preds}(\alpha)| = 1 \vee 2$
$\wedge \forall(\{\alpha^- \leq \tau\} \in C, \tau \notin \{\tau_{s1}, ..., \tau_{sn}\}).\tau \in \{\alpha_i, \tau_\iota, \tau_\rho\}$

(struct-cmp)
$\{\alpha^- \leq [\mathrm{m} : \tau_a \to \tau_r]\} \cup C \Rightarrow \{\alpha^- \leq \tau_a, \alpha^- \leq [\mathrm{m} : \tau_a \to \tau_r]\} \cup C$
if $\mathrm{m} \in \{==, \neq\} \wedge \tau_a \neq \mathsf{nil} \wedge \tau_a \notin \alpha_i \wedge \tau_r \in \alpha_i$
$\wedge \nexists\{\alpha^- \leq [m' : \tau'_a \to \tau'_r]\} \in C.$
$(m' \in \{==, \neq\} \wedge [\mathrm{m}' : \tau'_a \to \tau'_r] \neq [\mathrm{m} : \tau_a \to \tau_r])$

(struct-include)
$\{\mathrm{Array}\langle\tau\rangle \leq [\mathrm{include?} : \alpha \to \tau_r]\} \cup C \Rightarrow \{\alpha \leq \tau, \mathrm{Array}\langle\tau\rangle \leq [\mathrm{include?} : \alpha \to \tau_r]\} \cup C$

Figure 4.11: Practical Constraint Resolution Rules

n-err $\{\alpha \leq A, \alpha \leq B\} \cup C \Rightarrow \mathsf{Err}$ if $A \not\leq B \cap B \not\leq A$

Error if a variable has two distinct classes as successors, unless A is a subtype of B, or B is a subtype of A.

g-g-v- $\{\alpha^- \leq A\langle\tau_1\rangle, \alpha^- \leq A\langle\tau_2\rangle\} \cup C \Rightarrow \{\alpha^- \leq A\langle\tau_1\rangle, \tau_1 \leq \tau_2, \tau_2 \leq \tau_1\} \cup C$

If a negative variable is a subtype of two generic types that have the same base, then assume the parameters are equal. Also keep one of the original constraints.

g-g-v+ $\{A\langle\tau_1\rangle \leq \alpha^+, A\langle\tau_2\rangle \leq \alpha^+, \tau_1 \leq \tau_2, \tau_2 \leq \tau_1\} \cup C \Rightarrow$

$\{A\langle\tau_1\rangle \leq \alpha^+, \tau_1 \leq \tau_2, \tau_2 \leq \tau_1\} \cup C$

If two generic types with the same base are subtypes the same positive variable and the parameters are equivalent, then remove the first constraint between

the generic type and the type variable.

n-s-v- $\{\alpha^- \leq \tau_s, \alpha^- \leq A\} \cup C \Rightarrow \{\alpha^- \leq A, A \leq \alpha^-, A \leq \tau_s\} \cup C$

If a negative variable has a structural successor and a class successor, then assume 1) the negative variable is equivalent to the class, and 2) the class is a subtype of the structural type.

param-read $\{\mathsf{params}_{k1} \leq [] : (k_2) \rightarrow v_2\} \cup C \Rightarrow \{\mathsf{params}_{k1\_k2} = v_2\} \cup C$

We rename each Rails params hash access of the form $\mathsf{params}[key]$ to a special type variable $\mathsf{params\_}$: key in order to avoid using finite hash types. In this resolution rule, we generate constraints to support the inference of double param hash types through hash reads.

Consider params[:k1], a generic type that is represented as type variable params_:k1 in our system. During the constraint generation phase, a hash read on params[:k1][:k2] generates the constraint

params_:k1 $\leq$ [[]: (k2) $\rightarrow$ v2], where v2 represents the return type of the read. In this case, we convert params[:k1][:k2] to a new special variable params_:k1_:k2 and set it equivalent to v2. This constraint simply indicates that the variable representing the result of the hash read is equivalent to the actual result of the hash read.

Finally, we remove the original constraint connecting the params variable to the structura type.

param-write $\{\mathsf{params}_{k1} \leq [] =: (k_2, v_2) \rightarrow v_3\} \cup C \Rightarrow \{\mathsf{params}_{k1\_k2} = v_3, v_2 \leq$

$v_3\} \cup C$

Similiar to above rule, this rule generates constraints to support the inference of double params hash types. However, in this rule we generate constraints based on hash writes.

Consider params[:k1][:k2] = v2, where we generate the constraint

params_:k1 $\leq$ [[]=: (k2, v2) $\rightarrow$ v3)], where v3 is the return of the method call. In this case, we set the special type variable params_:k1_:k2 to v3. Moreover, we assume v2 is a subtype of v3. Again, we also remove the original constraint connecting the param type variable to the structural type.

struct-to-nominal $\{\alpha^- \leq \tau_{s1}, ..., \alpha^- \leq \tau_{sn}\} \quad \cup C \Rightarrow$

$\{\alpha^- = \mathsf{mc}'(\mathsf{mc}([\tau_{s1}, ..., \tau_{sn}], \text{preds}(\alpha))\} \cup C$

if $|\mathsf{mc}([\tau_{s1}, ..., \tau_{sn})], \text{preds}(\alpha)| = 1 \vee 2$

$\wedge \forall(\{\alpha^- \leq \tau\} \in C, \tau \notin \{\tau_{s1}, ..., \tau_{sn}\}).\tau \in \{\alpha_i, \tau_\iota, \tau_\rho\}$

If the successors of a negative variable include only structural types, type variables, and possible types, then convert the structural types into actual classes if applicable.

We obtain the list of actual matching classes with two steps. First, we obtain the list of actual classes by applying mc_0 on all structural type successors to gather all actual classes that have the structural type methods defined, or formally

$\mathsf{mc}_0([\text{m}_1 : \_, \text{m}_2 : \_, ..., \text{m}_n]) = cls\_lst$

where $\forall A \in cls\_lst.\{\text{m}_1, \text{m}_2, ..., \text{m}_n\} \in A$.

Furthermore, we ignore the actual method types because it is sufficient to search for matching classes based on the method names alone in our apps.

In the second step, we apply mc on the above result with type variable α in an attempt to reduce the number of matching classes.

$$\mathrm{mc}'(cls\_lst, \alpha) = \begin{cases} [Model], & \text{if } Model \in cls\_lst \\ [\mathrm{Array}\langle \tau_a \rangle], & \mathrm{Array} \in cls\_lst \wedge \\ & \mathrm{structs}(\mathrm{succs}(\alpha)) = [\mathrm{each}{:}()\{\tau_a \to \tau_r\} \to \_] \\ [\mathrm{params}\langle k', v' \rangle], & \forall [m : \_] \in \mathrm{structs}(\mathrm{succs}(\alpha)).m \in \mathrm{meths}(\mathrm{Hash}) \\ & \mathrm{params}\langle k, v \rangle \in \mathrm{preds}(\alpha) \\ cls\_lst, & \text{otherwise} \end{cases} \tag{4.9}$$

In the first three cases, we reduce the number of matching classes.

1. This case is Rails specific, where $Model$ represents an actual Rails model class. As long the original matching class list contains $Model$, we eliminate all other classes.

2. If 1) the original matching classes include Array and 2) the only structural successor is the specified structural type, then keep Array as the only matching class. The reason behind this is that although there are a large set of classes with each defined, normally each is associated with Array, especially if there is only one block parameter. Furthermore, we set the

block parameter τ_a in the structural type as the Array paramter, based on the annotated type of Array#each.

3. This is another Rails specific case. If all structural successor methods of α are defined on Hash and params$\langle k, v\rangle$ is a predecessor of α, then include params$\langle k', v'\rangle$ as the only matching class.

We find these rules very useful in practice, as they can often elminate a large number of classes that are irrelevant to the ideal inferred types.

Finally, we define the conversion operation mc to convert the matching classes to proper types, if applicable.

$$\text{mc}(cls\_lst) = \begin{cases} A, & \text{if } cls\_lst = \{A\} \\ A \cup B, & \text{if } cls\_lst = \{A, B\} \\ \text{undefined}, & \text{if } |cls\_lst| > 2 \end{cases} \tag{4.10}$$

In other words, we only convert a negative variable's structural successors if the final number of actual matching classes is 1 or 2. We find that usually there are either 1 or 2 matching classes, or a very large number of matching classes that are impractical to convert. One example of too many matching classes is [to_s: () → String]. Almost every Ruby class meets this structural type requirement, thus it is only practical to keep the type as is.

struct-cmp $\{\alpha^- \leq [\text{m} : \tau_a \to \tau_r]\} \cup C \Rightarrow \{\alpha^- \leq \tau_a, \alpha^- \leq [\text{m} : \tau_a \to \tau_r]\} \cup C$

if $\text{m} \in \{==, \neq\} \wedge \tau_a \neq \mathsf{nil} \wedge \tau_a \notin \alpha_i \wedge \tau_r \in \alpha_i$

$\wedge \nexists \{\alpha^- \leq [m' : \tau'_a \to \tau'_r]\} \in C.$

$(m' \in \{==, \neq\} \wedge [\mathrm{m}' : \tau'_a \to \tau'_r] \neq [\mathrm{m} : \tau_a \to \tau_r])$

This is a special rule based on the name of the method call. For conventional classes, the argument of a comparison method could have any type. However, if type variable α^- is a subtype of a structural type with a comparison method (== or !=), then usually α has the same type as the method's argument. We add a constraint from α^- to the method argument if the following conditions are met: 1) the argument is not a type variable or nil. 2) the return is a type variable. 3) α^- has only one comparison method successor.

struct-include $\{\mathrm{Array}\langle\tau\rangle \leq [\mathrm{include?} : \alpha \to \tau_r]\} \quad \cup C \Rightarrow$

$\{\alpha \leq \tau, \mathrm{Array}\langle\tau\rangle \leq [\mathrm{include?} : \alpha \to \tau_r]\} \cup C$

This is also a special rule based on the method name. Although the argument to Array#include? could be any type, this argument usually has the same type as the Array parameter. If the argument is a type variable, we add a constraint from the argument to the Array parameter.

4.3.5 Practical Solution Extraction

Figure 4.12 shows the practical solution extraction algorithm. This algorithm builds on the standard solution extraction algorithm and adds support for extracting solutions from the opposite sides. The changes are highlighted in pred_sol and succ_sol. In pred_sol, α is normally a positive variable and thus we extract its solutions from the predecessors. However, in case it has no predecessors but has

successors, then we extract its solution by merging its successors to intersection. Then, succ_sol is updated in a similar way, except if negative variable α has no successors, then we merge its predecessors to union. These modifications may help us to find solutions that are more specific and constrained, but resemble manual annotations.

In practical solution extraction, merge_to_union and merge_to_intersection are defined the same as in standard solution extraction, with one exception. In standard solution extraction, intersection merges A and B to nil. In practical solution extraction, this merge results in error.

Finally, recall that in standard type inference, we perform a solution simplification step after solution extraction to substitute variables with their solutions. In practical type inference, we also perform this solution substitution step after solution extraction. In addition, we simplify union types of the form A $\cup$ [foo: ...] to just A if foo is defined on A.

4.4 Implementation

Our type inference system is implemented purely in Ruby and piggybacks on a newer version Hummingbird that includes support for static type checking. This newer version no longer relies on OCaml and the Ruby Intermediate Language (RIL). Instead, we use Ruby's Parser gem to parse Ruby programs into abstract syntax trees. We next discuss the major challenges of implementing the type inference system.

```
1  preds(α) = ∀τ.τ ≤ α ∧ τ ∉ {α_i, τ_ι, τ_ρ}
2  succs(α) = ∀τ.α ≤ τ ∧ τ ∉ {α_i, τ_ι, τ_ρ}
3
4  def read_sol(α)
5    if α+
6      sol = pred_sol(α)
7    elsif α−
8      sol = succ_sol(α)
9    elsif α+−
10     if preds(α) = ∅
11       sol = succ_sol(α)
12     elsif succs(α) = ∅
13       sol = pred_sol(α)
14     else
15       sol = merge_to_intersection(succ_sol(α), pred_sol(α))
16     end
17   end
18   return sol
19 end
20
21 def pred_sol(α)
22   if preds(α) = ∅ and succs(α) ≠ ∅
23     sol = succ_sol(α)
24   else
25     sol = ⊥
26     preds(α).each {|p| sol = merge_to_union(sol, p) }
27   end
28   return sol
29 end
30
31 def succ_sol(α)
32   if succs(α)= ∅ and preds(α) ≠ ∅
33     sol = pred_sol(α)
34   else
35     sol = ⊤
36     succs(α).each {|s| sol = merge_to_intersection(sol, s) }
37   end
38   return sol
39 end
```

Figure 4.12: Practical Solution Extraction

Variable renaming. In the inference type system, we must be able to uniquely identify variable names. Thus, we rename all argument and return variable names to include their corresponding classes and method names. In addition, we uniquely rename type parameters based on their source locations. Further, if the annotated type of a method is an intersection type with shared type parameter variables, then we rename the variables in each arm to uniquely identify them.

Special params hashes. Our system supports the inference of key-value pairs of Rails specific params hashes. In Rails, there is one params hash associated with each web request, and there is generally one params associated with each method. Although our type checker implementation already provides the finite hash type, it complicates the inference process. Thus, during constraint generation, we track hash reads and writes through methods :[] and :[]= and use special variable renaming to avoid using finite hashes. Since we infer one params type per controller method, we create a special variable for each params key access, where this variable is identified by its class, method, and key.

For example, in foo(params[:name]), we set the call params[:name] to special variable *id*_params_:name, where *id* identifies the class and method. Then, the only constraint that we generate is that this variable is a subtype of foo's argument. In our test apps, params are accessed directly with keys that are symbols, so we are able to immediatey use the key names in variable names. In Rails, a session hash is used to save data across multiple requests. We also use a similar technique to infer Rail's session hashes, except we infer one session hash per Rails app.

Struct-include rule. Recall that we introduced the struct-include practical constraint resolution rule in Section 4.3.4. This rule states that if $\mathsf{Array}\langle\tau\rangle$ is a subtype of [include?: $(\alpha \to ...)$], then α is a subtype of τ. We listed this feature as a practical type inference rule for demonstration purposes. However, this is a special rule where the new constraint is instead added during constraint generation. The reason is that during constraint generation, if an actual class is a subtype of a structural type, then we can already look up the annotated type of the method in the structural type, so we do not keep $\mathsf{Array}\langle\tau\rangle \leq$ [include?: $(\alpha \to ...)$] after the type annotation lookup. However, we do add $\alpha \leq \tau$ immediately after the type lookup.

Dependent Rails controller methods. Recall that in a Rails controller class, params and fields are generally not shared across methods. One exception to this is that params and fields are shared between a method and its callees. Thus, we combine params and fields between these methods in the implementation. We also find that this feature helps us to infer much better types in some cases.

4.5 Evaluation

We evaluated standard type inference and practical type inference by applying them to three of the Ruby apps used in Hummingbird's evaluation: *Talks*, *Pubs*, and Credit Card Transactions (*CCT*), where *Talks* and *Pubs* are Rails apps. In this experiment, we inferred all method types that were statically type-checked with Hummingbird. In addition, we inferred all fields types, and Rails-specific params and session key-value pairs. We also assumed that type annotations existed on the

rest of the methods. At the end, we compared the inferred types to the previously manually annotated types.

Before discussing the results, we first point out the new type assumptions in our type inference system.

1) **Method-based Rails controller fields and params.** In Hummingbird, we made a simplifying assumption that like all normal classes, each field is shared by all methods within a controller class. Thus, we assumed that every field is tied to the class that defined it. Although almost all fields in different methods within the same controller class have same type in our Rails apps, each controller call gets a fresh set of fields. Similarly, in Hummingbird, we assumed that there is a single params hash per Rails apps, and annotated a single params hash with key-value pairs for the entire app. However, there are also generally no relationships between params in different controller methods, even though the vast majority of the params keys do have the same value types in our apps. In the current experiment, we instead infer fields and params based on controller methods, not classes, with the exception of the callees of a method. In our Rails app, there is a small number of controller methods with dependent methods in the same class. We combine fields and params types in such methods during type inference.

2) **More precise types for params.** In Hummingibrd, there was no direct support for finite hashes, and it was not possible to write down types that resembled nested finited hashes. Thus, params key-value pairs were annotated as an

intersection of the hash access method [], separated by keys, e.g.,

```
100 class Params
101   type  :[],  '(:key1) → v1'
102   type  :[],  '(:key2) → v2'
103   type  :[],  '(:key3) → v3'
104   ...
```

Thus, a few params key-value pairs were imprecisely annotated. For example, in *Talks*, there is a nested params hash params[:talk][:start_time], and :start_time is not a top-level key in params. Since there was no nested finite hash support, we annotated :start_time as another top-level key of the intersection, which incorrectly makes parmas[:start_time] valid. To rectify this issue in the current experiment, we infer finite hash types as well as nested hash types during type inference. As we have discussed earlier in the chapter, each params key access is represented with a special variable whose name includes the params key as well the method id m. For example, the inferred type of params[:talk][:start_time] is represented with params_m_:talk_:start_time.

3) **Numeric hierarchy.** Both our Hummingbird and type inference experiments were conducted with older versions Ruby. These versions of Ruby come with a Numeric hierarchy. For example, both Fixnum and Bignum are subclasses of Numeric. There are some numeric hierarchy differences between some of our inferred types and the annotated types. For example, there is one case where we inferred Fixnum as Fixnum ∪ Bignum. However, this numeric hierarchy has been eliminated in newer versions of Ruby. Thus, we do not distinguish between different number classes when recording statistics in our experiments.

4) **Tuples.** Tuples are used in the current experiment, but were not in Hummingbird. For example, one of the inferred types is $\mathsf{Array}\langle[\mathsf{List}, \mathsf{Symbol}]\rangle$, which is the best type for the variable. But this variable was previously annotated as $\mathsf{Array}\langle\mathsf{Array}\langle\mathsf{List} \cup \mathsf{Symbol}\rangle\rangle$, which then required type casts for the type checker to pass.

5) **Initialize methods.** We now infer argument types of object creation methods, or initialize methods, which were not annotated in Hummingbird. Note that we do not infer return types of intialize methods as they are not explicited used.

To have a fair comparison between the previously annotated types and the inferred types, we assume that the annotated types have been adjusted to meet the above changes in type inference.

4.5.1 Overall Results

Overall, we find the practical type inference system very effective compared to standard type inference. The number of inferred types that are at least as good as the previously manually types are 98%, 94%, and 83%, respectively, even without full access to the application code in the last two apps. On the other hand, the corresponding numbers are only 67%, 70%, and 63% with standard type inference.

Table 4.2 summarizes the overall results of applying both standard and practical inference compared to the manually annotated types in Hummingbird.

The "Total" column lists the total number of types we have inferred, including

App	Total	Match	Struct	Partial	None	Other
CCT	58	57 / 39	- / 14	- / -	- / 5	1 / -
Pubs	124	117 / 87	4 / 30	3 / 5	- / 2	- / -
Talks	349	290 / 219	12 / 45	12 / 12	23 / 67	12 / 6

Total = # types inferred
Match = # types at least as good as annotated
Struct = # struct types
Partial = # types with incomplete info
None = # types with no useful info
Other = # types that do not fit in the previous categories

Table 4.2: Practical vs. standard inference results.
(practical on left, standard on right, separately by /)

argument types, return types, fields, and params and session types. For params and session types, we count each key-value pair as one type. In our Rails apps, there is a very small number of empty methods and Rails controller methods with no valid routes; these methods are ignored.

For all columns except the "total" column, we show two counts separated by /, for practical type inference and standard type inference, respectively.

Then, the "match" column lists the number of inferred types that either match the annotated types exactly or are better than the annotated types. During type comparisons between the inferred types and the annotated types, we found a small number of incorrectly annotated types. In these cases, we consider the inferred types better than their annotated types. These incorrect type annotations were not caught by our type checker because the correct return types are subtypes of the incorrectly annotated return types.

The "struct" column includes actual classes that are inferred as structural types, including generic types with one or more structural type parameters.

The "partial" column lists the number of types that provide imcomplete information, including possible types, implication types, and generic types with incomplete parameter information, e.g., Array⟨User⟩ inferred as Array⟨⊥⟩.

Next, the "none" column lists the number of types that we were unable to infer, which includes ⊤ and ⊥ for types that should have been more precise.

Finally, the "other" columns lists the number of types that do not fit in the previous categories. We will provide details on the "other" types found in our apps when we discuss the the results on each application.

We now discuss the inference results on each of our apps in detail.

CCT. We begin by discussing the results of practical inference. In this app, approximately 98% of the types from practical inference are at least as good as the annotated types. In fact, the inferred types reveal five incorrect annotations caused by two mistakes.

First, consider the type of @file in CSV.foreach(@file, col_sep: SEPARATOR). We annnotated @file as File, which appears correct. However, the type of @file should match the annotated type of the first argument of CSV.foreach. This annotated type is actually a broader type File ∪ String, which matches the inferred type.

The next case is a slightly complicated. Consider the simplified version of a method in the app. In this method, we know that the annotated type of @instructions is Array⟨Hash⟨:a ∪ :t, String⟩⟩. We also know that the return type of the whole method is an Array whose elements have the type of the last statement in the method's block. During manual annotation, we mistakenly assumed that if the last

```
1  def filter_transactions
2    @instructions.map do |x|
3      type = x[:t]
4      x if type != ADD
5    end.compact
6  end
```

statetment does not execute, then the type of the element is then the type of the previous statement in the block, or the type of type = x[:t]. This means that the type of each Array element is the union of the two statements in the block. Since the type of x[:t] is String, we incorrectly annotated the return type of the method as Array⟨Hash⟨:a ∪ :t, String⟩ ∪ String ⟩. On the other hand, we correctly infer this type as Array⟨Hash⟨:a ∪ :t, String⟩⟩.

In this app, there are no "struct", "partial" or "none" type matches. But there is one type that does not fit in any of the previous categories. Consider another simplified set of methods in our app in Figure 4.13.

```
1  def parse
2    ...
3    CSV.foreach(@file, col_sep: SEPARATOR) do |r|
4      ... = to_downcase(r)
5    end
6    ...
7  end
8
9  def to_downcase(row)
10   row.map! { |x| x.to_s.downcase }
11 end
12 ...
13 parse()
```

Figure 4.13: CCT Methods.

During type inference, we first infer Array⟨String⟩ as the type of r, based on the annotation of CSV.foreach. Then, to infer the type of to_downcase's argument

row, we normally consider its successors. In this case, the only successor is the structural type with map!, but too many actual classes have map! as an instance method. Recall that in practical type inference, one goal is to eliminate structural types when approriate, and practical inference allows us to instead consider row's predecessors in this case, or r. Thus, we infer row as Array⟨String⟩, which appears correct. In fact, we examined all library files in the app and there are no other types passed to row because all calls to to_down are invoked from parse. However, during manual type annotations, we assumed that to_downcase can be called directly, and we annotated the type of row as Array⟨Fixnum ∪ String⟩. This is because each element of row can also be a Fixnum for to_downcase to run. In fact, there is a unit test in the app that directly calls to_downcase with an array of numbers and the test runs correctly. This suggests that occasionally inferring types from the opposite direction is not ideal.

We now discuss the overall standard inference results. The number of "matches" from standard type inference is only about 67% of that from practical inference. This is mostly because there is a large number of method arguments that are inferred as structural types. In standard inference, we have some "None" types, which includes $\top$ for a few arguments and fields, and $\bot$ for a few returns. These $\top$'s are the results of ignoring types on the opposite directions of the constraints when types on the normal sides have no useful information. The return types are $\bot$ because the last statements in the methods are calls from structural type callers, and we were unable to look up annotated types from non-actual classes.

Finally, recall that we found one "other" type in practical type inference where

we inferred Array⟨Fixnum ∪ String⟩ as Array⟨String⟩. In standard inference, this number is 0 in because we inferred the same negative variable by considering its successors, which is the type below

```
1   [ map!: () {([ to_s: () → String ]) → String} → Array<String> ]
```

This is a type that has already been accounted for as an "struct" type. Although this type is correct, it is harder to understand than one or two actual classes.

Pubs. *Pubs* is a Rails app, which means that the list of total types includes params key-value pairs. In our Rails apps, params keys almost always appear as symbols, and we rely on these symbols to generate special variables that specify key names during constraint generation and inference. In this app, there are 13 params key-values pairs where the keys directly appear as symbols. Then there is one controller method where keys are not used as symbols.

```
1  def update
2    if params[:new_tag] ...
3    params.keys.each { |k|
4      next unless k =~ /^tag_(.*)/
5      t = Tag.find($1)
6      ...
7  end
```

In this method, params appears only in the first two lines of the method. The first line tells us that :new_tag is a params key. On the second line, we iterate through the params keys and compare each key with a regular expression. If the key is of the form :tag_i, where i is an integer, then the method extracts the value of i and feeds it to Tag.find. This suggests that the method's params has an unknown number

of keys, ex. :tag_1, :tag_2, :tag_3, ... In fact, RDL currently has no support for representing a type with variable keys that meet certain conditions. Thus, we count :new_tag as the only valid key in both inference and type annotations.

Again, we begin by discussing the results of practical inference. The "match" column shows that the vast majority of all types inferred are at least as good as the annotated types. In this app, we also discovered types that had been incorrectly annotated.

First, consider the method below.

```
1  def self. fix_periods (out)
2    while (out =~ /\.\./)
3      out.sub!(" ..", ".")
4    end
5  end
```

In this example, we incorrectly annotated the method's return as String. We mistakenly assumed that the method's last statement is the last statement inside the while loop, which evaluates to String. However, the actual last statement is the entire while statement, which always evaluates to nil in Ruby, and matches its inferred type.

Now consider

```
1 def self.empty(s)
2   return s == nil || s == ""
3 end
```

In this method, we inferred s as String. We also examined the app and found no other types used as the method's argument. However, we previously annotated s as Object. Based on the method alone, it appears that Object is the right type.

However, this is a Rails app where we do not expect the programmer to call this model helper method directly. In fact, we only expect this method to be invoked through methods that pass String to it.

Next, the "struct" types in this app include structural types and generic types with structural parameters. For example, there are a few cases where the annotated type of a params is:

params_:paper_:month = String,

params_:paper_:month_other = String,

params_:paper_:tag_list = Array⟨String⟩,

and params_:paper_:tags = Array⟨Tag⟩,

meaning that params has one top level key :paper, and its value is another params hash with keys :month, :month_other, :tag_list, and :tags. In this case, we infer the values of :month, :month_other, and :tag exactly as the annotated types. However, we infer a structural type for :tag_list,

```
1 [ split : (String) → [ map: () {([ strip : () → ⊥ ]) →⊥} →
2 [ each: () {(String) → Array<Tag> or String} →⊥ ] ] ]
```

because there are too many actual classes that match this structural type.

The "Partial" column shows that there are three types with incomplete information.

First, consider the type of @tag_name.

```
1 def publications
2   @tag_name = :all
3   ...
4 end
```

In this method, @tag_name appears only once and it is obvious that this type is :all. However, we annotated it as Symbol.

Now consider

```
1  type :Paper, :self.find, '(Fixnum or String) → Paper'
2  type :Paper, :self.find, '(Array<Fixnum or String>) →Array<Paper>'
3  ...
4
5  def bibtex
6    @paper = Paper.find(params[:id])
7    render :layout ⇒ false, :content_type ⇒ "text/plain"
8  end
```

In this method, we infer "partial" types on both @paper and params[:id]. We infer that the type of params[:id] is a possible type that is either Fixnum ∪ String or Array⟨Fixnum ∪ String⟩, based on the annotated intersection type of Paper.find. We are unable to choose the valid type from the list of possible types because there is not enough information on the variable. As a result, we infer that @paper is an implication type

params[:id] ≤ Fixnum ∪ String → Paper

params[:id] ≤ Array⟨Fixnum ∪ String⟩ → Array⟨Paper⟩.

We now discuss the standard inference results. In this app, only about 70% of types inferred using the standard approach are at least as good as the annotated types, compared to approximately 94% with the practical approach. Like *CCT*, there is also a large number types that are inferred as structural types, and the majority of these types are argument types.

The "partial"types from standard inference includes two additional types not included in the practical inference. One of these types is a possible type, and the

other is a params type where we were unable to infer key-value pairs because the constraint resolution rule on inferring nested hashes is not available in standard inference.

The "none" column includes two types in the method below.

```
1  def adjust(p)
2    if p[:paper][:month] == "other" then
3      p[:paper][:month] = ...
4    end
5    ...
6    p[:paper][:tags] = tags
7  end
```

Unlike practical inference, we are unable to resolve the method argument p to an actual special hash type params. In fact, in standard inference, we infer p as []: (:paper) $\rightarrow \bot$, which also means that we were unable to look up the annotated type of [] in the last statement, which resolves the return type to nil. Morever, with practical inference, we were able to infer the type of p[:paper] as another params with key-value pairs for :month and :tags. Again, since p is inferred as a structural type with standard inference, we also infer the key-value pair of p[:paper] as $\bot$.

Talks. The "total" column shows that *Talks* is a much larger app than the previous two apps. Like *pubs*, there are three controller methods where the params keys are compared against regular expressions. Again, since it is not possible to precisely represent these keys, we do not include such key-value pairs in our statistics. Despite this, we still include 69 key-value params pairs in the total number of types.

Again, we start the discussion with the practical inference results.

The "match" column shows that approximately 83% of the inferred are at least

good as the annotated types, which is a lower percentage than the previous two apps. The main reason is that in this app, we do not have access to the full application code. As a Rails application, *Talks* contains a set of embedded Ruby templating (ERB) files in addition to the regular Ruby files. We do not currently support extracting information from ERB files, as it can be very complicated to process such files. For example, the embedded code indicates that @owners $\leq$ @users. But it

```
1  <%= render :partial => "shared/ expanding_list",
2  : locals => { :name =>"owner", : current_elts => @owners, : all_elts => @users, ..
```

is very difficult to extract this constraint as it involves reasoning behind :current_elts and :all_elts, which are used with certain web form helper methods in complicated code.

The inferred types in this app also revealed a few incorrect annotations. First, there is a simple method with just one statement: return false. It is obvious that this return type is False, but we annotated it as True $\cup$ False. Note that in Ruby, true and false belong to different classes.

Second, we incorrectly annotated a method's return type, mostly likely due to a simple typo. The last statement in the method is a method call that clearly returns only String, which matches its inferred type. However, we incorrectly annotated this type as String $\cup$ Array$\langle$String$\rangle$.

We now describe an incorrect annotation on a more complicated method.

In this case, we incorrectly annotated the return of this method as Hash$\langle$Talk, Symbol$\rangle$, but the Hash value can be more than just Symbol. Notice that there are two places where we assign types to this value, s.kind and :kind_subscriber_through.

```
1  def subscribed_talks(range, filter = [:kind_subscriber, :kind_watcher, ....])
2    talks = {}
3    subscriptionns.where(...).map {|s|
4      t = Talk.find(s.subscribable_id)
5      talks[t] = s.kind if filter.member?(s.kind)
6    }
7    ...
8    l.talks.each {|t|
9      talks[t] = :kind_subscriber_through
10   }
11   return talks
12 end
```

It is obvious that we assign a Symbol to the value in the second case. Now, in the first use, the type of s.kind is the type of Subscription#kind. We mistakenly assumed that the type of s.kind is Symbol, because in the first code block, we test to see if it is a member of the default argument filter, which includes only Symbols. However, we failed to consider the type of Subscription#kind in the database schema, which indicates that that it also be a String. During type inference, we correctly infer Symbol $\cup$ String for this type.

Next, the "struct" column shows 12 out of the 349 total types inferred are structural types. Again, we were unable to convert these structural types to actual classes because each one of the structural types has a large number of matching classes. However, we have examined the ERB files and expect these files to provide additional constraints on all 12 variables to help us reduce the number of structural types.

In this app, we also have a list of "partial" and "none" types, where "partial" includes generic types that includes $\bot$ parameters, possible types, and implication types, and "none" again includes just $\top$ and $\bot$. For both classes of types, we also

expect the ERB files to provide additional constraints on these types to improve the preciseness on the majority of these types, including both inferring actual classes, generic types with actual class parameters, and structural types. For example, consider

```
1  def send_admin_message(u, h)
2    @message = h[:message]
3    mail :to ⇒ "#{u.name} <#{u.email}>",
4         :subject ⇒ h[:subject],
5         :from ⇒ "Talks <talks@cs.umd.edu>"
6  end
7
8  # ERB
9  <%= @message.sanitize %>
```

Consider the type of @message. There are no useful constraints on this variable in the regular ruby files, thus its inferred type is $\top$. We could use the ERB file to infer that @message is a structrual type with sanitize defined on it. But we cannot currently convert this structrual type to an actual class, as there are 19 classes whose instance methods inlude santize.

The list of "none" types also includes one argument type used in string interpolation. Our parser does not currently detect string interpolations.

Finally, the "other" column shows a list of 12 types. Some of these types are caused by an a structural to nominal conversion, where the structural type is [email: () → _, name: () → _]. There are two actual classes that match these types: Registration and User. Recall that the structural to nominal rule is applied if one or two classes match. Again, the reason behind this rule is that based on our apps, each structural type either matches one class, two classes, or a much larger

number of classes. The union type we inferred would not actually raise exceptions at run-time. On the other hand, we expect only User used as this structural type, which is also its previously annoated type.

Another "other" case is on the result the Ruby && operation. Consider

```
1  def watcher?(user)
2    s = subscription(user)
3    return s && (s.kind == :kind_watcher)
4  end
```

The annotated return type of this method is true ∪ false. During type inference, we know that s is Subscription, if not nil. In Ruby, Object && Bool evaluates to the value of Bool. However, there is a limitation in the type checker that evaluates the last method in the statement to the type of s.

The rest of the "other" types come from this method

```
1  def fix_range(p)
2    p[:range] = :current unless p[:range]
3    p[:range] = p[:range].to_sym
4  end
```

In this method, we infer that the method's argument is a params with just one key :range and the return type is :current, which appear correct if we consider this method application alone. However, we did not incorprate type constraints passed to p from other controllers, which includes params with more key-value pairs. Recall that we use constraints on an argument from the reversed direction only if the argument's successors are all structural types or if there are no sucessors, which does not apply in this case.

Finally, our standard inference results show that like the previous two apps, the

App	Total	Match	-SN	-Meth	-Rev
CCT	58	57	57	57	55
Pubs	124	117	101	108	117
Talks	349	290	246	271	281

-SN = matches w/o structural-to-class conversion
-Meth = matches w/o struct-cmp rule
-Rev = matches w/o reversed solution extraction

Table 4.3: Practical Inference Features

number of "Match" types is much smaller than that with practical inference, because standard inference infers more "struct", "partial", and "none" types. The number of "other" types is much smaller than the corresponding number from practical inference. This is because although the structural to nominal conversion in practical inference is generally very helpful, it may ocassionally introduce errors, as mentioned in the Registration ∪ User case. By keeping structrual types as is, we have reduced the number of "other" types.

4.5.2 Importance of Key Practical Inference Features

In this subsection, we discuss the importance of the key practial inference features.

Table 4.3 reports the practical inference results from removing three key features. The "total" and "match" columns are copied from the overall results table. Again, the "total" column shows the total number of types inferred, and the "match" count is the number of inferred types that either match the annotated types or are better than the annotated types. For the rest of the columns, we report the total number of matches after removing each of the three key features. The "-SN"

column shows the number of matches after the structuralvtype-to-actual class conversion rule has been removed from constraint resolution. The "-Meth" column reports the number of matches after the method name based constraint resolution rule of struct-cmp has been removed. The "-Rev" count reports the number of matches after the reversed solution extraction feature has been eliminated.

We found these features generally very helpful with increasing the number of matches in the Rails apps. However, these key features have minimal effect on CCT as many of its matches rely on the generic type parameter assignment rules.

CCT. The "-SN" column shows that removing the structural type to actual class conversion rule did not actually change the number of matches. Although this conversion rule would have converted a few structural types to actual classes, we achieved the same results on their dependent variables through the normal transitivity rule (trans) and reversed solution extaction. In addition, the "-Meth" column also shows no change in the number of matches. The "-Rev" count shows a decrease of 2 matches after the reversed solution extraction has been eliminated. All 2 types are arguments that result in $\top$.

Pubs. The "-SN" column shows that removing this conversion rule eliminated 16 matches, all of which resulted in structural types instead of actual classes.

The "-Meth" column shows that the number of matches decreased by 9 after we removed the struct-cmp rule. In addition, this test also reveals a case where struct-cmp fails to infer the right type. Recall that the struct-cmp rule states that if

a variable is a subtype of a structural type with methods such as ==, then the type of the variable is the type of the argument to ==, which is the most likely case. We now revisit Figure 4.4, where struct-cmp rule infers that the type of the argument is String, which appeared to be the right type. After we removed the struct-cmp rule, the argument became String ∪ Fixnum through the normal transitivity rule. We then verified that there are in fact calls to this method from Fixnums, and the later inferred type is correct. In addition, we found that out of the 55 calls to empty, 51 have String arguments, and 4 have Fixnum arguments.

Finally, although there is no change in the number of matches in the "-Rev" count, there are 3 types that are more precise compared to the original types with reversed solution extraction. For example, before removing the reversed solution feature, one of the inferred types is split: (String) → [map: () ([strip: () → ⊥]) → ⊥ → [each: () (String) → Array⟨Tag⟩ ∪ String → ⊥]]]. After the removal of this feature, this type becomes [split: (String) → ⊥]. Although the former type is very complicated to understand, it does contain more information than the later type.

Talks. The "-SN" count includes a few interesting cases. First, recall that in the overall evaluation results section (Section 4.5.1), we showed a case where we converted a structural type to Registration and User, even though the only expected actual class is User. Interestingly, after removing the structural type to actual class conversion rule, we were able to correctly infer User as opposed to Registration ∪ User as the type of variables. These correct types were generated from reversed

```
1   # Array is parameterized by u
2   type :Array, :[], '(Fixnum) →u'
3   type :Array, :[], '(Range<Fixnum>) →Array<u>'
4
5   def organize_talks (talks)
6     h = Hash.new
7     h[:past] = []
8     h[: later_this_week ] = []
9     (0..6). each { |wday| h[: later_this_week ][wday] = [] }
10    ...
11    talks .each {|t|
12      if t. start_time ≤ the_past
13        h[:past] ≪t
14      elsif later_this_week .cover? t. start_time
15        h[: later_this_week ][t. start_time .wday] ≪t
16      else
17        ...
18      end
19    }
20    h[:past]. sort ! { |a,b| a. start_time <=> b.start_time }
21    ...
22  end
```

Figure 4.14: Talks method

solution extraction and the solution simplification rule that simplifies a type of the form A ∪ [foo: ...] to A if foo is an instance method of A.

Another interesting case is that deleting the structural type-to-actual class conversion rule caused a type error with the method shown in Figure 4.14. First, suppose that this conversion rule still exists in the inference system. In the block of talks.each, the conversion rule sets t to Talk because Talk is the only class with start_time as an instance method. Then on line 20, we know that each element of h[:past] is t or Talk, and thus a and b both have type Talk. Now consider the case where the conversion rule is removed. On line 15, the type of h[:later_this_week] is Array from the initialization on line 8. We now explain how we fail to resolve the

type of h[:later_this_week][t.start_time.wday]. To infer this type, we look up the type annotation of Array#[], which is the intersection type shown on lines 2 and 3. But we are unable to choose the appropriate type from this intersection type because we do not know the actual class of t, and thus the type of t.start_time.wday is unknown. In this case, our inference system delays solving for the argument t.start_time.wday in hopes of getting future constraints on this variable. Unfortunately, this is a case where it is not possible to gather future constraints to resolve the argument to an actual class. Our type inference system then converts the annotated intersection type to a single regular method type whose argument is the union of the arguments in the intersection type, and the return type is the union of the returns in the intersection, or Fixnum $\cup$ Range$\langle$Fixnum$\rangle$ $\rightarrow$ u $\cup$ Array$\langle$u$\rangle$. This means that each element t of h[:later_this_week][t.start_time.wday] is u $\cup$ Array$\langle$u$\rangle$. Let t be Array$\langle$u$\rangle$, since we append t to h[:past] on line 13, each element of h[:past] can also be Array$\langle$u$\rangle$. But this means that on line 20, both a and b can be Array$\langle$u$\rangle$. This is a type error on a.start_time because Array does not have start_time defined on it.

Other than the above cases, removing the structural type to actual class conversion rule causes a large decrease in the number of matches. In most of the non-match cases, the new types are either structural types instead of actual classes or $\perp$ instead of actual classes.

The "-Meth" count shows a decrease of 19 matches after the struct-cmp rule has been removed. The new results include more structural types as well as new $\perp$'s.

Finally, the "-Rev" count shows a decrease of 9 matches after the removal of

App	Orig	Std Type Inf	Prac Type Inf	Std Or Ratio	Prac Or Ratio
CCT	0.62s	2.19s	7.31s	3.53×	11.79×
Pubs	2.96s	8.08s	15.19s	2.73×	5.13×
Talks	57.47s	82.30s	115.21s	1.43×	2.00×

Table 4.4: Type Inference Running Time

the reversed solution extraction feature, where the new types are either structural types or $\perp$ as opposed to the original actual classes.

4.5.3 Efficiency

Table 4.4 reports the overhead of running standard and practical type inference with the apps. The "Orig" column shows the running time without type inference. The "Std Type Inf" and "Prac Type Inf" columns report the running time with standard and practical type inference, respectively. The "Std Or Ratio" and "Prac Or Ratio" columns list the ratio of standard type inference running time and the "orig" column, and the ratio of practical type inference running time and the "orig" column, respectively. We found the ratios range from 2x to 11.79x for practical type inference, and 1.43x to 3.53x for standard type infereence.

For *CCT*, we measured the running time of the app's test suite. For *Pubs* and *Talks*, we measured the running time of the controller test suite. For all apps, we performed each measurement five times and took the arithmetic mean. Investigating further, we found that if the struct-to-nominal constraint resolution rule is removed from practical inference, then the runninng time becomes very similar to that of standard inference for all apps. This rule often iterates through a very large set of

classes to select classes that match the structural types, and we expect performance can be improved with further engineering effort. Overall, the practical type inference overhead is much better than Rubydust [1], a run-time type inference system for Ruby.

4.6 Related Work

Many researchers have investigated type inference for Ruby. The closest work to ours is DRuby [8], a purely static type inference system for Ruby. Our type inference system is based on the DRuby type system, which includes features such as union, intersection, generic, and structural types. However, DRuby lacks advanced features such as implication types, possible types, delay operations, and the practical constraint resolution and solution extraction rules. In addition, unlike our flow-insensitive type inference system, DRuby includes a parser [22] that renames local variables to model their flow sensitivity. Subsequent work on type inference for Ruby include $\mathcal{P}$Ruby [7], DRails [25], and RubyDust [1]. $\mathcal{P}$Ruby is a profile-guided extension to DRuby that analyzes Ruby's highly dynamic constructs. To use $\mathcal{P}$Ruby, the programmer first runs the program to record the dynamic behavior, such as methods passed to `send` and strings passed to `eval`. These profiled strings are then used with DRuby to analyze the program. DRails extends DRuby to bring type inference to Rails. DRails statically analyzes the program and converts Rails implicit calls to explicit Ruby code. The modified code is then type checked by DRuby. RubyDust is a flow-sensitive type inference system based on dynamic ex-

ecutions. In RubyDust, each run-time value is wrapped with a type variable, and wrappers generate constraints when the wrapped values are used. RubyDust infers sound types as long as all paths are observed at run-time. Furthermore, the RubyDust experiments show a signficantly higher performance overhead than our type inference system. In RubyDust, the ratios between the solving time and the original running time range from 17× to over 1000×. DRuby and RubyDust share similar typing features and algorithms for constraint solving.

There have also been efforts in bringing static type systems to other dynamic languages. Palsberg et al [50] present a type inference system for a SmallTalk-like language. In this system, types are sets of classes, and subtyping is set inclusion. The system constructs a set of conditional type constraints and computes the solution by least fixed-point derivation. Although the type system is simple, it also performs actual class lookups based on method calls, like our struct-to-nominal rule in the practical inference system. However, the reason for this lookup in the set-based system is the lack of structural types. In addition, our struct-to-nominal rule has more advanced features such as reducing the number of matching classes. This paper also points out that there is no actual implementation of this set-based type inference system. Oxhoj et al [49] subsequently present an improvement of this set-based type system, mainly to support collection classes such as List from expanding inheritance. For example, a list of integers is defined as IntList, a subclass of List. The method definitions in List are duplicated in IntList. In our system, this integer list is represented as a generic type List⟨Int⟩.

Agesen [51] proposes the cartesian product algorithm, a set-based type in-

ference algorithm that analyzes each send with multiple sets of argument types separately. For each send, the algorithm first computes the cartesian product of the receiver and the arguments, and then analyzes each tuple in the product separately. The method below is a simplified version of an example taken from Agesen's paper.

```
1  def mod(self, arg)
2    ...
3      x = div(self, arg)
4    ...
5  end
```

In this method, div is an integer division method that fails on floats. The paper assumes self and arg are both {Int, Float}, and thus the cartesian product is {(Int, Int), (Int, Float), (Float, Int), (Float, Float)}. The algorithm infers that self and arg can only be (Int, Int) because all other tuples fail on div. While this result is precise, it is unclear whether the assumption of having Float as the type of the receiver or argument may indicate a type error elsewhere in the program. In addition, this system does not include intersection types and does not infer types that resemble intersection types. The result of each send is simply the union of all results from the separate tuples. Although our type inference system includes logic to choose a single type from an intersection type annotation, we also do not currently support the inference of intersection types. Madsen et al [52] subsequently present Ecstatic, a type inference system for Ruby based on the cartesian product algorithm. In Ecstatic, types are again sets of classes, and Ecstatic does not support many types of code blocks. Moreover, the test applications used with Ecstatic only depend on

Ruby core and standard libraries.

Ancona et al [15] present RPython, a subset of python that is statically typed and can be compiled for the CLI and JVM platforms. The Translation Toolchain in RPython performs static type inference. This inference tool appears to have a limited type language. For example, RPython forbids a method to return different types on different branches. On the other hand, RPython provides some metaprogramming support through a separate initial, load time phase, during which highly dynamic features may be used. After this phase, RPython performs type inference and no longer allows dynamic modifications of classes and methods. Aycock [4] presents aggressive type inference for Python. He performs experiments to show that in Python programs, only a small percentage of variables have flow sensitivity, and dynamic features are rarely used. Thus, he makes an aggressive but simplifying assumption that variables maintain a constant type in Python programs, which makes static type inference possible without features such as union types. Salib [10] proposes StarKiller, a static type inference system and compiler for Python. The algorithm is based on the cartesian product algorithm, and handles data and parameteric polymorphism. In addition, Starkiller handles foreign code interactions with programmer provided descriptions. The system does not handle dynamic features.

Hackett et al [54] present a hybrid type inference approach that combines unsound static type inference with dynamic checks for JavaScript. The dynamic checks account for special cases such as numeric overflow and undefined values from out of bounds array access, which may adjust type assumptions. In contrast, our type inference system does not consider these special cases, and such errors lead to

runtime exceptions. Cartwright et al [53] propose soft typing for an ML language. During type inference, the system also automatically inserts explicit runtime checks on "suspect" type errors. The key idea is that possibly ill-typed programs may still execute. On the other hand, our type inference system does not insert any runtime checks, and the statically inferred types are final. If our type inference system fails to unify types during static analysis, then the system reports an exception and halts execution.

Heintze [56] describes a type inference system where program variables are sets of values. In this system, types include unions, intersections, as well as projections, complementations and quantifications. However, the system ignores inter-variable dependencies of the form $\mathsf{x} = \mathsf{a}$ iff $\mathsf{y} = \mathsf{b}$ for simplicity. The paper shows that the set constraints can be described with regular grammars, and computing the least model of the constraints is decidable. Although our type system does not consider inter-variable dependencies from conditionals, we do consider inter-variable dependencies based on intersection type annotations, which are necessary for some of our apps to type check.

Aiken et al [55] present another soft typing system that performs ML style type inference. The most novel feature of this system is conditional types, which are used to model precise control flow. The result of a case statement only includes the types of reachable branches. Since types are sets in this system, a branch is considered reachable if the intersection type of the case variable and the set of all values that match the branch guard is non-empty. Although our type inference system does not model such control flow, we use implication types in conjuction with possible types

to select reachable arms from intersection types.

Thiemann [13] presents a type system that tracks type convertibility for a subset of JavaScript. The system does not consider features such as polymoprhism, and there is no implementation of the system. Anderson et al [3] introduce a type inference system for a subset of JavaScript. The system models runtime modifications of objects through assignments to members of the object. However, Ruby does not support such modifications.

Finally, Tobin-Hochstadt et al [44] present a type system for Scheme that uses propostional logic to infer more precise types based on the branch guards. Our type inferencce system uses implication and possible types to infer more precise types by selecting a single type from an intersection type annotation. However, we do not currently consider conditional predicates prior to the ocurrence of the types. Guo et al [48] propose a system that infers abstract types based on the interactions between types. Although Guo et al's system has only been applied to C and Java programs, the ideas could be applied to dynamic languages as well. In this system, an abstract type is a group of program variables that interact in the program. The system assumes that the two operands of a comparison operator interact and belong to the same abstract type. This idea is similar to our method-name based constraint resolution rule **struct-cmp**, where we may add subtyping constraints between the receiver type and the argument type of a comparison method call.

Chapter 5: Conclusion and Future Directions

In this dissertation, I presented several pieces of work showing how we could bring some benefits of static types to dynamic languages.

First, I described the Ruby Type Checker (rtc), is a purely dynamic tool that adds type checking to Ruby. Rtc type checks values at method entrance and exit, which is later than a purely static system, but earlier than a traditional dynamic type system. Rtc supports type annotations on classes, methods, and objects. In addition, rtc includes union and intersection types, higherorder (block) types, and parametric polymorphism among other features. Furthermore, programmers can control where type checking occurs to reduce runtime overhead.

Next, while rtc is effective in type checking, it has trouble dealing with metaprogramming, which generates code as the program executes. Thus, I developed Hummingbird, a run-time static type checking tool that type checks Ruby code even in the presence of metaprogramming. In Hummingbird, type annotations execute at run-time, including type annotations created by arbitrarily complex metaprogramming. When a method is called, Hummingbird statically type checks its body in the current dynamic type environment. Moreover, Hummingbird uses a caching system to reduce performance overhead.

Finally, to reduce the burden of manual annotations required with rtc and Hummingbird, I presented a practical type inference system. The main goal of this system is to infer types that are concise, meaning the types are precise enough to cover all expected classes while appearing as short as possible. In addition, our practical inference system provides better support on intersection and generic types. The inference system gathers constraints on a method's body as soon as it is called at run-time. After all constraints have been gathered, the inference system uses a set of advanced constraint resolution rules and an unconventional solution extraction algorithm to infer types. We applied our practical type inference approach to three apps and have shown it to be very successful, even without access to the full app code.

High Level Results and Conclusions Throughout the dissertation I studied how to bring the benefits of static typing to dynamic languages. I presented three tools for Ruby, including a dynamic type checker, a just-in-time static type checker, and a practical type inference system. I applied all three tools to various Ruby programs and obtained promising results. I conclude that these specialized type systems effectively increase the type safety of Ruby programs with static typing features. In addition, the ideas of these type systems can be ported to other dynamic languages.

Future Work Rtc, Hummingbird, and the practical type inference system have several limitations we plan to address in future work.

More method name-based constraint resolution rules. In this dissertation, we introduced some method named-based constraint resolution rules, including rules on comparison methods, hash access methods, and methods such as include?. We plan to add more varieties of method-based rules.

Eliminating unlikely classes that match structural types. We found the structural-to-actual class conversion to be very helpful in practical type inference. However, we encountered many structural types with too many actual class matches during evalution, especially with the *Talks* apps, where we did not have access to the full code. We plan to develop a set of special rules to help us eliminate matching classes that are highly unlikely to be the expected actual classes. For example, we could simply eliminate matching classes that are not defined in the application and the Ruby core library.

Backtracking support on intersection types. The type inferencce system currently use possible types, intersection types and delay operations to help us resolve intersection types to a single type. Alternatively, we could subsume all three features by developing a backtracking system to resolve intersection types. In particular, whenever we are unable to choose the right arm of an intersection type, set it to one of the types and continue with the inference as unsual. If we encounter a type error, then backtrack and choose another arm of the intersection type. In theory, it may be difficult to design such a backtracking system efficiently as a variable may rely on many intersection types where each intersection type has many arms. However, in practice, most

```
1  case v
2  when Fixnum
3    x = A.new
4  when String
5    x = B.new
6  else
7    x = C.new
8  end
```

variables appear to have no intersection type dependencies, and variables with such dependencies tend to depend on only one intersection type. Furthermore, intersection type annotations could be rewritten to so that the arms are listed from the most common case to the least common case.

Intermediate variable inference elimination. The sequencing of type inference rule states that the type of a sequence of statements is the type of the last statement. This suggests that we can eliminate the inference of certain intermediate types. However, we may not catch certain type errors with this elimination.

The Not type. We plan to infer more precise types by introducing the Not type into our system. A Not type represents the guard's type in the else branch of a case or if statement For example, in the else branch below, we know that v is not a Fixnum or String. In addition, the Not type can be used in conjunction with implication types. We know that one implication for x is v $\neq$ {Fixnum, String} $\rightarrow$ C.

Intersection type inference. Although our inference type system can extract information from intersection type annotations during inference, we do not sup-

port the inference of method types that are intersection types. We plan to add this feature in the future.

Inference with incomplete set of annotations. Recall that our type inference system assumes that we have existing type annotations on the dependent methods. In the future, we plan to test the effectiveness of practical type inference with incomplete sets of dependent method annotations, as well as developing new techniques to infer practical types with partial dependent method annotations.

User study. We plan to conduct a user study to evaluate the understandability of the inferred types.

A richer type language. Although the current type language supports many categories of types, it is still not expressive enough to represent some types. Recall that it is not possible to write down a type for Array#flatten, wich returns a new array in which arbitrary depth nestings of the array have been removed from the caller. We plan to enrich our type system to support such types.

Runtime types with static types Recall that in the practical type inference system, we do not consider runtime types during static analysis. We plan to explore the benefits of incorporating runtime values into static analysis. For example, if we cannot resolve a type during inference, then we could use its runtime type. In addition, we can also use runtime values to catch type errors if the inferred types do not cover all expected classes.

Chapter A: Appendix

This chapter contains the full definitions and proofs for the formalism in chapter 3.

We show soundness by first showing preservation and progress. As is typical, the hardest part of the proof is preservation, which shows that an expression's type is preserved under a step in the dynamic semantics. To make the theorem work, we also need to reason about preserving key properties about the typing environment, run-time stack, and cache. Here is the statement of the theorem, which we explain in detail next:

Theorem 2 (Preservation)**.** *If*

(1) $\langle X, TT, DT, E, e, S\rangle \rightarrow \langle X', TT', DT', E', e', S'\rangle$

(2) $TT \vdash \langle \Gamma, e\rangle \Rightarrow \langle \Gamma', \tau\rangle$

(3) $\tau \leq TS$

(4) $\Gamma \sim E$

(5) $TT \vdash TS \sim S$

(6) $X \sim (TT, DT)$

Then there exist $\Delta, \Delta', TS', \tau'$ *such that*

(a) $TT' \vdash \langle \Delta, e' \rangle \Rightarrow \langle \Delta', \tau' \rangle$

(b) $\tau' \leq TS'$

(c) If $S = S'$ *then* $\Delta' \leq \Gamma'$

(d) $\Delta \sim E'$

(e) $TT' \vdash TS' \sim S'$

(f) $X' \sim (TT', DT')$

Let's step through the assumptions and conclusions of the theorem. (1) and (2) are standard—they assume that e takes a step and is well-typed, respectively. The corresponding conclusion (a) states that e' is also well-typed.

(4) assumes the type and dynamic environments are consistent—meaning values in E have the corresponding types in Γ—and conclusion (d) states that they are still consistent after reduction. Formally:

Definition 3 (Environment consistency)**.** *Type environment* Γ *is* consistent *with dynamic environment* E*, written* $\Gamma \sim E$*, if* $dom(\Gamma) \subseteq dom(E)$ *and for all* $x \in dom(\Gamma)$ *there exists* τ *such that* $\cdot \vdash \langle \Gamma, E(x) \rangle \Rightarrow \langle \Gamma, \tau \rangle$ *and* $\tau \leq \Gamma(x)$*.*

Notice this definition allows E to include some variables that are not bound in Γ. This is necessary to handle (TIf), which discards any variables from the type environment that are bound in one arm of the conditional but not the other.

Next, (3) and (5) concern the type of e and the stack. The goal of preservation is to show e's type is preserved, but consider (EApp*) and (ERet). These rules both push and pop the stack and change the expression being evaluated—hence e' could potentially have an entirely different type than e.

Our solution is to introduce the notion of a *type stack* TS to mirror the run-time stack. To understand how the type stack works, suppose we want to apply preservation to $C[v_1.m(v_2)]$, i.e., we are about to call a method. The typing judgment is $TT \vdash \langle \Gamma, C[v_1.m(v_2)\rangle \Rightarrow \langle \Gamma', \tau'\rangle$. In the dynamic semantics, the (EApp*) rules will push the current environment E and the context C on the stack. Correspondingly, we will push the current typing judgment onto the type stack—at least the key pieces of it. More specifically, we push an element of the form $(\Gamma[\tau], \langle \Gamma', \tau'\rangle)$, where Γ and Γ' are the initial and final environments of the current typing judgment; C is the context; and τ is the type of expression $v_1.m(v_2)$, i.e., the type that the method must return.

Given this mechanism, the key invariant to maintain is that the type of the expression is compatible with what the calling functions expects. We define:

Definition 4 (Stack subtyping)**.** $\tau_0 \leq (\Gamma[\tau], \langle \Gamma', \tau'\rangle) :: TS$ *if* $\tau_0 \leq \tau$.

Then (3) assumes that the type of e is a subtype of the type expected by the calling function. (At the top-level, we initialize the type stack with a frame that expects whatever the top-level type is.) (b) states that the type of e' is also a subtype of the type expected by its calling function. Thus, if the stack does not change, this means that e' and e have the same type (up to subtyping). If the stack

does change, then we still maintain the invariant.

Of course, we need this invariant to hold no matter how many pushes and pops happen. Thus, rather than only talk about the top element of the type stack, we need to ensure that all elements of the type stack are consistent with all elements of the dynamic stack. Formally:

Definition 5 (Stack consistency)**.** *Type stack element* $(\Gamma[\tau], \langle \Gamma', \tau' \rangle)$ *is* consistent *with dynamic stack element* (E, C)*, written* $TT \vdash (\Gamma[\tau], \langle \Gamma', \tau' \rangle) \sim (E, C)$*, if* $\Gamma \sim E$ *and* $TT \vdash \langle \Gamma[\Box \mapsto \tau], C \rangle \Rightarrow \langle \Gamma', \tau' \rangle$*. (Here we abuse notation and treat* $\Box$ *as if it's a variable.)*

Type stack TS *is* consistent *with dynamic stack* S*, written* $TT \vdash TS \sim S$*, is defined inductively as*

1. $TT \vdash \cdot \sim \cdot$
2. $TT \vdash (\Gamma[\tau], \langle \Gamma', \tau' \rangle) :: TS \sim (E, C) :: S$ *if*
   (a) $(\Gamma[\tau], \langle \Gamma', \tau' \rangle) \sim (E, C)$
   (b) $TS \sim S$
   (c) $\tau' \leq TS$ *if* $TS \neq \cdot$

Thus, (5) assumes the type and dynamic stacks are consistent, and (e) concludes they remain consistent after taking a step.

Next, (c) relates the output environment of e' with the output environment of e.There are two cases. If the stack did not change (the antecedent of the conclusion

is true), then the output environment of e' should be compatible with Γ'. Again because of (TIf), we need to allow the environment to shrink:

Definition 6 (Type environment subsumption)**.** *We write* $\Gamma_1 \leq \Gamma_2$ *if* $dom(\Gamma_2) \subseteq dom(\Gamma_1)$ *and for all* $x \in dom(\Gamma_2)$*, it is the case that* $\Gamma_1(x) \leq \Gamma_2(x)$*.*

If the stack does change, then the output environment is irrelevant: It either is captured in the type stack if this is a push due to a method call. Or it is discarded as the stack frame is popped when a method returns. Hence in this case the antecedent of (c) is false, and the conclusion is trivial.

Finally, we need to reason about the cache. As we saw earlier, the key cache invariant to preserve is that all the derivations stored in the cache hold and apply to the premethod stored in DT and the type stored in TT. Formally:

Definition 7 (Cache consistency)**.** *We say that cache* X *is* consistent *with type class table* TT *and dynamic class table* DT*, written* $X \sim (TT, DT)$*, if for all* $A.m \in dom(X)$ *where* $X(A.m) = (\mathcal{D}_M, \mathcal{D}_{\leq})$*, with* $\mathcal{D}_M = (TT \vdash \langle [x \mapsto \tau_1, \mathsf{self} \mapsto A], e \rangle \Rightarrow \langle \Gamma', \tau \rangle)$ *and* $\mathcal{D}_{\leq} = (\tau \leq \tau_2)$*, it is the case that* $\mathcal{D}_M$ *and* $\mathcal{D}_{\leq}$ *hold and* $DT(A.m) = \lambda x.e$ *and* $TT(A.m) = \tau_1 \to \tau_2$*.*

Thus, (6) assumes the cache is consistent, and (f) concludes the new cache is also consistent.

To show preservation, we also need a few lemmas:

Lemma 1. *For all* Γ_1 *and* Γ_2*, it is the case that* $\Gamma_1 \leq (\Gamma_1 \sqcup \Gamma_2)$*.*

Lemma 2 (Contextual Substitution)**.** *If*

$$\frac{\begin{array}{c} TT \vdash \langle \Gamma, e\rangle \Rightarrow \langle \Gamma, \tau'\rangle \\ \vdots \end{array}}{TT \vdash \langle \Gamma_C, C[e]\rangle \Rightarrow \langle \Gamma'_C, \tau_C\rangle}$$

then $TT \vdash \langle \Gamma_C[\Box \mapsto \tau'], C\rangle \Rightarrow \langle \Gamma'_C, \tau_C\rangle$.

Lemma 3 (Substitution)**.** *If*

1. $TT \vdash \langle \Delta[\Box \mapsto \tau_C], C\rangle \Rightarrow \langle \Delta', \tau'_C\rangle$
2. $TT \vdash \langle \cdot, v\rangle \Rightarrow \langle \cdot, \tau\rangle$
3. $\tau \leq \tau_C$

Then $TT \vdash \langle \Delta, C[v]\rangle \Rightarrow \langle \Delta', \tau''_C\rangle$ *where* $\tau''_C \leq \tau'_C$.

Finally, we can prove preservation:

Proof. **(Preservation)** By induction on $\langle X, TT, DT, E, e, S\rangle \rightarrow \langle X', TT', DT', E', e', S'\rangle$.

- Case (EContext). Notice that we cannot have $S' \neq S$, since the only cases where that can happen is if (EApp) or (ERet) apply, and they cannot be used as a hypothesis of (EContext). Thus the left-hand side of the implication (c) is true, and we have $\Delta' \leq \Gamma'$. Using this fact, the remainder of the proof is routine.
- Case (ESelf). By assumption we have

(1) $\langle X, TT, DT, E, \mathsf{self}, S\rangle \to \langle X, TT, DT, E, E(\mathsf{self}), S\rangle$ by (ESelf)

(2) $TT \vdash \langle \Gamma, \mathsf{self}\rangle \Rightarrow \langle \Gamma, \Gamma(\mathsf{self})\rangle$ by (TSelf)

(3) $\Gamma(\mathsf{self}) \leq TS$

(4) $\Gamma \sim E$

(5) $TT \vdash TS \sim S$

(6) $X \sim (TT, DT)$

Let $\Delta = \Delta' = \Gamma$, and let $TS' = TS$. By (2) and (4) there exists τ' such that $\cdot \vdash \langle \Delta, E(\mathsf{self})\rangle \Rightarrow \langle \Delta, \tau'\rangle$ and $\tau' \leq \Delta(\mathsf{self})$. Then (a) holds, since typing of $E(\mathsf{self})$ was by (TNil) or (TObject), which do not depend of the type class table. Also, (b) holds since $\tau' \leq \Delta(\mathsf{self}) = \Gamma(\mathsf{self}) \leq TS$ by (3). Also, the right-hand side of the implication (c) holds trivially. Finally, (d) holds by (4), (e) holds by (5), and (f) holds by (6).

- Case (EVar). Similar to (ESelf) case.

- Case (EAssn). By assumption we have

(1) $\langle X, TT, DT, E, x = v, S\rangle \to \langle X, TT, DT, E[x \mapsto v], v, S\rangle$

(2)

$$\frac{TT \vdash \langle \Gamma, v\rangle \Rightarrow \langle \Gamma, \tau\rangle}{TT \vdash \langle \Gamma, x = v\rangle \Rightarrow \langle \Gamma[x \mapsto \tau], \tau\rangle}$$

by (TAssn) and either (TNil) or (TObject)

(3) $\tau \leq TS$

(4) $\Gamma \sim E$

(5) $TT \vdash TS \sim S$

(6) $X \sim (TT, DT)$

Let $\Delta = \Delta' = \Gamma[x \mapsto \tau]$, let $TS' = TS$, and let $\tau' = \tau$. Notice that in (2), the hypothesis can only be proven by either (TNil) or (TObject), both of which are insensitive to the type environment. Thus, by the hypothesis of (2), we also have $TT \vdash \langle \Delta, v \rangle \Rightarrow \langle \Delta, \tau \rangle$, which is (a). Also, (b), (e), and (f) hold trivially by (3), (5), and (6). Also, the right-hand side of the implication (c) holds trivially. Finally, from (4) and the hypothesis of (2) we have $\Delta = \Gamma[x \mapsto \tau] \sim E[x \mapsto v]$, which is (d).

- Case (ENew). Trivial.

- Case (ESeq). Trivial.

- Case (EIfTrue). By assumption we have

  (1) $\langle X, TT, DT, E, \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, S \rangle \rightarrow$

  $\langle X, TT, DT, E, e_1, S \rangle$ where $v \neq \mathsf{nil}$

  (2)

$$\frac{TT \vdash \langle \Gamma, v \rangle \Rightarrow \langle \Gamma, \tau \rangle \qquad TT \vdash \langle \Gamma, e_1 \rangle \Rightarrow \langle \Gamma_1, \tau_1 \rangle \qquad TT \vdash \langle \Gamma, e_2 \rangle \Rightarrow \langle \Gamma_2, \tau_2 \rangle}{TT \vdash \langle \Gamma, \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rangle \Rightarrow \langle \Gamma_1 \sqcup \Gamma_2, \tau_1 \sqcup \tau_2 \rangle}$$

  by (TIf) and (TObject), since $v \neq \mathsf{nil}$

(3) $\tau_1 \sqcup \tau_2 \leq TS$

(4) $\Gamma \sim E$

(5) $TT \vdash TS \sim S$

(6) $X \sim (TT, DT)$

Let $\Delta = \Gamma$, let $\Delta' = \Gamma_1$, let $TS' = TS$, and let $\tau' = \tau_1$. From the second hypothesis of (2) we trivially have (a). Moreover, $\tau' = \tau_1 \leq \tau_1 \sqcup \tau_2$, so by (3) we have $\tau' \leq TS$, which is (b). By (4), (5), and (6) we trivially have (d), (e), and (f). Finally, by Lemma 1 we have $\Delta' = \Gamma_1 \leq (\Gamma_1 \sqcup \Gamma_2)$, which is the right-hand side of the implication (c).

- Case (EIfFalse). Similar to (EIfTrue) case.

- Case (EDef). By assumption we have

  (1) $\langle X, TT, DT, E, \mathsf{def}\ A.m = \lambda x.e, S\rangle \rightarrow$

  $\langle X \backslash A.m, TT, DT[A.m \mapsto \lambda x.e], E, \mathsf{nil}, S\rangle$ by (EDef)

  (2) $TT \vdash \langle \Gamma, \mathsf{def}\ A.m = \lambda x.e\rangle \Rightarrow \langle \Gamma, \mathsf{nil}\rangle$ by (TDef)

  (3) $\mathsf{nil} \leq TS$

  (4) $\Gamma \sim E$

  (5) $TT \vdash TS \sim S$

  (6) $X \sim (TT, DT)$

  Let $\Delta = \Delta' = \Gamma$, let $TS' = TS$, and let $\tau' = \mathsf{nil}$. Then (a) holds trivially by (TNil). (c) holds trivially by definition. (b), (d), and (e) hold trivially by (3),

(4), and (5).

For (f), pick some $B.m' \in dom(X \backslash A.m)$. Observe that $B.m' \neq A.m$, by construction. By (6), we have

1. $\mathcal{D}_M = (TT \vdash \langle [y \mapsto \tau_y, \mathsf{self} \mapsto B], e \rangle \Rightarrow \langle \Gamma'_y, \tau'_y \rangle)$
2. $\mathcal{D}_{\leq} = (\tau'_y \leq \tau_2)$
3. $\mathcal{D}_M$ and $\mathcal{D}_{\leq}$ hold
4. $DT(B.m') = \lambda y.e'$
5. $TT(B.m') = \tau_y \to \tau_2$

We need to show the above with the same type class table and with dynamic class table $DT[A.m \mapsto \lambda x.e]$. But then 1, 2, 3, and 5 are trivial, and since $B.m' \neq A.m$ we have $(DT[A.m \mapsto \lambda x.e])(B.m') = DT(B.m')$, thus 4 is trivial.

- Case (EType). This case is very similar to (EDef), except the reduction in the semantics is different. By assumption, we have

  (1) $\langle X, TT, DT, E, \mathsf{type}\ A.m : \tau_m, S \rangle \to$
  $\langle (X \backslash A.m)[TT'], TT', DT, E, \mathsf{nil}, S \rangle$ where $TT' = TT[A.m \mapsto \tau_m]$, by (EType)

  (2) $TT \vdash \langle \Gamma, \mathsf{def}\ A.m = \lambda x.e \rangle \Rightarrow \langle \Gamma, \mathsf{nil} \rangle$ by (TDef)

  (3) $\mathsf{nil} \leq TS$

  (4) $\Gamma \sim E$

  (5) $TT \vdash TS \sim S$

(6) $X \sim (TT, DT)$

(a)–(d) hold by the same reasoning above. To see (e), observe that side condition $A.m \notin \text{TApp}(S)$ means that in the typing judgments internal to (5), (TApp) is never applied with $A.m$. Hence those same judgments hold under TT', which only differs from TT in its binding or $A.m$.

Let $X' = (X \backslash A.m)[TT']$. For (f), again pick some $B.m' \in dom(X')$. Observe that $B.m' \neq A.m$, by construction. By (6), we have

1. $\mathcal{D}_M = (TT \vdash \langle [y \mapsto \tau_y, \mathsf{self} \mapsto B], e \rangle \Rightarrow \langle \Gamma'_y, \tau'_y \rangle)$
2. $\mathcal{D}_{\leq} = (\tau'_y \leq \tau_2)$
3. $\mathcal{D}_M$ and $\mathcal{D}_{\leq}$ hold
4. $DT(B.m') = \lambda y.e'$
5. $TT(B.m') = \tau_y \to \tau_2$

We need to show the above, but in X' and with type class table TT' and the same dynamic class table. By construction, $X'(B.m') = (\mathcal{D}'_M, \mathcal{D}_{\leq})$ where $\mathcal{D}'_M = (TT' \vdash \langle [y \mapsto \tau_y, \mathsf{self} \mapsto B], e \rangle \Rightarrow \langle \Gamma'_y, \tau'_y \rangle)$, which is 1 and 2. Notice by construction that $\mathcal{D}_M$ and $\mathcal{D}'_M$ cannot refer to $A.m$, thus we have 3. Finally, 4 holds trivially, and 5 holds since $B.m' \neq A.m$ by construction.

- Case (EAppMiss). The inductive cases are similar to (EContext). In the non-inductive case, by assumption we have

  (1) $\langle X, TT, DT, E, C[[A].m(v_2)], S \rangle \to$

$\langle X[A.m \mapsto (\mathcal{D}_M, \mathcal{D}_{\leq})], TT, DT, [\mathsf{self} \mapsto [A], x \mapsto v_2], e, (E, C) :: S\rangle$

where

(1a) $DT(A.m) = \lambda x.e$

(1b) $TT(A.m) = \tau_1 \to \tau_2$

(1c) $\mathsf{type\_of}(v_2) \leq \tau_1$

(1d) $\mathcal{D}_M = (TT \vdash \langle [x \mapsto \tau_1, \mathsf{self} \mapsto A], e\rangle \Rightarrow \langle \Gamma', \tau_2'\rangle)$ holds

(1e) $\mathcal{D}_{\leq} = (\tau_2' \leq \tau_2)$ holds

(1f) $A.m \notin dom(X)$

(2)

$$\frac{\dfrac{\dfrac{}{TT \vdash \langle \Gamma, [A]\rangle \Rightarrow \langle \Gamma, A\rangle} \quad \dfrac{}{TT \vdash \langle \Gamma, v_2\rangle \Rightarrow \langle \Gamma, \tau\rangle} \quad TT(A.m) = \tau_1 \to \tau_2 \quad \tau \leq \tau_1}{TT \vdash \langle \Gamma, [A].m(v_2)\rangle \Rightarrow \langle \Gamma, \tau_2\rangle} \quad \vdots}{TT \vdash \langle \Gamma_C, C[[A].m(v_2)]\rangle \Rightarrow \langle \Gamma'_C, \tau_C\rangle}$$

by (TApp) and (TObject) and possible (TNil).

(3) $\tau_C \leq TS$

(4) $\Gamma_C \sim E_C$

(5) $TT \vdash TS \sim S$

(6) $X \sim (TT, DT)$

Let $\Delta = [x \mapsto \tau_1, \mathsf{self} \mapsto A]$, let $\Delta' = \Gamma'$, let $TS' = (\Gamma_C[\tau_2], \langle \Gamma'_C, \tau_C\rangle) :: TS$,

and let $\tau' = \tau_2'$. Then (a) holds immediately by (1d). (b) holds by (1e) and construction of TS'. In this case the stack changes, so the left-hand side of the implication (c) is false, hence (c) holds trivially. (d) holds because by (TObject) we have $[A]$ has type A, and by the second hypothesis of (2), which is either (TObject) or (TNil), we have v_2 has type τ, and by the last hypothesis of (2) we have $\tau \leq \tau_1$.

Next we show (e). By (4) we have $\Gamma_C \sim E$, and by (2) and the Contextual Substitution Lemma we have $TT \vdash \langle \Gamma_C[\Box \mapsto \tau_2], C\rangle \Rightarrow \langle \Gamma_C', \tau_C\rangle$. Thus we have $TT \vdash (\Gamma_C[\tau_2], \langle \Gamma_C', \tau_C\rangle) \sim (E, C)$. Further, by (3) we have $\tau_C \leq TS$. Finally, by (5) we have $TT \vdash TS \sim S$. Putting this all together, we have $TT \vdash (\Gamma_C[\tau_2], \langle \Gamma_C', \tau_C\rangle) \sim (E, C) :: TS \sim (E, C) :: S$, which is (e).

Finally, to show (f), pick some element in the domain of $X' = X[A.m \mapsto (\mathcal{D}_M, \mathcal{D}_\leq)]$. If we pick some $B.m' \neq A.m$ then all the necessary properties hold by (6). If we pick $A.m$, then 1 and 2 hold by construction, 3 holds by (1d) and (1e), 4 holds by (1a), and 5 holds by (1b).

- Case (EAppHit). This case follows mostly the same reasoning as above. The inductive cases are similar to (EContext). In the non-inductive case, by assumption we have

  (1) $\langle X, TT, DT, E, C[[A].m(v_2)], S\rangle \rightarrow$

  $\langle X, TT, DT, [\mathsf{self} \mapsto [A], x \mapsto v_2], e, (E, C) :: S\rangle$ where

  (1a) $DT(A.m) = \lambda x.e$

  (1b) $TT(A.m) = \tau_1 \rightarrow \tau_2$

(1c) $\mathsf{type\_of}(v_2) \leq \tau_1$

(1d) $A.m \in dom(X)$

(2)

$$\frac{\begin{array}{c} TT \vdash \langle \Gamma, [A] \rangle \Rightarrow \langle \Gamma, A \rangle \\ TT \vdash \langle \Gamma, v_2 \rangle \Rightarrow \langle \Gamma, \tau \rangle \\ \dfrac{TT(A.m) = \tau_1 \to \tau_2 \qquad \tau \leq \tau_1}{TT \vdash \langle \Gamma, [A].m(v_2) \rangle \Rightarrow \langle \Gamma, \tau_2 \rangle} \\ \vdots \end{array}}{TT \vdash \langle \Gamma_C, C[[A].m(v_2)] \rangle \Rightarrow \langle \Gamma'_C, \tau_C \rangle}$$

by (TApp) and (TObject) and possible (TNil).

(3) $\tau_C \leq TS$

(4) $\Gamma_C \sim E_C$

(5) $TT \vdash TS \sim S$

(6) $X \sim (TT, DT)$

By (6), we have $X(A.m) = (\mathcal{D}_M, \mathcal{D}_{\leq})$ where $\mathcal{D}_M = (TT \vdash \langle [x \mapsto \tau_1, \mathsf{self} \mapsto A], e \rangle \Rightarrow \langle \Gamma', \tau'_2 \rangle)$ holds and $\mathcal{D}_{\leq} = (\tau'_2 \leq \tau_2)$ holds. Notice that we use properties 4 and 5 of the cache in combination with (1a) and (1b) to know the assigned types in the cache, and the method body at run-time, match in the cached derivation.

Let $\Delta = [x \mapsto \tau_1, \mathsf{self} \mapsto A]$, let $\Delta' = \Gamma'$, let $TS' = (\Gamma_C[\tau_2], \langle \Gamma'_C, \tau_C \rangle) :: TS$, and let $\tau' = \tau'_2$. Then (a) holds immediately by $\mathcal{D}_M$. (b) holds by $\mathcal{D}_{\leq}$ and

construction of TS'.

The reasoning for (c)–(e) are the same as the (EAppMiss) case. Finally, (f) holds trivially by (6), since the cache did not change.

- Case (ERet). We have

  (1) $\langle X, TT, DT, E', v, (E, C) :: S\rangle \rightarrow$

  $\langle X, TT, DT, E, C[v], S\rangle$

  (2) $TT \vdash \langle \Gamma', v\rangle \Rightarrow \langle \Gamma', \tau\rangle$ by either (TObject) or (TNil).

  (3) $\tau \leq \tau_C$

  (4) $\Gamma' \sim E'$

  (5) $TT \vdash (\Gamma_C[\tau_C], \langle \Gamma'_C, \tau'_C\rangle) :: TS \sim (E, C) :: S$

  (6) $X \sim (TT, DT)$

  Let $\Delta = \Gamma_C$, let $\Delta' = \Gamma'_C$, and let $TS' = TS$. By (5), we have $TT \vdash \langle \Delta[\Box \mapsto \tau_C], C\rangle \Rightarrow \langle \Delta', \tau'_C\rangle$. Putting that together with (2) and (3) via the substitution lemma, we have $TT \vdash \langle \Delta, C[v]\rangle \Rightarrow \langle \Delta', \tau''_C\rangle$ where $\tau''_C \leq \tau'_C$. Let $\tau' = \tau''_C$, and we have (a). By (3) we have $\tau'_C \leq TS$, and since $\tau''_C \leq \tau'_C$ we therefore have (b) In this case the stack changes, so the left-hand side of the implication (c) is false, hence (c) holds trivially. (d) holds by (5), as does (e). Finally, (f) holds by (6)

□

The progress theorem is much simpler:

Theorem 3 (Progress)**.** *If*

(1) $TT \vdash \langle \Gamma, e \rangle \Rightarrow \langle \Gamma', \tau \rangle$

(2) $\tau \leq TS$

(3) $\Gamma \sim E$

(4) $TT \vdash TS \sim S$

(5) $X \sim (TT, DT)$

then one of the following holds

1. *e is a value, or*
2. *There exist* X', TT', DT', E', e', S' *such that*

   $\langle X, TT, DT, E, e, S \rangle \rightarrow \langle X', TT', DT', E', e', S' \rangle$*, or*
3. $\langle X, TT, DT, E, e, S \rangle \rightarrow$ *blame*

Proof. By induction on e.

- Case $e = \mathsf{nil}$ or $e = [A]$. These are values, so the theorem holds trivially.
- Case self. By assumption (1) we have

$$\frac{}{TT \vdash \langle \Gamma, \mathsf{self} \rangle \Rightarrow \langle \Gamma, \Gamma(\mathsf{self}) \rangle}$$

  Thus, $\mathsf{self} \in dom(\Gamma)$. But then by (3), $\mathsf{self} \in dom(E)$. Thus (ESelf) can be applied.

(Note that assuming we start executing the program in a standard environment, self will in fact always be bound in all type and dynamic environments, unlike variables.)

- Case x. Similar to self.

- Case $x = v$, $A.\mathsf{new}$, $v; e$, $\mathsf{def}\ A.m = (\lambda x.e)$, $\mathsf{type}\ A.m : \tau_m$. These cases are trivial, as there is one semantics rule for each of these forms, and it will always be able to take a step.

- Case $\mathsf{if}\ v\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$. This case is trivial, since either (EIfTrue) or (EIfFalse) will apply.

- Case $v_0.m(v_1)$. By assumption (1) we have

$$\frac{TT \vdash \langle \Gamma, v_0 \rangle \Rightarrow \langle \Gamma_0, A \rangle \qquad TT \vdash \langle \Gamma_0, v_1 \rangle \Rightarrow \langle \Gamma_1, \tau \rangle \qquad TT(A.m) = \tau_1 \to \tau_2 \qquad \tau \leq \tau_1}{TT \vdash \langle \Gamma, v_0.m(v_1) \rangle \Rightarrow \langle \Gamma_1, \tau_2 \rangle}$$

There are a few cases. If (EAppNil), (EAppNExist), or (EAppNTyp) apply, then the theorem holds trivially. Otherwise, we must have $v_1 = [A]$ and $DT(A.m) = \lambda x.e$. More importantly, by (1) we have $\mathsf{type\_of}(v_1) \leq \tau_1$, since $\tau = \mathsf{type\_of}(v_1)$ by (1), i.e., v_1 has the expected argument type. Also by (1) we have $TT(A.m) = \tau_1 \to \tau_2$.

Now there are two cases. If $A.m \in dom(X)$ we can immediately apply (EAp-

pHit). Otherwise, if $A.m \notin dom(X)$, then we must have

$$\mathcal{D}_M = (TT \vdash \langle [x \mapsto \tau_1, \mathsf{self} \mapsto A], e \rangle \Rightarrow \langle \Gamma', \tau \rangle)$$

holds and $\mathcal{D}_{\leq} = (\tau \leq \tau_2)$ holds because (EAppNTyp) did not apply. But then combining this with our previous assumptions we can apply (EAppMiss).

- Else $e = C[e']$. Holds by induction and (EContext).

□

Finally, we can put these together to prove soundness.

Theorem 4 (Soundness)**.** *If* $\emptyset \vdash \langle \emptyset, e \rangle \Rightarrow \langle \Gamma', \tau \rangle$ *then either* e *reduces to a value,* e *reduces to blame, or* e *does not terminate.*

Proof. Let $X = \emptyset$, let $TT = \emptyset$, let $\Gamma = \emptyset$, let $E = \emptyset$, let $DT = \emptyset$, let $S = (\emptyset, \square) :: \cdot$, and let $TS = (\emptyset[\tau], \langle \emptyset, \tau \rangle)$. Then by assumption we have $TT \vdash \langle \Gamma, e \rangle \Rightarrow \langle \Gamma', \tau \rangle$. By construction we have $\tau \leq TS$ and $\Gamma \sim E$ and $TT \vdash TS \sim S$ and $X \sim (TT, DT)$. Thus, these choices of X, TT, Γ, DT, S, and TS satisfy the preconditions of progress and preservation. Thus soundness holds by standard arguments. □

Bibliography

[1] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic Inference of Static Types for Ruby. In *POPL*, 2011.

[2] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Position Paper: Dynamically Inferred Types for Dynamic Languages. In *STOP*, 2011.

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

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

[5] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: Robust, Concurrent, Extensible Scripting on the JVM. In *OOPSLA*, 2009.

[6] Robert Bruce Findler and Matthias Felleisen. Contracts for Higher-Order Functions. In *ICFP*, 2002.

[7] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-Guided Static Typing for Dynamic Scripting Languages. In *OOPSLA*, 2009.

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

[9] Bertrand Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, October 1992.

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

[11] Jeremy G. Siek and Walid Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, 2006.

[12] T. Stephen Strickland and Matthias Felleisen. Contracts for First-Class Classes. In *DLS*, 2010.

[13] Peter Thiemann. Towards a Type System for Analyzing JavaScript Programs. In *ESOP*, 2005.

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

[15] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, pages 53–64, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-868-8.

[16] G. Bierman, M. Abadi, and M. Torgersen. Understanding typescript. In *ECOOP 2014–Object-Oriented Programming*, pages 257–281. Springer, 2014.

[17] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged Information Flow for Javascript. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 50–62, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1.

[18] Dart. Dart, 2015. `https://www.dartlang.org`.

[19] DRuby. Diamondback Ruby, 2009. `http://www.cs.umd.edu/projects/PL/druby/`.

[20] F #. Type Provider, 2016. `https://msdn.microsoft.com/en-us/library/hh156509.aspx`.

[21] A. Feldthaus and A. Møller. Checking Correctness of TypeScript Interfaces for JavaScript Libraries. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 1–16, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1.

[22] M. Furr, J. hoon (David) An, J. S. Foster, and M. Hicks. The Ruby Intermediate Language. In *Dynamic Languages Symposium (DLS)*, pages 89–98, Orlando, Florida, October 2009.

[23] GHCLanguageFeatures. Deferring Type Errors to Runtime, 2016. `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/defer-type-errors.html`.

[24] M. Hermenegildo, G. Puebla, K. Marriott, and P. J. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Trans. Program. Lang. Syst.*, 22 (2):187–223, Mar. 2000. ISSN 0164-0925.

[25] J. hoon (David) An, A. Chaudhuri, and J. S. Foster. Static Typing for Ruby on Rails. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 590–594, Auckland, New Zealand, November 2009. Short paper.

[26] E. Koukoutos and V. Kuncak. Checking Data Structure Properties Orders of Magnitude Faster. In *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, pages 263–268, 2014.

[27] B. S. Lerner, L. Elberty, J. Li, and S. Krishnamurthi. Combining Form and Function: Static Types for JQuery Programs. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 79–103, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-39037-1.

[28] B. S. Lerner, J. G. Politz, A. Guha, and S. Krishnamurthi. TeJaS: Retrofitting Type Systems for JavaScript. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 1–16, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2433-5.

[29] A. M. Maidl, F. Mascarenhas, and R. Ierusalimschy. Typed Lua: An Optional Type System for Lua. In *Proceedings of the Workshop on Dynamic Languages and Applications*, Dyla'14, pages 3:1–3:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2916-3.

[30] RailsGuides. Active Record Associations, 2015. `http://guides.rubyonrails.org/association_basics.html`.

[31] RailsGuides. Autoloading and Reloading Constants, 2016. `http://guides.rubyonrails.org/autoloading_and_reloading_constants.html`.

[32] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 167–180, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9.

[33] RDL. RDL, 2015. `https://github.com/plum-umd/rdl`.

[34] B. Ren and J. S. Foster. Just-in-Time Static Type Checking for Dynamic Languages, 2016. preprint, `http://arxiv.org/abs/1604.03641`.

[35] B. M. Ren and J. S. Foster. Just-in-time static type checking for dynamic languages. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 462–476, New York, NY, USA, 2016. ACM.

[36] B. M. Ren, J. Toman, T. S. Strickland, and J. S. Foster. The Ruby Type Checker. In *Object-Oriented Program Languages and Systems (OOPS) Track at ACM Symposium on Applied Computing*, pages 1565–1572, Coimbra, Portugal, March 2013.

[37] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3.

[38] G. Richards, C. Hammer, B. Burg, and J. Vitek. The Eval That Men Do: A Large-scale Study of the Use of Eval in Javascript Applications. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22654-0.

[39] T. Sheard and S. P. Jones. Template Meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, Dec. 2002. ISSN 0362-1340.

[40] V. St-Amour, S. Tobin-Hochstadt, M. Flatt, and M. Felleisen. Typing the Numeric Tower. In *Proceedings of the 14th International Conference on Practical Aspects of Declarative Languages*, PADL'12, pages 289–303, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-27693-4.

[41] T. S. Strickland, B. Ren, and J. S. Foster. Contracts for Domain-Specific Languages in Ruby. In *Dynamic Languages Symposium (DLS)*, Portland, OR, October 2014.

[42] N. Stulova, J. F. Morales, and M. V. Hermenegildo. Practical Run-time Checking via Unobtrusive Property Caching. *CoRR*, abs/1507.05986, 2015.

[43] W. Taha. MetaOcaml, 2016. `http://www.cs.rice.edu/~taha/MetaOCaml`.

[44] S. Tobin-Hochstadt and M. Felleisen. Logical Types for Untyped Languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 117–128, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3.

[45] M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker. Design and Evaluation of Gradual Typing for Python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 45–56, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3211-8.

[46] D. A. Wheeler. SLOCCount, 2015. `http://www.dwheeler.com/sloccount`.

[47] T. Wrigstad, F. Z. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating Typed and Untyped Code in a Scripting Language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Languages, POPL '10, pages 377–388, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9.

[48] Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. Dynamic inference of abstract types. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 255–265, New York, NY, USA, 2006. ACM.

[49] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '92, pages 329–349, Berlin, Heidelberg, 1992. Springer-Verlag.

[50] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 146–161, New York, NY, USA, 1991. ACM.

[51] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 2–26, Berlin, Heidelberg, 1995. Springer-Verlag.

[52] Martin Madsen, Peter Sørensen, and Kristian Kristensen. Ecstatic – Type Inference for Ruby Using the Cartesian Product Algorithm. Master's thesis, Aalborg University, 2007.

[53] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 278–292, New York, NY, USA, 1991. ACM.

[54] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for javascript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 239–250, New York, NY, USA, 2012. ACM.

[55] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 163–173, New York, NY, USA, 1994. ACM.

[56] Nevin Heintze. Set Based Program Analysis. PhD Dissertation, Carnegie Mellon University, 1992.